

Data Warehouse Service

# SQL Syntax Reference

Issue 01  
Date 2020-11-10



**Copyright © Huawei Technologies Co., Ltd. 2020. All rights reserved.**

No part of this document may be reproduced or transmitted in any form or by any means without prior written consent of Huawei Technologies Co., Ltd.

## **Trademarks and Permissions**



HUAWEI and other Huawei trademarks are trademarks of Huawei Technologies Co., Ltd.

All other trademarks and trade names mentioned in this document are the property of their respective holders.

## **Notice**

The purchased products, services and features are stipulated by the contract made between Huawei and the customer. All or part of the products, services and features described in this document may not be within the purchase scope or the usage scope. Unless otherwise specified in the contract, all statements, information, and recommendations in this document are provided "AS IS" without warranties, guarantees or representations of any kind, either express or implied.

The information in this document is subject to change without notice. Every effort has been made in the preparation of this document to ensure accuracy of the contents, but all statements, information, and recommendations in this document do not constitute a warranty of any kind, express or implied.

---

# Contents

---

<b>1 GaussDB(DWS) SQL.....</b>	<b>1</b>
<b>2 Differences Between GaussDB(DWS) and Postgres SQL.....</b>	<b>2</b>
2.1 GaussDB(DWS), PostgreSQL psql, and libpq.....	2
2.2 Data Type Differences.....	3
2.3 Function Differences.....	3
2.4 PostgreSQL Features Unsupported by GaussDB(DWS).....	3
<b>3 Keyword.....</b>	<b>5</b>
<b>4 Data Types.....</b>	<b>34</b>
4.1 Numeral.....	34
4.2 Monetary Types.....	39
4.3 Boolean Type.....	40
4.4 Character Types.....	41
4.5 Binary Data Types.....	43
4.6 Date/Time Types.....	44
4.7 Geometric Types.....	52
4.8 Network Address Types.....	54
4.9 Bit String Types.....	56
4.10 Text Search Types.....	56
4.11 UUID Type.....	59
4.12 JSON Types.....	60
4.13 HLL Data Types.....	60
4.14 Object Identifier Types.....	63
4.15 Pseudo-Types.....	65
4.16 Data Types Supported by Column-Store Tables.....	66
<b>5 Constant and Macro.....</b>	<b>68</b>
<b>6 Functions and Operators.....</b>	<b>70</b>
6.1 Logical Operators.....	70
6.2 Comparison Operators.....	70
6.3 Character Processing Functions and Operators.....	71
6.4 Binary String Functions and Operators.....	90
6.5 Bit String Functions and Operators.....	93

6.6 Pattern Matching Operators.....	94
6.7 Mathematical Functions and Operators.....	99
6.8 Date and Time Processing Functions and Operators.....	108
6.9 Type Conversion Functions.....	122
6.10 Geometric Functions and Operators.....	129
6.11 Network Address Functions and Operators.....	139
6.12 Text Search Functions and Operators.....	145
6.13 UUID Functions.....	151
6.14 JSON Functions.....	152
6.15 HLL Functions and Operators.....	152
6.16 SEQUENCE Functions.....	163
6.17 Array Functions and Operators.....	165
6.18 Range Functions and Operators.....	169
6.19 Aggregate Functions.....	174
6.20 Window Functions.....	184
6.21 Security Functions.....	194
6.22 Set Returning Functions.....	197
6.23 Conditional Expression Functions.....	199
6.24 System Information Functions.....	202
6.25 System Administration Functions.....	217
6.25.1 Configuration Settings Functions.....	217
6.25.2 Universal File Access Functions.....	217
6.25.3 Server Signaling Functions.....	219
6.25.4 Backup and Restoration Control Functions.....	220
6.25.5 Snapshot Synchronization Functions.....	224
6.25.6 Database Object Functions.....	224
6.25.7 Advisory Lock Functions.....	227
6.25.8 Replication Functions.....	229
6.25.9 Other Functions.....	236
6.26 Data Redaction Functions.....	239
6.26.1 Creating a Redact Policy.....	240
6.26.2 Modifying a Redact Policy.....	241
6.26.3 Deleting a Redact Policy.....	241
6.26.4 Enabling the Redact Function for a Table Object.....	242
6.26.5 Disabling the Redact Function for a Table Object.....	242
6.27 Statistics Information Functions.....	242
6.28 Trigger Functions.....	254
<b>7 Expressions.....</b>	<b>256</b>
7.1 Simple Expressions.....	256
7.2 Conditional Expressions.....	257
7.3 Subquery Expressions.....	261
7.4 Array Expressions.....	264

7.5 Row Expressions.....	266
<b>8 Type Conversion.....</b>	<b>267</b>
8.1 Overview.....	267
8.2 Operators.....	269
8.3 Functions.....	271
8.4 Value Storage.....	273
8.5 UNION, CASE, and Related Constructs.....	274
<b>9 Full Text Search.....</b>	<b>277</b>
9.1 Introduction.....	277
9.1.1 Full-Text Retrieval.....	277
9.1.2 What Is a Document?.....	278
9.1.3 Basic Text Matching.....	279
9.1.4 Configurations.....	280
9.2 Table and index.....	280
9.2.1 Searching a Table.....	280
9.2.2 Creating an Index.....	282
9.2.3 Constraints on Index Use.....	283
9.3 Controlling Text Search.....	284
9.3.1 Parsing Documents.....	284
9.3.2 Parsing Queries.....	285
9.3.3 Ranking Search Results.....	286
9.3.4 Highlighting Results.....	288
9.4 Additional Features.....	289
9.4.1 Manipulating tsvector.....	290
9.4.2 Manipulating Queries.....	290
9.4.3 Rewriting Queries.....	291
9.4.4 Gathering Document Statistics.....	292
9.5 <b>Parsers</b> .....	293
9.6 Dictionaries.....	297
9.6.1 Overview.....	297
9.6.2 Stop Words.....	298
9.6.3 Simple Dictionary.....	299
9.6.4 Synonym Dictionary.....	300
9.6.5 Thesaurus Dictionary.....	302
9.6.6 Ispell Dictionary.....	303
9.6.7 Snowball Dictionary.....	304
9.7 Configuration Examples.....	304
9.8 Testing and Debugging Text Search.....	306
9.8.1 Testing a Configuration.....	306
9.8.2 Testing a Parser.....	307
9.8.3 Testing a Dictionary.....	308
9.9 Limitations.....	308

<b>10 System Operation.....</b>	<b>309</b>
<b>11 Controlling Transactions.....</b>	<b>310</b>
<b>12 DDL Syntax.....</b>	<b>311</b>
12.1 DDL Syntax Overview.....	311
12.2 ALTER DATABASE.....	316
12.3 ALTER FOREIGN TABLE (For GDS).....	318
12.4 ALTER FOREIGN TABLE (For HDFS or OBS).....	319
12.5 ALTER FUNCTION.....	321
12.6 ALTER GROUP.....	324
12.7 ALTER INDEX.....	324
12.8 ALTER LARGE OBJECT.....	326
12.9 ALTER NODE.....	327
12.10 ALTER NODE GROUP.....	327
12.11 ALTER RESOURCE POOL.....	329
12.12 ALTER ROLE.....	331
12.13 ALTER ROW LEVEL SECURITY POLICY.....	333
12.14 ALTER SCHEMA.....	334
12.15 ALTER SEQUENCE.....	335
12.16 ALTER SERVER.....	337
12.17 ALTER SESSION.....	339
12.18 ALTER SYNONYM.....	340
12.19 ALTER SYSTEM KILL SESSION.....	341
12.20 ALTER TABLE.....	342
12.21 ALTER TABLE PARTITION.....	351
12.22 ALTER TEXT SEARCH CONFIGURATION.....	355
12.23 ALTER TEXT SEARCH DICTIONARY.....	358
12.24 ALTER TRIGGER.....	359
12.25 ALTER TYPE.....	360
12.26 ALTER USER.....	362
12.27 ALTER VIEW.....	364
12.28 CLEAN CONNECTION.....	366
12.29 CLOSE.....	367
12.30 CLUSTER.....	368
12.31 COMMENT.....	370
12.32 CREATE BARRIER.....	372
12.33 CREATE DATABASE.....	373
12.34 CREATE FOREIGN TABLE (for GDS Import and Export).....	376
12.35 CREATE FOREIGN TABLE (for OBS Import and Export).....	388
12.36 CREATE FOREIGN TABLE (SQL on Hadoop or OBS).....	397
12.37 CREATE FUNCTION.....	409
12.38 CREATE GROUP.....	415
12.39 CREATE INDEX.....	416

12.40 CREATE NODE.....	421
12.41 CREATE NODE GROUP.....	423
12.42 CREATE ROW LEVEL SECURITY POLICY.....	424
12.43 CREATE PROCEDURE.....	427
12.44 CREATE RESOURCE POOL.....	430
12.45 CREATE ROLE.....	432
12.46 CREATE SCHEMA.....	438
12.47 CREATE SEQUENCE.....	440
12.48 CREATE SERVER.....	442
12.49 CREATE SYNONYM.....	444
12.50 CREATE TABLE.....	446
12.51 CREATE TABLE AS.....	463
12.52 CREATE TABLE PARTITION.....	466
12.53 CREATE TEXT SEARCH CONFIGURATION.....	480
12.54 CREATE TEXT SEARCH DICTIONARY.....	483
12.55 CREATE TRIGGER.....	486
12.56 CREATE TYPE.....	492
12.57 CREATE USER.....	500
12.58 CREATE VIEW.....	502
12.59 CURSOR.....	503
12.60 DROP DATABASE.....	504
12.61 DROP FOREIGN TABLE.....	505
12.62 DROP FUNCTION.....	506
12.63 DROP GROUP.....	507
12.64 DROP INDEX.....	508
12.65 DROP NODE.....	509
12.66 DROP NODE GROUP.....	509
12.67 DROP OWNED.....	510
12.68 DROP ROW LEVEL SECURITY POLICY.....	511
12.69 DROP PROCEDURE.....	511
12.70 DROP RESOURCE POOL.....	512
12.71 DROP ROLE.....	513
12.72 DROP SCHEMA.....	513
12.73 DROP SEQUENCE.....	514
12.74 DROP SERVER.....	515
12.75 DROP SYNONYM.....	516
12.76 DROP TABLE.....	517
12.77 DROP TEXT SEARCH CONFIGURATION.....	517
12.78 DROP TEXT SEARCH DICTIONARY.....	518
12.79 DROP TRIGGER.....	519
12.80 DROP TYPE.....	520
12.81 DROP USER.....	521

12.82 DROP VIEW.....	522
12.83 FETCH.....	523
12.84 MOVE.....	526
12.85 REINDEX.....	528
12.86 RESET.....	530
12.87 SET.....	531
12.88 SET CONSTRAINTS.....	533
12.89 SET ROLE.....	534
12.90 SET SESSION AUTHORIZATION.....	535
12.91 SHOW.....	536
12.92 TRUNCATE.....	537
12.93 VACUUM.....	539
<b>13 DML Syntax.....</b>	<b>543</b>
13.1 DML Syntax Overview.....	543
13.2 CALL.....	544
13.3 COPY.....	546
13.4 DELETE.....	559
13.5 EXPLAIN.....	561
13.6 EXPLAIN PLAN.....	565
13.7 LOCK.....	567
13.8 MERGE INTO.....	570
13.9 INSERT.....	573
13.10 SELECT.....	575
13.11 SELECT INTO.....	587
13.12 UPDATE.....	589
13.13 VALUES.....	591
<b>14 DCL Syntax.....</b>	<b>593</b>
14.1 DCL Syntax Overview.....	593
14.2 ALTER DEFAULT PRIVILEGES.....	593
14.3 ANALYZE   ANALYSE.....	595
14.4 DEALLOCATE.....	599
14.5 DO.....	600
14.6 EXECUTE.....	601
14.7 EXECUTE DIRECT.....	602
14.8 GRANT.....	603
14.9 PREPARE.....	609
14.10 REASSIGN OWNED.....	609
14.11 REVOKE.....	610
<b>15 TCL Syntax.....</b>	<b>613</b>
15.1 TCL Syntax Overview.....	613
15.2 ABORT.....	613



15.3 BEGIN.....	614
15.4 CHECKPOINT.....	615
15.5 COMMIT   END.....	616
15.6 COMMIT PREPARED.....	617
15.7 PREPARE TRANSACTION.....	618
15.8 SAVEPOINT.....	619
15.9 SET TRANSACTION.....	620
15.10 START TRANSACTION.....	622
15.11 ROLLBACK.....	623
15.12 RELEASE SAVEPOINT.....	624
15.13 ROLLBACK PREPARED.....	625
15.14 ROLLBACK TO SAVEPOINT.....	625
<b>16 Appendix.....</b>	<b>627</b>
16.1 GIN Indexes.....	627
16.1.1 Introduction.....	627
16.1.2 Scalability.....	627
16.1.3 Implementation.....	630
16.1.4 GIN Tips and Tricks.....	631
16.2 Extended Functions.....	631
16.3 Extended Syntax.....	632

# 1 GaussDB(DWS) SQL

---

## What Is SQL?

SQL is a standard computer language used to control the access to databases and manage data in databases.

SQL provides different statements to enable you to:

- Query data.
- Insert, update, and delete rows.
- Create, replace, modify, and delete objects.
- Control the access to a database and its objects.
- Maintain the consistency and integrity of a database.

SQL consists of commands and functions that are used to manage databases and database objects. SQL can also forcibly implement the rules for data types, expressions, and texts. Therefore, section "SQL Reference" describes data types, expressions, functions, and operators in addition to SQL syntax.

## Development of SQL Standards

Released SQL standards are as follows:

- 1986: ANSI X3.135-1986, ISO/IEC 9075:1986, SQL-86
- 1989: ANSI X3.135-1989, ISO/IEC 9075:1989, SQL-89
- 1992: ANSI X3.135-1992, ISO/IEC 9075:1992, SQL-92 (SQL2)
- 1999: ISO/IEC 9075:1999, SQL:1999 (SQL3)
- 2003: ISO/IEC 9075:2003, SQL:2003 (SQL4)
- 2011: ISO/IEC 9075:200N, SQL:2011 (SQL5)

## SQL Standards Supported by GaussDB(DWS)

GaussDB(DWS) is compatible with Postgres-XC and supports the major features of SQL2, SQL3, and SQL4 by default.

# 2 Differences Between GaussDB(DWS) and Postgres SQL

---

## 2.1 GaussDB(DWS), PostgreSQL psql, and libpq

### GaussDB(DWS) gsql and PostgreSQL psql

GaussDB(DWS) **gsql** differs from PostgreSQL **psql** in that the former has made the following changes to enhance security:

- User passwords cannot be set by running the **\password** command.
- The **\i+**, **\ir+**, and **\include\_relative+** commands and the input and output parameter **-k** are added to encrypt imported and exported files.
- Historical command lines cannot be printed to files using the **\s** metacommand.
- SQL statements related to sensitive operations, such as those containing passwords, are not recorded. Users cannot see such records when they turn pages or press up or down arrow keys to view the SQL history.
- After a connection is set up, a message is displayed to inform users of password expiration and to show version information.

**gsql** also has the following functions in addition to **psql** functions:

- The output format parameter **-r** is added. Allows users to adjust the focus by pressing the **Tab** key or arrow keys when entering commands.
- The **\parallel** metacommand is added to improve execution performance.
- The **\set RETRY** statement is added to support retry upon statement errors.
- Slashes (/) are used as default terminators at the end of PL/SQL statements (such as those used for creating or replacing a function or procedure) for convenience.

### libpq

During the development of certain GaussDB(DWS) functions, such as the **gsql** client connection tool, PostgreSQL libpq is greatly modified. However, the libpq

interface is not verified in application development scenarios. You are not advised to use this set of interfaces for application development, because underlying risks probably exist. You can use the ODBC or JDBC interface instead.

## 2.2 Data Type Differences

For details about supported data types in GaussDB(DWS), see [Data Types](#).

The following PostgreSQL data types are not supported in GaussDB(DWS):

- Line in the geometric type
- XML type

## 2.3 Function Differences

For details about supported functions in GaussDB(DWS), see [Functions and Operators](#).

The following PostgreSQL functions are not supported in GaussDB(DWS):

- Enumerate functions
- XML functions
- Access privilege inquiry functions
  - `has_sequence_privilege(user, sequence, privilege)`
  - `has_sequence_privilege(sequence, privilege)`
- System catalog information functions
  - `pg_get_triggerdef(trigger_oid)`
  - `pg_get_triggerdef(trigger_oid, pretty_bool)`
- Line functions
- `pg_node_tree`

## 2.4 PostgreSQL Features Unsupported by GaussDB(DWS)

- Tablespaces
- Table inheritance
- The following table creation features:
  - Create tables in ROUNDROBIN or MODULO distribution mode.
  - Use **TO NODE/GROUP** to specify the nodes where table data is to be distributed.
  - Use **REFERENCES reftable [ (refcolumn) ] [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ] [ ON DELETE action ] [ ON UPDATE action ]** to create a foreign key constraint for a table.
  - Use **EXCLUDE [ USING index\_method ] ( exclude\_element WITH operator [, ... ] )** to create exclusion constraints for a table.

- The following table modification features:
  - Modify the distribution mode of table data.
  - The list of nodes modified for data distribution can be invoked only during scaling at the GaussDB(DWS) background. Users or applications cannot use the list.
- **CREATE EXTENSION**, **ALTER EXTENSION**, and **DROP EXTENSION** can be invoked only at the GaussDB(DWS) background during upgrade. Users or applications cannot use them.
- Define or change the security tag of an object.
- Aggregate function
- User-defined C functions
- Create, modify, and delete operators.
- Create, modify, and delete operator classes.
- Create, modify, and delete operator families.
- Create, modify, and delete types.
- Create, modify, and delete text search dictionaries.
- Create, modify, and delete text search parsers.
- Create, modify, and delete text search templates.
- Create, modify, and delete collations.
- Create and delete rules.
- Create, modify, and delete triggers.
- Register, modify, and delete languages.
- Create, modify, and delete domains.
- Define, modify, and delete the conversion of character set encoding.
- Define and delete type cast.
- Define, modify, and delete foreign data wrappers.
- Define, modify, and delete user mapping.
- Generate a notification.
- Listen for a notification.
- Stop listening for a notification.
- Load or reload a shared library file.
- Release the session resources of a database.
- Move a cursor backward.

The following features are disabled in GaussDB(DWS) for separation of rights:

- **WITH GRANT OPTION** and **TO PUBLIC** of **GRANT**
- **COPY FROM FILE** and **COPY TO FILE** of **COPY**

# 3 Keyword

The SQL contains reserved and non-reserved words. Standards require that reserved keywords not be used as other identifiers. Non-reserved keywords have special meanings only in a specific environment and can be used as identifiers in other environments.

**Table 3-1** SQL keywords

Keyword	GaussDB(DWS)	SQL:1999	SQL-92
ABORT	Non-reserved	-	-
ABS	-	Non-reserved	-
ABSOLUTE	Non-reserved	Reserved	Reserved
ACCESS	Non-reserved	-	-
ACCOUNT	Non-reserved	-	-
ACTION	Non-reserved	Reserved	Reserved
ADA	-	Non-reserved	Non-reserved
ADD	Non-reserved	Reserved	Reserved
ADMIN	Non-reserved	Reserved	-
AFTER	Non-reserved	Reserved	-
AGGREGATE	Non-reserved	Reserved	-
ALIAS	-	Reserved	-
ALL	Reserved	Reserved	Reserved
ALLOCATE	-	Reserved	Reserved
ALSO	Non-reserved	-	-
ALTER	Non-reserved	Reserved	Reserved

Keyword	GaussDB(DWS)	SQL:1999	SQL-92
ALWAYS	Non-reserved	-	-
ANALYSE	Reserved	-	-
ANALYZE	Reserved	-	-
AND	Reserved	Reserved	Reserved
ANY	Reserved	Reserved	Reserved
APP	Non-reserved	-	-
ARE	-	Reserved	Reserved
ARRAY	Reserved	Reserved	-
AS	Reserved	Reserved	Reserved
ASC	Reserved	Reserved	Reserved
ASENSITIVE	-	Non-reserved	-
ASSERTION	Non-reserved	Reserved	Reserved
ASSIGNMENT	Non-reserved	Non-reserved	-
ASYMMETRIC	Reserved	Non-reserved	-
AT	Non-reserved	Reserved	Reserved
ATOMIC	-	Non-reserved	-
ATTRIBUTE	Non-reserved	-	-
AUTHID	Reserved	-	-
AUTHORIZATION	Reserved (functions and types allowed)	Reserved	Reserved
AUTOEXTEND	Non-reserved	-	-
AUTOMAPPED	Non-reserved	-	-
AVG	-	Non-reserved	Reserved
BACKWARD	Non-reserved	-	-
BARRIER	Non-reserved	-	-
BEFORE	Non-reserved	Reserved	-
BEGIN	Non-reserved	Reserved	Reserved

Keyword	GaussDB(DWS)	SQL:1999	SQL-92
BETWEEN	Non-reserved (excluding functions and types)	Non-reserved	Reserved
BIGINT	Non-reserved (excluding functions and types)	-	-
BINARY	Reserved (functions and types allowed)	Reserved	-
BINARY_DOUBLE	Non-reserved (excluding functions and types)	-	-
BINARY_INTEGER	Non-reserved (excluding functions and types)	-	-
BIT	Non-reserved (excluding functions and types)	Reserved	Reserved
BITVAR	-	Non-reserved	-
BIT_LENGTH	-	Non-reserved	Reserved
BLOB	Non-reserved	Reserved	-
BOOLEAN	Non-reserved (excluding functions and types)	Reserved	-
BOTH	Reserved	Reserved	Reserved
BUCKETS	Reserved	-	-
BREADTH	-	Reserved	-
BY	Non-reserved	Reserved	Reserved
C	-	Non-reserved	Non-reserved
CACHE	Non-reserved	-	-
CALL	Non-reserved	Reserved	-
CALLED	Non-reserved	Non-reserved	-
CARDINALITY	-	Non-reserved	-
CASCADE	Non-reserved	Reserved	Reserved
CASCADED	Non-reserved	Reserved	Reserved
CASE	Reserved	Reserved	Reserved
CAST	Reserved	Reserved	Reserved



Keyword	GaussDB(DWS)	SQL:1999	SQL-92
CATALOG	Non-reserved	Reserved	Reserved
CATALOG_NAME	-	Non-reserved	Non-reserved
CHAIN	Non-reserved	Non-reserved	-
CHAR	Non-reserved (excluding functions and types)	Reserved	Reserved
CHARACTER	Non-reserved (excluding functions and types)	Reserved	Reserved
CHARACTERISTICS	Non-reserved	-	-
CHARACTER_LENGTH	-	Non-reserved	Reserved
CHARACTER_SET_CATALOG	-	Non-reserved	Non-reserved
CHARACTER_SET_NAME	-	Non-reserved	Non-reserved
CHARACTER_SET_SCHEMA	-	Non-reserved	Non-reserved
CHAR_LENGTH	-	Non-reserved	Reserved
CHECK	Reserved	Reserved	Reserved
CHECKED	-	Non-reserved	-
CHECKPOINT	Non-reserved	-	-
CLASS	Non-reserved	Reserved	-
CLEAN	Non-reserved	-	-
CLASS_ORIGIN	-	Non-reserved	Non-reserved
CLOB	Non-reserved	Reserved	-
CLOSE	Non-reserved	Reserved	Reserved
CLUSTER	Non-reserved	-	-
COALESCE	Non-reserved (excluding functions and types)	Non-reserved	Reserved
COBOL	-	Non-reserved	Non-reserved

Keyword	GaussDB(DWS)	SQL:1999	SQL-92
COLLATE	Reserved	Reserved	Reserved
COLLATION	Reserved (functions and types allowed)	Reserved	Reserved
COLLATION_CATALOG	-	Non-reserved	Non-reserved
COLLATION_NAME	-	Non-reserved	Non-reserved
COLLATION_SCHEMA	-	Non-reserved	Non-reserved
COLUMN	Reserved	Reserved	Reserved
COLUMN_NAME	-	Non-reserved	Non-reserved
COMMAND_FUNCTION	-	Non-reserved	Non-reserved
COMMAND_FUNCTION_CODE	-	Non-reserved	-
COMMENT	Non-reserved	-	-
COMMENTS	Non-reserved	-	-
COMMIT	Non-reserved	Reserved	Reserved
COMMITTED	Non-reserved	Non-reserved	Non-reserved
COMPATIBLE_ILLEGAL_CHARS	Non-reserved	-	-
COMPLETE	Non-reserved	-	-
COMPRESS	Non-reserved	-	-
COMPLETION	-	Reserved	-
CONCURRENTLY	Reserved (functions and types allowed)	-	-
CONDITION	-	-	-
CONDITION_NUMBER	-	Non-reserved	Non-reserved
CONFIGURATION	Non-reserved	-	-
CONNECT	-	Reserved	Reserved
CONNECTION	Non-reserved	Reserved	Reserved

Keyword	GaussDB(DWS)	SQL:1999	SQL-92
CONNECTION_NAME	-	Non-reserved	Non-reserved
CONSTRAINT	Reserved	Reserved	Reserved
CONSTRAINTS	Non-reserved	Reserved	Reserved
CONSTRAINT_CATALOG	-	Non-reserved	Non-reserved
CONSTRAINT_NAME	-	Non-reserved	Non-reserved
CONSTRAINT_SCHEMA	-	Non-reserved	Non-reserved
CONSTRUCTOR	-	Reserved	-
CONTAINS	-	Non-reserved	-
CONTENT	Non-reserved	-	-
CONTINUE	Non-reserved	Reserved	Reserved
CONVERSION	Non-reserved	-	-
CONVERT	-	Non-reserved	Reserved
COORDINATOR	Non-reserved	-	-
COPY	Non-reserved	-	-
CORRESPONDING	-	Reserved	Reserved
COST	Non-reserved	-	-
COUNT	-	Non-reserved	Reserved
CREATE	Reserved	Reserved	Reserved
CROSS	Reserved (functions and types allowed)	Reserved	Reserved
CSV	Non-reserved	-	-
CUBE	-	Reserved	-
CURRENT	Non-reserved	Reserved	Reserved
CURRENT_CATALOG	Reserved	-	-
CURRENT_DATE	Reserved	Reserved	Reserved
CURRENT_PATH	-	Reserved	-

Keyword	GaussDB(DWS)	SQL:1999	SQL-92
CURRENT_ROLE	Reserved	Reserved	-
CURRENT_SCHEMA	Reserved (functions and types allowed)	-	-
CURRENT_TIME	Reserved	Reserved	Reserved
CURRENT_TIMESTAMP	Reserved	Reserved	Reserved
CURRENT_USER	Reserved	Reserved	Reserved
CURSOR	Non-reserved	Reserved	Reserved
CURSOR_NAME	-	Non-reserved	Non-reserved
CYCLE	Non-reserved	Reserved	-
DATA	Non-reserved	Reserved	Non-reserved
DATE_FORMAT	Non-reserved	-	-
DATABASE	Non-reserved	-	-
DATAFILE	Non-reserved	-	-
DATE	Non-reserved (excluding functions and types)	Reserved	Reserved
DATETIME_INTERVAL_CODE	-	Non-reserved	Non-reserved
DATETIME_INTERVAL_PRECISION	-	Non-reserved	Non-reserved
DAY	Non-reserved	Reserved	Reserved
DBCOMPATIBILITY	Non-reserved	-	-
DEALLOCATE	Non-reserved	Reserved	Reserved
DEC	Non-reserved (excluding functions and types)	Reserved	Reserved
DECIMAL	Non-reserved (excluding functions and types)	Reserved	Reserved
DECLARE	Non-reserved	Reserved	Reserved
DECODE	Non-reserved (excluding functions and types)	-	-
DEFAULT	Reserved	Reserved	Reserved
DEFAULTS	Non-reserved	-	-

Keyword	GaussDB(DWS)	SQL:1999	SQL-92
DEFERRABLE	Reserved	Reserved	Reserved
DEFERRED	Non-reserved	Reserved	Reserved
DEFINED	-	Non-reserved	-
DEFINER	Non-reserved	Non-reserved	-
DELETE	Non-reserved	Reserved	Reserved
DELIMITER	Non-reserved	-	-
DELIMITERS	Non-reserved	-	-
DELTA	Non-reserved	-	-
DEPTH	-	Reserved	-
DEREF	-	Reserved	-
DESC	Reserved	Reserved	Reserved
DESCRIBE	-	Reserved	Reserved
DESCRIPTOR	-	Reserved	Reserved
DESTROY	-	Reserved	-
DESTRUCTOR	-	Reserved	-
DETERMINISTIC	Non-reserved	Reserved	-
DIAGNOSTICS	-	Reserved	Reserved
DICTIONARY	Non-reserved	Reserved	-
DIRECT	Non-reserved	-	-
DISABLE	Non-reserved	-	-
DISCARD	Non-reserved	-	-
DISCONNECT	-	Reserved	Reserved
DISPATCH	-	Non-reserved	-
DISTINCT	Reserved	Reserved	Reserved
DISTRIBUTE	Non-reserved	-	-
DISTRIBUTION	Non-reserved	-	-
DO	Reserved	-	-
DOCUMENT	Non-reserved	-	-

Keyword	GaussDB(DWS)	SQL:1999	SQL-92
DOMAIN	Non-reserved	Reserved	Reserved
DOUBLE	Non-reserved	Reserved	Reserved
DROP	Non-reserved	Reserved	Reserved
DYNAMIC	-	Reserved	-
DYNAMIC_FUNCTION	-	Non-reserved	Non-reserved
DYNAMIC_FUNCTION_CODE	-	Non-reserved	-
EACH	Non-reserved	Reserved	-
ELASTIC	Non-reserved	-	-
ELSE	Reserved	Reserved	Reserved
ENABLE	Non-reserved	-	-
ENCODING	Non-reserved	-	-
ENCRYPTED	Non-reserved	-	-
END	Reserved	Reserved	Reserved
END-EXEC	-	Reserved	Reserved
ENFORCED	Non-reserved	-	-
ENUM	Non-reserved	-	-
EOL	Non-reserved	-	-
EQUALS	-	Reserved	-
ERRORS	Non-reserved	-	-
ESCAPE	Non-reserved	Reserved	Reserved
ESCAPING	Non-reserved	-	-
EVERY	Non-reserved	Reserved	-
EXCEPT	Reserved	Reserved	Reserved
EXCEPTION	-	Reserved	Reserved
EXCHANGE	Non-reserved	-	-
EXCLUDE	Non-reserved	-	-
EXCLUDING	Non-reserved	-	-
EXCLUSIVE	Non-reserved	-	-
EXEC	-	Reserved	Reserved

Keyword	GaussDB(DWS)	SQL:1999	SQL-92
EXECUTE	Non-reserved	Reserved	Reserved
EXISTING	-	Non-reserved	-
EXISTS	Non-reserved (excluding functions and types)	Non-reserved	Reserved
EXPLAIN	Non-reserved	-	-
EXTENSION	Non-reserved	-	-
EXTERNAL	Non-reserved	Reserved	Reserved
EXTRACT	Non-reserved (excluding functions and types)	Non-reserved	Reserved
FALSE	Reserved	Reserved	Reserved
FAMILY	Non-reserved	-	-
FAST	Non-reserved	-	-
FENCED	Non-reserved	-	-
FETCH	Reserved	Reserved	Reserved
FILEHEADER	Non-reserved	-	-
FILL_MISSING_FIELDS	Non-reserved	-	-
FINAL	-	Non-reserved	-
FIRST	Non-reserved	Reserved	Reserved
FIXED	Non-reserved	Reserved	Reserved
FLOAT	Non-reserved (excluding functions and types)	Reserved	Reserved
FOLLOWING	Non-reserved	-	-
FOR	Reserved	Reserved	Reserved
FORCE	Non-reserved	-	-
FOREIGN	Reserved	Reserved	Reserved
FORMATTER	Non-reserved	-	-
FORTRAN	-	Non-reserved	Non-reserved
FORWARD	Non-reserved	-	-
FOUND	-	Reserved	Reserved

Keyword	GaussDB(DWS)	SQL:1999	SQL-92
FREE	-	Reserved	-
FREEZE	Reserved (functions and types allowed)	-	-
FROM	Reserved	Reserved	Reserved
FULL	Reserved (functions and types allowed)	Reserved	Reserved
FUNCTION	Non-reserved	Reserved	-
FUNCTIONS	Non-reserved	-	-
G	-	Non-reserved	-
GENERAL	-	Reserved	-
GENERATED	-	Non-reserved	-
GET	-	Reserved	Reserved
GLOBAL	Non-reserved	Reserved	Reserved
GO	-	Reserved	Reserved
GOTO	-	Reserved	Reserved
GRANT	Reserved	Reserved	Reserved
GRANTED	Non-reserved	Non-reserved	-
GREATEST	Non-reserved (excluding functions and types)	-	-
GROUP	Reserved	Reserved	Reserved
GROUPING	-	Reserved	-
HANDLER	Non-reserved	-	-
HAVING	Reserved	Reserved	Reserved
HEADER	Non-reserved	-	-
HIERARCHY	-	Non-reserved	-
HOLD	Non-reserved	Non-reserved	-
HOST	-	Reserved	-
HOURL	Non-reserved	Reserved	Reserved



Keyword	GaussDB(DWS)	SQL:1999	SQL-92
IDENTIFIED	Non-reserved	-	-
IDENTITY	Non-reserved	Reserved	Reserved
IF	Non-reserved	-	-
IGNORE	-	Reserved	-
IGNORE_EXTRA_DATA	Non-reserved	-	-
ILIKE	Reserved (functions and types allowed)	-	-
IMMEDIATE	Non-reserved	Reserved	Reserved
IMMUTABLE	Non-reserved	-	-
IMPLEMENTATION	-	Non-reserved	-
IMPLICIT	Non-reserved	-	-
IN	Reserved	Reserved	Reserved
INCLUDING	Non-reserved	-	-
INCREMENT	Non-reserved	-	-
INDEX	Non-reserved	-	-
INDEXES	Non-reserved	-	-
INDICATOR	-	Reserved	Reserved
INFIX	-	Non-reserved	-
INHERIT	Non-reserved	-	-
INHERITS	Non-reserved	-	-
INITIAL	Non-reserved	-	-
INITIALIZE	-	Reserved	-
INITIALLY	Reserved	Reserved	Reserved
INITTRANS	Non-reserved	-	-
INLINE	Non-reserved	-	-
INNER	Reserved (functions and types allowed)	Reserved	Reserved
INOUT	Non-reserved (excluding functions and types)	Reserved	-
INPUT	Non-reserved	Reserved	Reserved

Keyword	GaussDB(DWS)	SQL:1999	SQL-92
INSENSITIVE	Non-reserved	Non-reserved	Reserved
INSERT	Non-reserved	Reserved	Reserved
INSTANCE	-	Non-reserved	-
INSTANTIABLE	-	Non-reserved	-
INSTEAD	Non-reserved	-	-
INT	Non-reserved (excluding functions and types)	Reserved	Reserved
INTEGER	Non-reserved (excluding functions and types)	Reserved	Reserved
INTERNAL	Reserved	-	-
INTERSECT	Reserved	Reserved	Reserved
INTERVAL	Non-reserved (excluding functions and types)	Reserved	Reserved
INTO	Reserved	Reserved	Reserved
INVOKER	Non-reserved	Non-reserved	-
IS	Reserved	Reserved	Reserved
ISNULL	Non-reserved	-	-
ISOLATION	Non-reserved	Reserved	Reserved
ITERATE	-	Reserved	-
JOIN	Reserved (functions and types allowed)	Reserved	Reserved
K	-	Non-reserved	-
KEY	Non-reserved	Reserved	Reserved
KEY_MEMBER	-	Non-reserved	-
KEY_TYPE	-	Non-reserved	-
LABEL	Non-reserved	-	-
LANGUAGE	Non-reserved	Reserved	Reserved

Keyword	GaussDB(DWS)	SQL:1999	SQL-92
LARGE	Non-reserved	Reserved	-
LAST	Non-reserved	Reserved	Reserved
LATERAL	-	Reserved	-
LC_COLLATE	Non-reserved	-	-
LC_CTYPE	Non-reserved	-	-
LEADING	Reserved	Reserved	Reserved
LEAKPROOF	Non-reserved	-	-
LEAST	Non-reserved (excluding functions and types)	-	-
LEFT	Reserved (functions and types allowed)	Reserved	Reserved
LENGTH	-	Non-reserved	Non-reserved
LESS	Reserved	Reserved	-
LEVEL	Non-reserved	Reserved	Reserved
LIKE	Reserved (functions and types allowed)	Reserved	Reserved
LIMIT	Reserved	Reserved	-
LISTEN	Non-reserved	-	-
LOAD	Non-reserved	-	-
LOCAL	Non-reserved	Reserved	Reserved
LOCALTIME	Reserved	Reserved	-
LOCALTIMESTAMP	Reserved	Reserved	-
LOCATION	Non-reserved	-	-
LOCATOR	-	Reserved	-
LOCK	Non-reserved	-	-
LOG	Non-reserved	-	-
LOGGING	Non-reserved	-	-
LOGIN	Non-reserved	-	-
LOOP	Non-reserved	-	-
LOWER	-	Non-reserved	Reserved

Keyword	GaussDB(DWS)	SQL:1999	SQL-92
M	-	Non-reserved	-
MAP	-	Reserved	-
MAPPING	Non-reserved	-	-
MATCH	Non-reserved	Reserved	Reserved
MATCHED	Non-reserved	-	-
MAX	-	Non-reserved	Reserved
MAXEXTENTS	Non-reserved	-	-
MAXSIZE	Non-reserved	-	-
MAXTRANS	Non-reserved	-	-
MAXVALUE	Reserved	-	-
MERGE	Non-reserved	-	-
MESSAGE_LENGTH	-	Non-reserved	Non-reserved
MESSAGE_OCTET_LENGTH	-	Non-reserved	Non-reserved
MESSAGE_TEXT	-	Non-reserved	Non-reserved
METHOD	-	Non-reserved	-
MIN	-	Non-reserved	Reserved
MINEXTENTS	Non-reserved	-	-
MINUS	Reserved	-	-
MINUTE	Non-reserved	Reserved	Reserved
MINVALUE	Non-reserved	-	-
MOD	-	Non-reserved	-
MODE	Non-reserved	-	-
MODIFIES	-	Reserved	-
MODIFY	Reserved	Reserved	-
MODULE	-	Reserved	Reserved

Keyword	GaussDB(DWS)	SQL:1999	SQL-92
MONTH	Non-reserved	Reserved	Reserved
MORE	-	Non-reserved	Non-reserved
MOVE	Non-reserved	-	-
MOVEMENT	Non-reserved	-	-
MUMPS	-	Non-reserved	Non-reserved
NAME	Non-reserved	Non-reserved	Non-reserved
NAMES	Non-reserved	Reserved	Reserved
NATIONAL	Non-reserved (excluding functions and types)	Reserved	Reserved
NATURAL	Reserved (functions and types allowed)	Reserved	Reserved
NCHAR	Non-reserved (excluding functions and types)	Reserved	Reserved
NCLOB	-	Reserved	-
NEW	-	Reserved	-
NEXT	Non-reserved	Reserved	Reserved
NLSSORT	Reserved	-	-
NO	Non-reserved	Reserved	Reserved
NOCOMPRESS	Non-reserved	-	-
NOCYCLE	Non-reserved	-	-
NODE	Non-reserved	-	-
NOLOGGING	Non-reserved	-	-
NOLOGIN	Non-reserved	-	-
NOMAXVALUE	Non-reserved	-	-
NOMINVALUE	Non-reserved	-	-
NONE	Non-reserved (excluding functions and types)	Reserved	-
NOT	Reserved	Reserved	Reserved
NOTHING	Non-reserved	-	-
NOTIFY	Non-reserved	-	-

Keyword	GaussDB(DWS)	SQL:1999	SQL-92
NOTNULL	Reserved (functions and types allowed)	-	-
NOWAIT	Non-reserved	-	-
NULL	Reserved	Reserved	Reserved
NULLABLE	-	Non-reserved	Non-reserved
NULLIF	Non-reserved (excluding functions and types)	Non-reserved	Reserved
NULLS	Non-reserved	-	-
NUMBER	Non-reserved (excluding functions and types)	Non-reserved	Non-reserved
NUMERIC	Non-reserved (excluding functions and types)	Reserved	Reserved
NUMSTR	Non-reserved	-	-
NVARCHAR2	Non-reserved (excluding functions and types)	-	-
NVL	Non-reserved (excluding functions and types)	-	-
OBJECT	Non-reserved	Reserved	-
OCTET_LENGTH	-	Non-reserved	Reserved
OF	Non-reserved	Reserved	Reserved
OFF	Non-reserved	Reserved	-
OFFSET	Reserved	-	-
OIDS	Non-reserved	-	-
OLD	-	Reserved	-
ON	Reserved	Reserved	Reserved
ONLY	Reserved	Reserved	Reserved
OPEN	-	Reserved	Reserved
OPERATION	-	Reserved	-
OPERATOR	Non-reserved	-	-
OPTIMIZATION	Non-reserved	-	-
OPTION	Non-reserved	Reserved	Reserved

Keyword	GaussDB(DWS)	SQL:1999	SQL-92
OPTIONS	Non-reserved	Non-reserved	-
OR	Reserved	Reserved	Reserved
ORDER	Reserved	Reserved	Reserved
ORDINALITY	-	Reserved	-
OUT	Non-reserved (excluding functions and types)	Reserved	-
OUTER	Reserved (functions and types allowed)	Reserved	Reserved
OUTPUT	-	Reserved	Reserved
OVER	Non-reserved	-	-
OVERLAPS	Reserved (functions and types allowed)	Non-reserved	Reserved
OVERLAY	Non-reserved (excluding functions and types)	Non-reserved	-
OVERRIDING	-	Non-reserved	-
OWNED	Non-reserved	-	-
OWNER	Non-reserved	-	-
PACKAGE	Non-reserved	-	-
PAD	-	Reserved	Reserved
PARAMETER	-	Reserved	-
PARAMETERS	-	Reserved	-
PARAMETER_MODE	-	Non-reserved	-
PARAMETER_NAME	-	Non-reserved	-
PARAMETER_ORDINAL_POSITION	-	Non-reserved	-
PARAMETER_SPECIFIC_CATALOG	-	Non-reserved	-
PARAMETER_SPECIFIC_NAME	-	Non-reserved	-
PARAMETER_SPECIFIC_SCHEMA	-	Non-reserved	-

Keyword	GaussDB(DWS)	SQL:1999	SQL-92
PARSER	Non-reserved	-	-
PARTIAL	Non-reserved	Reserved	Reserved
PARTITION	Non-reserved	-	-
PARTITIONS	Non-reserved	-	-
PASCAL	-	Non-reserved	Non-reserved
PASSING	Non-reserved	-	-
PASSWORD	Non-reserved	-	-
PATH	-	Reserved	-
PCTFREE	Non-reserved	-	-
PER	Non-reserved	-	-
PERM	Non-reserved	-	-
PERCENT	Non-reserved	-	-
PERFORMANCE	Reserved	-	-
PLACING	Reserved	-	-
PLAN	Reserved	-	-
PLANS	Non-reserved	-	-
PLI	-	Non-reserved	Non-reserved
POLICY	Non-reserved	-	-
POOL	Non-reserved	-	-
POSITION	Non-reserved (excluding functions and types)	Non-reserved	Reserved
POSTFIX	-	Reserved	-
PRECEDING	Non-reserved	-	-
PRECISION	Non-reserved (excluding functions and types)	Reserved	Reserved
PREFERRED	Non-reserved	-	-
PREFIX	Non-reserved	Reserved	-
PREORDER	-	Reserved	-
PREPARE	Non-reserved	Reserved	Reserved
PREPARED	Non-reserved	-	-



<b>Keyword</b>	<b>GaussDB(DWS)</b>	<b>SQL:1999</b>	<b>SQL-92</b>
PRESERVE	Non-reserved	Reserved	Reserved
PRIMARY	Reserved	Reserved	Reserved
PRIOR	Non-reserved	Reserved	Reserved
PRIVATE	Non-reserved	-	-
PRIVILEGE	Non-reserved	-	-
PRIVILEGES	Non-reserved	Reserved	Reserved
PROCEDURAL	Non-reserved	-	-
PROCEDURE	Reserved	Reserved	Reserved
PROFILE	Non-reserved	-	-
PUBLIC	-	Reserved	Reserved
QUERY	Non-reserved	-	-
QUOTE	Non-reserved	-	-
RANGE	Non-reserved	-	-
RAW	Non-reserved	-	-
READ	Non-reserved	Reserved	Reserved
READS	-	Reserved	-
REAL	Non-reserved (excluding functions and types)	Reserved	Reserved
REASSIGN	Non-reserved	-	-
REBUILD	Non-reserved	-	-
RECHECK	Non-reserved	-	-
RECURSIVE	Non-reserved	Reserved	-
REF	Non-reserved	Reserved	-
REFERENCES	Reserved	Reserved	Reserved
REFERENCING	-	Reserved	-
REINDEX	Non-reserved	-	-
REJECT	Reserved	-	-
RELATIVE	Non-reserved	Reserved	Reserved
RELEASE	Non-reserved	-	-
REOPTIONS	Non-reserved	-	-

Keyword	GaussDB(DWS)	SQL:1999	SQL-92
REMOTE	Non-reserved	-	-
RENAME	Non-reserved	-	-
REPEATABLE	Non-reserved	Non-reserved	Non-reserved
REPLACE	Non-reserved	-	-
REPLICA	Non-reserved	-	-
RESET	Non-reserved	-	-
RESIZE	Non-reserved	-	-
RESOURCE	Non-reserved	-	-
RESTART	Non-reserved	-	-
RESTRICT	Non-reserved	Reserved	Reserved
RESULT	-	Reserved	-
RETURN	Non-reserved	Reserved	-
RETURNED_LENGTH	-	Non-reserved	Non-reserved
RETURNED_OCTET_LENGTH	-	Non-reserved	Non-reserved
RETURNED_SQLSTATE	-	Non-reserved	Non-reserved
RETURNING	Reserved	-	-
RETURNS	Non-reserved	Reserved	-
REUSE	Non-reserved	-	-
REVOKE	Non-reserved	Reserved	Reserved
RIGHT	Reserved (functions and types allowed)	Reserved	Reserved
ROLE	Non-reserved	Reserved	-
ROLLBACK	Non-reserved	Reserved	Reserved
ROLLUP	-	Reserved	-
ROUTINE	-	Reserved	-
ROUTINE_CATALOG	-	Non-reserved	-
ROUTINE_NAME	-	Non-reserved	-

Keyword	GaussDB(DWS)	SQL:1999	SQL-92
ROUTINE_SCHEMA	-	Non-reserved	-
ROW	Non-reserved (excluding functions and types)	Reserved	-
ROWS	Non-reserved	Reserved	Reserved
ROW_COUNT	-	Non-reserved	Non-reserved
RULE	Non-reserved	-	-
SAVEPOINT	Non-reserved	Reserved	-
SCALE	-	Non-reserved	Non-reserved
SCHEMA	Non-reserved	Reserved	Reserved
SCHEMA_NAME	-	Non-reserved	Non-reserved
SCOPE	-	Reserved	-
SCROLL	Non-reserved	Reserved	Reserved
SEARCH	Non-reserved	Reserved	-
SECOND	Non-reserved	Reserved	Reserved
SECTION	-	Reserved	Reserved
SECURITY	Non-reserved	Non-reserved	-
SELECT	Reserved	Reserved	Reserved
SELF	-	Non-reserved	-
SENSITIVE	-	Non-reserved	-
SEQUENCE	Non-reserved	Reserved	-
SEQUENCES	Non-reserved	-	-
SERIALIZABLE	Non-reserved	Non-reserved	Non-reserved
SERVER	Non-reserved	-	-
SERVER_NAME	-	Non-reserved	Non-reserved
SESSION	Non-reserved	Reserved	Reserved

Keyword	GaussDB(DWS)	SQL:1999	SQL-92
SESSION_USER	Reserved	Reserved	Reserved
SET	Non-reserved	Reserved	Reserved
SETOF	Non-reserved (excluding functions and types)	-	-
SETS	-	Reserved	-
SHARE	Non-reserved	-	-
SHIPPABLE	Non-reserved	-	-
SHOW	Non-reserved	-	-
SIMILAR	Reserved (functions and types allowed)	Non-reserved	-
SIMPLE	Non-reserved	Non-reserved	-
SIZE	Non-reserved	Reserved	Reserved
SMALLDATETIME	Non-reserved (excluding functions and types)	-	-
SMALLDATETIME_FORMAT	Non-reserved	-	-
SMALLINT	Non-reserved (excluding functions and types)	Reserved	Reserved
SNAPSHOT	Non-reserved	-	-
SOME	Reserved	Reserved	Reserved
SOURCE	Non-reserved	Non-reserved	-
SPACE	-	Reserved	Reserved
SPECIFIC	-	Reserved	-
SPECIFICTYPE	-	Reserved	-
SPECIFIC_NAME	-	Non-reserved	-
SPILL	Non-reserved	-	-
SPLIT	Non-reserved	-	-
SQL	-	Reserved	Reserved
SQLCODE	-	-	Reserved
SQLERROR	-	-	Reserved

Keyword	GaussDB(DWS)	SQL:1999	SQL-92
SQLEXCEPTION	-	Reserved	-
SQLSTATE	-	Reserved	Reserved
SQLWARNING	-	Reserved	-
STABLE	Non-reserved	-	-
STANDALONE	Non-reserved	-	-
START	Non-reserved	Reserved	-
STATE	-	Reserved	-
STATEMENT	Non-reserved	Reserved	-
STATEMENT_ID	Non-reserved	-	-
STATIC	-	Reserved	-
STATISTICS	Non-reserved	-	-
STDIN	Non-reserved	-	-
STDOUT	Non-reserved	-	-
STORAGE	Non-reserved	-	-
STORE	Non-reserved	-	-
STRICT	Non-reserved	-	-
STRIP	Non-reserved	-	-
STRUCTURE	-	Reserved	-
STYLE	-	Non-reserved	-
SUBCLASS_ORIGIN	-	Non-reserved	Non-reserved
SUBLIST	-	Non-reserved	-
SUBSTRING	Non-reserved (excluding functions and types)	Non-reserved	Reserved
SUM	-	Non-reserved	Reserved
SUPERUSER	Non-reserved	-	-
SYMMETRIC	Reserved	Non-reserved	-
SYNONYM	Non-reserved	-	-

Keyword	GaussDB(DWS)	SQL:1999	SQL-92
SYS_REFCURSOR	Non-reserved	-	-
SYSDATE	Reserved	-	-
SYSID	Non-reserved	-	-
SYSTEM	Non-reserved	Non-reserved	-
SYSTEM_USER	-	Reserved	Reserved
TABLE	Reserved	Reserved	Reserved
TABLES	Non-reserved	-	-
TABLE_NAME	-	Non-reserved	Non-reserved
TEMP	Non-reserved	-	-
TEMPLATE	Non-reserved	-	-
TEMPORARY	Non-reserved	Reserved	Reserved
TERMINATE	-	Reserved	-
TEXT	Non-reserved	-	-
THAN	Non-reserved	Reserved	-
THEN	Reserved	Reserved	Reserved
TIME	Non-reserved (excluding functions and types)	Reserved	Reserved
TIME_FORMAT	Non-reserved	-	-
TIMESTAMP	Non-reserved (excluding functions and types)	Reserved	Reserved
TIMESTAMP_FORMAT	Non-reserved	-	-
TIMEZONE_HOUR	-	Reserved	Reserved
TIMEZONE_MINUTE	-	Reserved	Reserved
TINYINT	Non-reserved (excluding functions and types)	-	-
TO	Reserved	Reserved	Reserved
TRAILING	Reserved	Reserved	Reserved
TRANSACTION	Non-reserved	Reserved	Reserved
TRANSACTIONS_COMMITTED	-	Non-reserved	-

Keyword	GaussDB(DWS)	SQL:1999	SQL-92
TRANSACTIONS_ROLLED_BACK	-	Non-reserved	-
TRANSACTION_ACTIVE	-	Non-reserved	-
TRANSFORM	-	Non-reserved	-
TRANSFORMS	-	Non-reserved	-
TRANSLATE	-	Non-reserved	Reserved
TRANSLATION	-	Reserved	Reserved
TREAT	Non-reserved (excluding functions and types)	Reserved	-
TRIGGER	Non-reserved	Reserved	-
TRIGGER_CATALOG	-	Non-reserved	-
TRIGGER_NAME	-	Non-reserved	-
TRIGGER_SCHEMA	-	Non-reserved	-
TRIM	Non-reserved (excluding functions and types)	Non-reserved	Reserved
TRUE	Reserved	Reserved	Reserved
TRUNCATE	Non-reserved	-	-
TRUSTED	Non-reserved	-	-
TYPE	Non-reserved	Non-reserved	Non-reserved
TYPES	Non-reserved	-	-
UESCAPE	-	-	-
UNBOUNDED	Non-reserved	-	-
UNCOMMITTED	Non-reserved	Non-reserved	Non-reserved
UNDER	-	Reserved	-
UNENCRYPTED	Non-reserved	-	-
UNION	Reserved	Reserved	Reserved

Keyword	GaussDB(DWS)	SQL:1999	SQL-92
UNIQUE	Reserved	Reserved	Reserved
UNKNOWN	Non-reserved	Reserved	Reserved
UNLIMITED	Non-reserved	-	-
UNLISTEN	Non-reserved	-	-
UNLOCK	Non-reserved	-	-
UNLOGGED	Non-reserved	-	-
UNNAMED	-	Non-reserved	Non-reserved
UNNEST	-	Reserved	-
UNTIL	Non-reserved	-	-
UNUSABLE	Non-reserved	-	-
UPDATE	Non-reserved	Reserved	Reserved
UPPER	-	Non-reserved	Reserved
USAGE	-	Reserved	Reserved
USER	Reserved	Reserved	Reserved
USER_DEFINED_TYPE_CATALOG	-	Non-reserved	-
USER_DEFINED_TYPE_NAME	-	Non-reserved	-
USER_DEFINED_TYPE_SCHEMA	-	Non-reserved	-
USING	Reserved	Reserved	Reserved
VACUUM	Non-reserved	-	-
VALID	Non-reserved	-	-
VALIDATE	Non-reserved	-	-
VALIDATION	Non-reserved	-	-
VALIDATOR	Non-reserved	-	-
VALUE	Non-reserved	Reserved	Reserved
VALUES	Non-reserved (excluding functions and types)	Reserved	Reserved
VARCHAR	Non-reserved (excluding functions and types)	Reserved	Reserved



Keyword	GaussDB(DWS)	SQL:1999	SQL-92
VARCHAR2	Non-reserved (excluding functions and types)	-	-
VARIABLE	-	Reserved	-
VARIADIC	Reserved	-	-
VARYING	Non-reserved	Reserved	Reserved
VCGROUP	Non-reserved	-	-
VERBOSE	Reserved (functions and types allowed)	-	-
VERIFY	Reserved	-	-
VERSION	Non-reserved	-	-
VIEW	Non-reserved	Reserved	Reserved
VOLATILE	Non-reserved	-	-
WHEN	Reserved	Reserved	Reserved
WHENEVER	-	Reserved	Reserved
WHERE	Reserved	Reserved	Reserved
WHITESPACE	Non-reserved	-	-
WINDOW	Reserved	-	-
WITH	Reserved	Reserved	Reserved
WITHIN	Non-reserved	-	-
WITHOUT	Non-reserved	Reserved	-
WORK	Non-reserved	Reserved	Reserved
WORKLOAD	Non-reserved	-	-
WRAPPER	Non-reserved	-	-
WRITE	Non-reserved	Reserved	Reserved
XML	Non-reserved	-	-
XMLATTRIBUTES	Non-reserved (excluding functions and types)	-	-
XMLCONCAT	Non-reserved (excluding functions and types)	-	-
XMLELEMENT	Non-reserved (excluding functions and types)	-	-

<b>Keyword</b>	<b>GaussDB(DWS)</b>	<b>SQL:1999</b>	<b>SQL-92</b>
XML EXISTS	Non-reserved (excluding functions and types)	-	-
XML FOREST	Non-reserved (excluding functions and types)	-	-
XML PARSE	Non-reserved (excluding functions and types)	-	-
XML PI	Non-reserved (excluding functions and types)	-	-
XML ROOT	Non-reserved (excluding functions and types)	-	-
XML SERIALIZE	Non-reserved (excluding functions and types)	-	-
YEAR	Non-reserved	Reserved	Reserved
YES	Non-reserved	-	-
ZONE	Non-reserved	Reserved	Reserved

# 4 Data Types

## 4.1 Numeral

[Table 4-1](#) lists all available types. For digit operators and related built-in functions, see [Mathematical Functions and Operators](#).

**Table 4-1** Integer types

Name	Description	Storage Space	Value Range
TINYINT	Tiny integer, also called INT1	1 byte	0 ~ 255
SMALLINT	Small integer, also called INT2	2 bytes	-32,768 ~ +32,767
INTEGER	Typical choice for integer, also called INT4	4 bytes	-2,147,483,648 ~ +2,147,483,647
BINARY_INTEGER	INTEGER alias, compatible with Oracle	4 bytes	-2,147,483,648 ~ +2,147,483,647
BIGINT	Big integer, also called INT8	8 bytes	-9,223,372,036,854,775,808 ~ 9,223,372,036,854,775,807

For example:

```
-- Create a table containing TINYINT data:
CREATE TABLE int_type_t1
(
  IT_COL1 TINYINT
);

-- Create a table containing TINYINT data:
```

```

INSERT INTO int_type_t1 VALUES(10);

-- View data:
SELECT * FROM int_type_t1;
it_col1
-----
10
(1 row)

-- Delete the tables:
DROP TABLE int_type_t1;
-- Create a table containing TINYINT, INTEGER, and BIGINT data:
CREATE TABLE int_type_t2
(
  a TINYINT,
  b TINYINT,
  c INTEGER,
  d BIGINT
);

-- Insert data:
INSERT INTO int_type_t2 VALUES(100, 10, 1000, 10000);

-- View data:
SELECT * FROM int_type_t2;
 a | b | c | d
-----+-----
100 | 10 | 1000 | 10000
(1 row)

-- Delete the tables:
DROP TABLE int_type_t2;

```

 **NOTE**

- The TINYINT, SMALLINT, INTEGER, and BIGINT types store whole numbers, that is, numbers without fractional components, of various ranges. Saving a number with a decimal in any of the data types will result in errors.
- The INTEGER type is the common choice, as it offers the proper balance between the range, storage size, and performance. Generally, use the SMALLINT type only if you are sure that the value range is within the SMALLINT value range. The storage speed of INTEGER is much faster. BIGINT is used only when the range of INTEGER is not large enough.

**Table 4-2** Any-precision types

Name	Description	Storage Space	Value Range
NUMERIC[(p[,s])], DECIMAL[(p[,s])]	The value range of p (precision) is [1,1000], and the value range of s (standard) is [0,p]. <b>NOTE</b> p indicates the total digits, and s indicates the decimal digit.	The precision is specified by users. Every four decimal digits occupy two bytes, and an extra eight-byte overhead is added to the entire data.	Up to 131,072 digits before the decimal point; and up to 16,383 digits after the decimal point when no precision is specified

Name	Description	Storage Space	Value Range
NUMBER[(p[,s])]	Alias for type NUMERIC, compatible with Oracle	The precision is specified by users. Every four decimal digits occupy two bytes, and an extra eight-byte overhead is added to the entire data.	Up to 131,072 digits before the decimal point; and up to 16,383 digits after the decimal point when no precision is specified

### Examples

```
-- Create a table:
CREATE TABLE decimal_type_t1
(
  DT_COL1 DECIMAL(10,4)
);

-- Insert data:
INSERT INTO decimal_type_t1 VALUES(123456.122331);

-- Query data in the table:
SELECT * FROM decimal_type_t1;
  dt_col1
-----
123456.1223
(1 row)

-- Delete the tables:
DROP TABLE decimal_type_t1;
-- Create a table:
CREATE TABLE numeric_type_t1
(
  NT_COL1 NUMERIC(10,4)
);

-- Insert data:
INSERT INTO numeric_type_t1 VALUES(123456.12354);

-- Query data in the table:
SELECT * FROM numeric_type_t1;
  nt_col1
-----
123456.1235
(1 row)

-- Delete the tables:
DROP TABLE numeric_type_t1;
```

#### NOTE

- Compared to the integer types, the arbitrary precision numbers require larger storage space and have lower storage efficiency, operation efficiency, and poorer compression ratio results. The INTEGER type is the common choice when number types are defined. Arbitrary precision numbers are used only when numbers exceed the maximum range indicated by the integers.
- When NUMERIC/DECIMAL is used for defining a column, you are advised to specify the precision (p) and scale (s) for the column.

**Table 4-3** Sequence integer

Name	Description	Storage Space	Value Range
SMALLSERIAL	Two-byte auto-incrementing integer	2 bytes	1 ~ 32,767
SERIAL	Four-byte auto-incrementing integer	4 bytes	1 ~ 2,147,483,647
BIGSERIAL	Eight-byte auto-incrementing integer	8 bytes	1 ~ 9,223,372,036,854,775,807

### Examples

```
-- Create a table:
CREATE TABLE smallserial_type_tab(a SMALLSERIAL);

-- Insert data:
INSERT INTO smallserial_type_tab VALUES(default);

-- Insert data again:
INSERT INTO smallserial_type_tab VALUES(default);

-- View data:
SELECT * FROM smallserial_type_tab;
a
---
1
2
(2 rows)

-- Create a table:
CREATE TABLE serial_type_tab(b SERIAL);

-- Insert data:
INSERT INTO serial_type_tab VALUES(default);

-- Insert data again:
INSERT INTO serial_type_tab VALUES(default);

-- View data:
SELECT * FROM serial_type_tab;
b
---
1
2
(2 rows)

-- Create a table:
CREATE TABLE bigserial_type_tab(c BIGSERIAL);

-- Insert data:
INSERT INTO bigserial_type_tab VALUES(default);

-- Insert data:
INSERT INTO bigserial_type_tab VALUES(default);

-- View data:
SELECT * FROM bigserial_type_tab;
```

```
c
---
1
2
(2 rows)

-- Delete the tables:
DROP TABLE smallserial_type_tab;

DROP TABLE serial_type_tab;

DROP TABLE bigserial_type_tab;
```

 **NOTE**

SMALLSERIAL, SERIAL, and BIGSERIAL are not real types. They are concepts used for setting a unique identifier for a table. Therefore, an integer column is created and its default value plans to be read from a sequencer. A NOT NULL constraint is used to ensure NULL is not inserted. In most cases you would also want to attach a UNIQUE or PRIMARY KEY constraint to prevent duplicate values from being inserted unexpectedly, but this is not automatic. The sequencer is set so that it belongs to the column. In this case, when the column or the table is deleted, the sequencer is also deleted. Currently, the SERIAL column can be specified only when you create a table. You cannot add the SERIAL column in an existing table. In addition, SERIAL columns cannot be created in temporary tables. Because SERIAL is not a data type, columns cannot be converted to this type.

**Table 4-4** Floating point types

Name	Description	Storage Space	Value Range
REAL, FLOAT4	Single precision floating points, inexact	4 bytes	Six bytes of decimal digits
DOUBLE PRECISION, FLOAT8	Double precision floating points, inexact	8 bytes	1E-307~1E+308, 15 bytes of decimal digits
FLOAT[(p)]	Floating points, inexact. The value range of precision (p) is [1,53]. <b>NOTE</b> p is the precision, indicating the total decimal digits.	4 or 8 bytes	REAL or DOUBLE PRECISION is selected as an internal identifier based on different precision (p). If no precision is specified, DOUBLE PRECISION is used as the internal identifier.
BINARY_DOUBLE	DOUBLE PRECISION alias, compatible with Oracle	8 bytes	1E-307~1E+308, 15 bytes of decimal digits

Name	Description	Storage Space	Value Range
DEC[(p[,s]) ]	The value range of p (precision) is [1,1000], and the value range of s (standard) is [0,p]. <b>NOTE</b> p indicates the total digits, and s indicates the decimal digit.	The precision is specified by users. Every four decimal digits occupy two bytes, and an extra eight-byte overhead is added to the entire data.	Up to 131,072 digits before the decimal point; and up to 16,383 digits after the decimal point when no precision is specified
INTEGER[( p[,s])]	The value range of p (precision) is [1,1000], and the value range of s (standard) is [0,p].	The precision is specified by users. Every four decimal digits occupy two bytes, and an extra eight-byte overhead is added to the entire data.	Up to 131,072 digits before the decimal point; and up to 16,383 digits after the decimal point when no precision is specified

### Examples

```
-- Create a table:
CREATE TABLE float_type_t2
(
  FT_COL1 INTEGER,
  FT_COL2 FLOAT4,
  FT_COL3 FLOAT8,
  FT_COL4 FLOAT(3),
  FT_COL5 BINARY_DOUBLE,
  FT_COL6 DECIMAL(10,4),
  FT_COL7 INTEGER(6,3)
) DISTRIBUTE BY HASH ( ft_col1);

-- Insert data:
INSERT INTO float_type_t2 VALUES(10,10.365456,123456.1234,10.3214, 321.321, 123.123654, 123.123654);

-- View data:
SELECT * FROM float_type_t2 ;
ft_col1 | ft_col2 | ft_col3 | ft_col4 | ft_col5 | ft_col6 | ft_col7
-----+-----+-----+-----+-----+-----+-----
      10 | 10.3655 | 123456.1234 | 10.3214 | 321.321 | 123.1237 | 123.124
(1 row)

-- Delete the tables:
DROP TABLE float_type_t2;
```

## 4.2 Monetary Types

The **money** type stores a currency amount with fixed fractional precision.

The range shown in [Table 4-5](#) assumes there are two fractional digits. Input is accepted in a variety of formats, including integer and floating-point literals, as well as typical currency formatting. Output is generally in the latter form but depends on the locale.



**Table 4-5** Monetary types

Name	Storage Size	Description	Applicable Scope
money	8 bytes	Currency amount	-92233720368547758.08 to +92233720368547758.07

Values of the **numeric**, **int**, and **bigint** data types can be cast to **money**. Conversion from the **real** and **double precision** data types can be done by casting to **numeric** first, for example:

```
SELECT '12.34'::float8::numeric::money;
```

However, this is not recommended. Floating point numbers should not be used to handle money due to the potential for rounding errors.

A **money** value can be cast to **numeric** without loss of precision. Conversion to other types could potentially lose precision, and must also be done in two stages:

```
SELECT '52093.89'::money::numeric::float8;
```

When a **money** value is divided by another **money** value, the result is **double precision** (that is, a pure number, not money); the currency units cancel each other out in the division.

## 4.3 Boolean Type

**Table 4-6** Boolean type

Name	Description	Storage Space	Value
BOOLEAN	Boolean type	1 byte	<ul style="list-style-type: none"> <li>• <b>true</b></li> <li>• <b>false</b></li> <li>• <b>null</b> (unknown)</li> </ul>

Valid literal values for the "true" state are:

**TRUE**, 't', 'true', 'y', 'yes', '1'

Valid literal values for the "false" state include:

**FALSE**, 'f', 'false', 'n', 'no', '0'

**TRUE** and **FALSE** are standard expressions, compatible with SQL statements.

### Examples

Boolean values are displayed using the letters t and f.

```
-- Create a table:
CREATE TABLE bool_type_t1
```

```
(
  BT_COL1 BOOLEAN,
  BT_COL2 TEXT
) DISTRIBUTE BY HASH(BT_COL2);

-- Insert data:
INSERT INTO bool_type_t1 VALUES (TRUE, 'sic est');

INSERT INTO bool_type_t1 VALUES (FALSE, 'non est');

-- View data:
SELECT * FROM bool_type_t1;
bt_col1 | bt_col2
-----+-----
t       | sic est
f       | non est
(2 rows)

SELECT * FROM bool_type_t1 WHERE bt_col1 = 't';
bt_col1 | bt_col2
-----+-----
t       | sic est
(1 row)

-- Delete the tables:
DROP TABLE bool_type_t1;
```

## 4.4 Character Types

**Table 4-7** lists the character types that can be used in GaussDB(DWS). For string operators and related built-in functions, see [Character Processing Functions and Operators](#).

**Table 4-7** Character types

Name	Description	Storage Space
CHAR(n) CHARACTER(n) NCHAR(n)	Fixed-length string, blank padded. <b>n</b> indicates the string length. If it is not specified, the default precision <b>1</b> is used.	The maximum size is 10 MB.
VARCHAR(n) CHARACTER VARYING(n)	Variable-length string. <b>n</b> indicates the string length.	The maximum size is 10 MB.
VARCHAR2(n)	Variable-length string. It is an alias for VARCHAR(n) type, compatible with Oracle. <b>n</b> indicates the string length.	The maximum size is 10 MB.
NVARCHAR2(n)	Variable-length string. <b>n</b> indicates the string length.	The maximum size is 10 MB.
CLOB	A big text object. It is an alias for TEXT type, compatible with Oracle.	The maximum size is 10,737,362,100 bytes (1 GB - 8203 bytes).

Name	Description	Storage Space
TEXT	Variable-length string.	The maximum size is 10,7373,3621 bytes (1 GB - 8203 bytes).

 **NOTE**

In addition to the size limitation on each column, the total size of each tuple is 10,7373,3621 bytes (1 GB - 8203 bytes).

GaussDB(DWS) has two other fixed-length character types, as listed in [Table 4-8](#). The **name** type exists only for the storage of identifiers in the internal system catalogs and is not intended for use by general users. Its length is currently defined as 64 bytes (63 usable characters plus terminator). The type "**char**" only uses one byte of storage. It is internally used in the system catalogs as a simplistic enumeration type.

**Table 4-8** Special character types

Name	Description	Storage Space
name	Internal type for object names	64 bytes
"char"	Single-byte internal type	1 byte

## Examples

```
-- Create a table:
CREATE TABLE char_type_t1
(
  CT_COL1 CHARACTER(4)
) DISTRIBUTE BY HASH (CT_COL1);

-- Insert data:
INSERT INTO char_type_t1 VALUES ('ok');

-- Query data in the table:
SELECT ct_col1, char_length(ct_col1) FROM char_type_t1;
ct_col1 | char_length
-----+-----
ok      |          4
(1 row)

-- Delete the tables:
DROP TABLE char_type_t1;
-- Create a table:
CREATE TABLE char_type_t2
(
  CT_COL1 VARCHAR(5)
) DISTRIBUTE BY HASH (CT_COL1);

-- Insert data:
INSERT INTO char_type_t2 VALUES ('ok');
```

```

INSERT INTO char_type_t2 VALUES ('good');

-- Specify the type length. An error is reported if an inserted string exceeds this length.
INSERT INTO char_type_t2 VALUES ('too long');
ERROR: value too long for type character varying(4)
CONTEXT: referenced column: ct_col1

-- Specify the type length. A string exceeding this length is truncated.
INSERT INTO char_type_t2 VALUES ('too long'::varchar(5));

-- Query data:
SELECT ct_col1, char_length(ct_col1) FROM char_type_t2;
ct_col1 | char_length
-----+-----
ok      |          2
good    |          5
too l   |          5
(3 rows)

-- Delete data:
DROP TABLE char_type_t2;

```

## 4.5 Binary Data Types

**Table 4-9** lists the binary data types that can be used in GaussDB(DWS).

**Table 4-9** Binary data types

Name	Description	Storage Space
BLOB	Binary large object. Currently, BLOB only supports the following external access interfaces: <ul style="list-style-type: none"> <li>• DBMS_LOB.GETLENGTH</li> <li>• DBMS_LOB.READ</li> <li>• DBMS_LOB.WRITE</li> <li>• DBMS_LOB.WRITEAPPEND</li> <li>• DBMS_LOB.COPY</li> <li>• DBMS_LOB.ERASE</li> </ul> For details about the interfaces, see DBMS_LOB. <b>NOTE</b> Column storage cannot be used for the BLOB type.	The maximum size is 8203 bytes less than 1 GB.
RAW	Variable-length hexadecimal string <b>NOTE</b> Column storage cannot be used for the raw type.	4 bytes plus the actual hexadecimal string. The maximum size is 8203 bytes less than 1 GB.

Name	Description	Storage Space
BYTEA	Variable-length binary string	4 bytes plus the actual binary string. The maximum size is 8203 bytes less than 1 GB.

 **NOTE**

In addition to the size limitation on each column, the total size of each tuple is 8203 bytes less than 1 GB.

**Examples**

```
-- Create a table:
CREATE TABLE blob_type_t1
(
  BT_COL1 INTEGER,
  BT_COL2 BLOB,
  BT_COL3 RAW,
  BT_COL4 BYTEA
) DISTRIBUTE BY REPLICATION;

-- Insert data:
INSERT INTO blob_type_t1 VALUES(10,empty_blob(),
HEXTORAW('DEADBEEF'),E'\xDEADBEEF');

-- Query data in the table:
SELECT * FROM blob_type_t1;
bt_col1 | bt_col2 | bt_col3 | bt_col4
-----+-----+-----+-----
    10 |      | DEADBEEF | \xdeadbeef
(1 row)

-- Delete the tables:
DROP TABLE blob_type_t1;
```

## 4.6 Date/Time Types

**Table 4-10** lists the date/time types that can be used in GaussDB(DWS). For the operators and built-in functions of the types, see [Date and Time Processing Functions and Operators](#).

 **NOTE**

If the time format of another database is different from that of GaussDB(DWS), modify the value of the **DateStyle** parameter to keep them consistent.

**Table 4-10** Date/time types

Name	Description	Storage Space
DATE	Specifies the date and time.	4 bytes (The actual storage space is 8 bytes.)

Name	Description	Storage Space
TIME [(p)] [WITHOUT TIME ZONE]	Specifies time within one day. <b>p</b> indicates the precision after the decimal point. The value ranges from 0 to 6.	8 bytes
TIME [(p)] [WITH TIME ZONE]	Specifies time within one day (with time zone). <b>p</b> indicates the precision after the decimal point. The value ranges from 0 to 6.	12 bytes
TIMESTAMP[(p)] [WITHOUT TIME ZONE]	Specifies the date and time. <b>p</b> indicates the precision after the decimal point. The value ranges from 0 to 6.	8 bytes
TIMESTAMP[(p)] [WITH TIME ZONE]	Specifies the date and time (with time zone). <b>TIMESTAMP</b> is also called <b>TIMESTAMPTZ</b> . <b>p</b> indicates the precision after the decimal point. The value ranges from 0 to 6.	8 bytes
SMALLDATETIME	Specifies the date and time (without time zone). The precision level is minute. 31s to 59s are rounded into 1 minute.	8 bytes
INTERVAL DAY (l) TO SECOND (p)	Specifies the time interval (X days X hours X minutes X seconds). <ul style="list-style-type: none"> <li>• <b>l</b>: indicates the precision of days. The value ranges from 0 to 6. To adapt to Oracle syntax, the precision functions are not supported.</li> <li>• <b>p</b>: indicates the precision of seconds. The value ranges from 0 to 6. The digit 0 at the end of a decimal number is not displayed.</li> </ul>	16 bytes

Name	Description	Storage Space
INTERVAL [FIELDS] [ (p) ]	<p>Specifies the time interval.</p> <ul style="list-style-type: none"> <li>fields: <b>YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, DAY TO HOUR, DAY TO MINUTE, DAY TO SECOND, HOUR TO MINUTE, HOUR TO SECOND, and MINUTE TO SECOND.</b></li> <li><b>p</b>: indicates the precision of seconds. The value ranges from 0 to 6. <b>p</b> takes effect only when fields are <b>SECOND, DAY TO SECOND, HOUR TO SECOND, or MINUTE TO SECOND.</b> The digit 0 at the end of a decimal number is not displayed.</li> </ul>	12 bytes
reltime	<p>Relative time interval. The format is: <i>X years X months X days XX:XX:XX</i></p> <ul style="list-style-type: none"> <li>The Julian calendar is used. It specifies that a year has 365.25 days and a month has 30 days. The relative time interval needs to be calculated based on the input value. The output format is POSTGRES.</li> </ul>	4 bytes

For example:

```

-- Create a table:
CREATE TABLE date_type_tab(coll date);

-- Insert data:
INSERT INTO date_type_tab VALUES (date '12-10-2010');

-- View data:
SELECT * FROM date_type_tab;
      coll
-----
2010-12-10 00:00:00
(1 row)

-- Delete the tables:
DROP TABLE date_type_tab;

-- Create a table:
CREATE TABLE time_type_tab (da time without time zone ,dai time with time zone,dfgh timestamp without
time zone,dfga timestamp with time zone, vbg smalldatetime);

-- Insert data:
INSERT INTO time_type_tab VALUES ('21:21:21','21:21:21 pst','2010-12-12','2013-12-11 pst','2003-04-12
04:05:06');

-- View data:
SELECT * FROM time_type_tab;

```

```

    da | dai | dfg | dfga | vbg
-----+-----+-----+-----+-----
21:21:21 | 21:21:21-08 | 2010-12-12 00:00:00 | 2013-12-11 16:00:00+08 | 2003-04-12 04:05:00
(1 row)

-- Delete the tables:
DROP TABLE time_type_tab;

-- Create a table:
CREATE TABLE day_type_tab (a int,b INTERVAL DAY(3) TO SECOND (4));

-- Insert data:
INSERT INTO day_type_tab VALUES (1, INTERVAL '3' DAY);

-- View data:
SELECT * FROM day_type_tab;
a | b
-----+-----
1 | 3 days
(1 row)

-- Delete the tables:
DROP TABLE day_type_tab;

-- Create a table:
CREATE TABLE year_type_tab(a int, b interval year (6));

-- Insert data:
INSERT INTO year_type_tab VALUES(1,interval '2' year);

-- View data:
SELECT * FROM year_type_tab;
a | b
-----+-----
1 | 2 years
(1 row)

-- Delete the tables:
DROP TABLE year_type_tab;
```

## Date Input

Date and time input is accepted in almost any reasonable formats, including ISO 8601, SQL-compatible, and traditional POSTGRES. The system allows you to customize the sequence of day, month, and year in the date input. Set the **DateStyle** parameter to **MDY** to select month-day-year interpretation, **DMY** to select day-month-year interpretation, or **YMD** to select year-month-day interpretation.

Remember that any date or time literal input needs to be enclosed with single quotes, and the syntax is as follows:

```
type [ ( p ) ] 'value'
```

The **p** that can be selected in the precision statement is an integer, indicating the number of fractional digits in the **seconds** column. [Table 4-11](#) shows some possible inputs for the **date** type.



**Table 4-11** Date input

Example	Description
1999-01-08	ISO 8601 (recommended format). January 8, 1999 in any mode
January 8, 1999	Unambiguous in any date input mode
1/8/1999	January 8 in <b>MDY</b> mode. August 1 in <b>DMY</b> mode
1/18/1999	January 18 in <b>MDY</b> mode, rejected in other modes
01/02/03	<ul style="list-style-type: none"> <li>January 2, 2003 in <b>MDY</b> mode</li> <li>February 1, 2003 in <b>DMY</b> mode</li> <li>February 3, 2001 in <b>YMD</b> mode</li> </ul>
1999-Jan-08	January 8 in any mode
Jan-08-1999	January 8 in any mode
08-Jan-1999	January 8 in any mode
99-Jan-08	January 8 in <b>YMD</b> mode, else error
08-Jan-99	January 8, except error in <b>YMD</b> mode
Jan-08-99	January 8, except error in <b>YMD</b> mode
19990108	ISO 8601. January 8, 1999 in any mode
990108	ISO 8601. January 8, 1999 in any mode
1999.008	Year and day of year
J2451187	Julian date
January 8, 99 BC	Year 99 BC

For example:

```
-- Create a table:
CREATE TABLE date_type_tab(coll date);

-- Insert data:
INSERT INTO date_type_tab VALUES (date '12-10-2010');

-- View data:
SELECT * FROM date_type_tab;
      coll
-----
2010-12-10 00:00:00
(1 row)

-- View the date format:
SHOW datestyle;
DateStyle
-----
ISO, MDY
(1 row)
```

```
-- Set the date format:
SET datestyle='YMD';
SET

-- Insert data:
INSERT INTO date_type_tab VALUES(date '2010-12-11');

-- View data:
SELECT * FROM date_type_tab;
      coll
-----
2010-12-10 00:00:00
2010-12-11 00:00:00
(2 rows)

-- Delete the tables:
DROP TABLE date_type_tab;
```

## Times

The time-of-day types are **TIME [(p)] [WITHOUT TIME ZONE]** and **TIME [(p)] [WITH TIME ZONE]**. **TIME** alone is equivalent to **TIME WITHOUT TIME ZONE**.

If a time zone is specified in the input for **TIME WITHOUT TIME ZONE**, it is silently ignored.

For details about the time input types, see [Table 4-12](#). For details about time zone input types, see [Table 4-13](#).

**Table 4-12** Time input

Example	Description
05:06.8	ISO 8601
4:05:06	ISO 8601
4:05	ISO 8601
40506	ISO 8601
4:05 AM	Same as 04:05. AM does not affect value
4:05 PM	Same as 16:05. Input hour must be <= 12
04:05:06.789-8	ISO 8601
04:05:06-08:00	ISO 8601
04:05-08:00	ISO 8601
040506-08	ISO 8601
04:05:06 PST	Time zone specified by abbreviation
2003-04-12 04:05:06 America/ New_York	Time zone specified by full name

**Table 4-13** Time zone input

Example	Description
PST	Abbreviation (for Pacific Standard Time)
America/New_York	Full time zone name
-8:00	ISO-8601 offset for PST
-800	ISO-8601 offset for PST
-8	ISO-8601 offset for PST

For example:

```
SELECT time '04:05:06';
time
-----
04:05:06
(1 row)

SELECT time '04:05:06 PST';
time
-----
04:05:06
(1 row)

SELECT time with time zone '04:05:06 PST';
timetz
-----
04:05:06-08
(1 row)
```

## Special Values

The special values supported by GaussDB(DWS) are converted to common date/time values when being read. For details, see [Table 4-14](#).

**Table 4-14** Special values

Input String	Applicable Type	Description
epoch	date, timestamp	1970-01-01 00:00:00+00 (Unix system time zero)
infinity	timestamp	Later than any other timestamps
-infinity	timestamp	Earlier than any other timestamps
now	date, time, timestamp	Start time of the current transaction
today	date, timestamp	Today midnight
tomorrow	date, timestamp	Tomorrow midnight
yesterday	date, timestamp	Yesterday midnight

Input String	Applicable Type	Description
allballs	time	00:00:00.00 UTC

## Interval Input

The input of **reltime** can be any valid interval in TEXT format. It can be a number (negative numbers and decimals are also allowed) or a specific time, which must be in SQL standard format, ISO-8601 format, or POSTGRES format. In addition, the text input needs to be enclosed with single quotation marks (').

For details, see [Table 4-15](#).

**Table 4-15** Interval input

Input	Output	Description
60	2 mons	Numbers are used to indicate intervals. The default unit is day. Decimals and negative numbers are also allowed. Particularly, a negative interval syntactically means how long before.
31.25	1 mons 1 days 06:00:00	
-365	-12 mons -5 days	
1 years 1 mons 8 days 12:00:00	1 years 1 mons 8 days 12:00:00	Intervals are in POSTGRES format. They can contain both positive and negative numbers and are case-insensitive. Output is a simplified POSTGRES interval converted from the input.
-13 months -10 hours	-1 years -25 days -04:00:00	
-2 YEARS +5 MONTHS 10 DAYS	-1 years -6 mons -25 days -06:00:00	
P-1.1Y10M	-3 mons -5 days -06:00:00	Intervals are in ISO-8601 format. They can contain both positive and negative numbers and are case-insensitive. Output is a simplified POSTGRES interval converted from the input.
-12H	-12:00:00	

For example:

```
-- Create a table.
CREATE TABLE reltime_type_tab(col1 character(30), col2 reltime);

-- Insert data.
INSERT INTO reltime_type_tab VALUES ('90', '90');
INSERT INTO reltime_type_tab VALUES ('-366', '-366');
INSERT INTO reltime_type_tab VALUES ('1975.25', '1975.25');
INSERT INTO reltime_type_tab VALUES ('-2 YEARS +5 MONTHS 10 DAYS', '-2 YEARS +5 MONTHS 10 DAYS');
```

```

INSERT INTO reltime_type_tab VALUES ('30 DAYS 12:00:00', '30 DAYS 12:00:00');
INSERT INTO reltime_type_tab VALUES ('P-1.1Y10M', 'P-1.1Y10M');

-- View data.
SELECT * FROM reltime_type_tab;
      col1          |          col2
-----+-----
1975.25             | 5 years 4 mons 29 days
-2 YEARS +5 MONTHS 10 DAYS | -1 years -6 mons -25 days -06:00:00
P-1.1Y10M           | -3 mons -5 days -06:00:00
-366                | -1 years -18:00:00
90                  | 3 mons
30 DAYS 12:00:00    | 1 mon 12:00:00
(6 rows)

-- Delete tables.
DROP TABLE reltime_type_tab;

```

## 4.7 Geometric Types

**Table 4-16** lists the geometric types that can be used in GaussDB(DWS). The most fundamental type, the point, forms the basis for all of the other types.

**Table 4-16** Geometric Type

Name	Storage Space	Description	Representation
point	16 bytes	Point on a plane	(x,y)
lseg	32 bytes	Finite line segment	((x1,y1),(x2,y2))
box	32 bytes	Rectangular Box	((x1,y1),(x2,y2))
path	16+16n bytes	Closed path (similar to polygon)	((x1,y1),...)
path	16+16n bytes	Open path	[(x1,y1),...]
polygon	40+16n bytes	Polygon (similar to closed path)	((x1,y1),...)
circle	24 bytes	Circle	<(x,y),r> (center point and radius)

A rich set of functions and operators is available in GaussDB(DWS) to perform various geometric operations, such as scaling, translation, rotation, and determining intersections. For details, see [Geometric Functions and Operators](#).

### Points

Points are the fundamental two-dimensional building block for geometric types. Values of the **point** type are specified using either of the following syntaxes:

```
( x , y )
x , y
```

where x and y are the respective coordinates, as floating-point numbers.  
Points are output using the first syntax.

## Line Segments

Line segments (**lseg**) are represented by pairs of points. Values of the **lseg** type are specified using any of the following syntaxes:

```
[ ( x1 , y1 ) , ( x2 , y2 ) ]  
( ( x1 , y1 ) , ( x2 , y2 ) )  
( x1 , y1 ) , ( x2 , y2 )  
x1 , y1 , x2 , y2
```

where (x1,y1) and (x2,y2) are the end points of the line segment.

Line segments are output using the first syntax.

## Rectangular Box

Boxes are represented by pairs of points that are opposite corners of the box. Values of the **box** type are specified using any of the following syntaxes:

```
(( x1 , y1 ) , ( x2 , y2 ) )  
( x1 , y1 ) , ( x2 , y2 )  
x1 , y1 , x2 , y2
```

where (x1,y1) and (x2,y2) are any two opposite corners of the box.

Boxes are output using the second syntax.

Any two opposite corners can be supplied on input, but in this order, the values will be reordered as needed to store the upper right and lower left corners.

## Path

Paths are represented by lists of connected points. Paths can be open, where the first and last points in the list are considered not connected, or closed, where the first and last points are considered connected.

Values of the **path** type are specified using any of the following syntaxes:

```
[ ( x1 , y1 ) , ... , ( xn , yn ) ]  
( ( x1 , y1 ) , ... , ( xn , yn ) )  
( x1 , y1 ) , ... , ( xn , yn )  
( x1 , y1 , ... , xn , yn )  
x1 , y1 , ... , xn , yn
```

where the points are the end points of the line segments comprising the path. Square brackets ([]) indicate an open path, while parentheses (()) indicate a closed path. When the outermost parentheses are omitted, as in the third through fifth syntaxes, a closed path is assumed.

Paths are output using the first or second syntax.

## Polygons

Polygons are represented by lists of points (the vertexes of the polygon). Polygons are very similar to closed paths, but are stored differently and have their own set of support functions.

Values of the **polygon** type are specified using any of the following syntaxes:

```
(( x1 , y1 ) , ... , ( xn , yn ) )
(x1 , y1 ) , ... , ( xn , yn )
(x1 , y1 , ... , xn , yn )
x1 , y1 , ... , xn , yn
```

where the points are the end points of the line segments comprising the boundary of the polygon.

Polygons are output using the first syntax.

## Circle

Circles are represented by a center point and radius. Values of the **circle** type are specified using any of the following syntaxes:

```
< ( x , y ) , r >
(( x , y ) , r )
( x , y ) , r
x , y , r
```

where **(x,y)** is the center point and **r** is the radius of the circle.

Circles are output using the first syntax.

## 4.8 Network Address Types

GaussDB(DWS) offers data types to store IPv4, IPv6, and MAC addresses.

It is better to use these types instead of plain text types to store network addresses, because these types offer input error check and specialized operators and functions (see [Network Address Functions and Operators](#)).

**Table 4-17** Network Address Types

Name	Storage Space	Description
cidr	7 or 19 bytes	IPv4 or IPv6 networks
inet	7 or 19 bytes	IPv4 or IPv6 hosts and networks
macaddr	6 bytes	MAC addresses

When sorting **inet** or **cidr** data types, IPv4 addresses will always sort before IPv6 addresses, including IPv4 addresses encapsulated or mapped to IPv6 addresses, such as `::10.2.3.4` or `::ffff:10.4.3.2`.

### cidr

The **cidr** type (Classless Inter-Domain Routing) holds an IPv4 or IPv6 network specification. The format for specifying networks is **address/y** where **address** is the network represented as an IPv4 or IPv6 address, and **y** is the number of bits in the netmask. If **y** is omitted, it is calculated using assumptions from the older classful network numbering system, except it will be at least large enough to include all of the octets written in the input.

**Table 4-18** cidr type input examples

cidr Input	cidr Output	abbrev (cidr)
192.168.100.128/25	192.168.100.128/25	192.168.100.128/25
192.168/24	192.168.0.0/24	192.168.0/24
192.168/25	192.168.0.0/25	192.168.0.0/25
192.168.1	192.168.1.0/24	192.168.1/24
192.168	192.168.0.0/24	192.168.0/24
10.1.2	10.1.2.0/24	10.1.2/24
10.1	10.1.0.0/16	10.1/16
10	10.0.0.0/8	10/8
10.1.2.3/32	10.1.2.3/32	10.1.2.3/32
2001:4f8:3:ba::/64	2001:4f8:3:ba::/64	2001:4f8:3:ba::/64
2001:4f8:3:ba: 2e0:81ff:fe22:d1f1/128	2001:4f8:3:ba: 2e0:81ff:fe22:d1f1/128	2001:4f8:3:ba: 2e0:81ff:fe22:d1f1
::ffff:1.2.3.0/120	::ffff:1.2.3.0/120	::ffff:1.2.3/120
::ffff:1.2.3.0/128	::ffff:1.2.3.0/128	::ffff:1.2.3.0/128

## inet

The **inet** type holds an IPv4 or IPv6 host address, and optionally its subnet, all in one field. The subnet is represented by the number of network address bits present in the host address (the "netmask"). If the netmask is 32 and the address is IPv4, then the value does not indicate a subnet, only a single host. In IPv6, the address length is 128 bits, so 128 bits specify a unique host address.

The input format for this type is **address/y** where **address** is an IPv4 or IPv6 address and **y** is the number of bits in the netmask. If the **/y** portion is missing, the netmask is 32 for IPv4 and 128 for IPv6, so the value represents just a single host. On display, the **/y** portion is suppressed if the netmask specifies a single host.

The essential difference between the **inet** and **cidr** data types is that **inet** accepts values with nonzero bits to the right of the netmask, whereas **cidr** does not.

## macaddr

The **macaddr** type stores MAC addresses, known for example from Ethernet card hardware addresses (although MAC addresses are used for other purposes as well). Input is accepted in the following formats:

```
'08:00:2b:01:02:03'  
'08-00-2b-01-02-03'  
'08002b:010203'  
'08002b-010203'
```



```
'0800.2b01.0203'  
'08002b010203'
```

These examples would all specify the same address. Upper and lower cases are accepted for the digits a through f. Output is always in the first of the forms shown.

## 4.9 Bit String Types

Bit strings are strings of 1's and 0's. They can be used to store bit masks.

GaussDB(DWS) supports two SQL bit types: **bit(n)** and **bit varying(n)**, where **n** is a positive integer.

The **bit** type data must match the length **n** exactly. It is an error to attempt to store shorter or longer bit strings. The **bit varying** data is of variable length up to the maximum length **n**; longer strings will be rejected. Writing **bit** without a length is equivalent to **bit(1)**, while **bit varying** without a length specification means unlimited length.

### NOTE

If one explicitly casts a bit-string value to **bit(n)**, it will be truncated or zero-padded on the right to be exactly **n** bits, without raising an error.

Similarly, if one explicitly casts a bit-string value to **bit varying(n)**, it will be truncated on the right if it is more than **n** bits.

```
-- Create a table:  
CREATE TABLE bit_type_t1  
(  
    BT_COL1 INTEGER,  
    BT_COL2 BIT(3),  
    BT_COL3 BIT VARYING(5)  
) DISTRIBUTE BY REPLICATION;  
  
-- Insert data:  
INSERT INTO bit_type_t1 VALUES(1, B'101', B'00');  
  
-- Specify the type length. An error is reported if an inserted string exceeds this length.  
INSERT INTO bit_type_t1 VALUES(2, B'10', B'101');  
ERROR: bit string length 2 does not match type bit(3)  
CONTEXT: referenced column: bt_col2  
  
-- Specify the type length. Data is converted if it exceeds this length.  
INSERT INTO bit_type_t1 VALUES(2, B'10'::bit(3), B'101');  
  
-- View data:  
SELECT * FROM bit_type_t1;  
bt_col1 | bt_col2 | bt_col3  
-----+-----+-----  
1 | 101 | 00  
2 | 100 | 101  
(2 rows)  
  
-- Delete the tables:  
DROP TABLE bit_type_t1;
```

## 4.10 Text Search Types

GaussDB(DWS) offers two data types that are designed to support full text search. The **tsvector** type represents a document in a form optimized for text search. The **tsquery** type similarly represents a text query.

## tsvector

The **tsvector** type represents a retrieval unit, usually a textual column within a row of a database table, or a combination of such columns. A **tsvector** value is a sorted list of distinct lexemes, which are words that have been normalized to merge different variants of the same word. Sorting and deduplication are done automatically during input. The **to\_tsvector** function is used to parse and normalize a document string. The **to\_tsvector** function is used to parse and normalize a document string.

A **tsvector** value is a sorted list of distinct lexemes, which are words that have been formatted different entries. During segmentation, **tsvector** automatically performs duplicate-elimination to the entries for input in a certain order. For example:

```
SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector;
           tsvector
-----
'a' 'and' 'ate' 'cat' 'fat' 'mat' 'on' 'rat' 'sat'
(1 row)
```

It can be seen from the preceding example that **tsvector** segments a string by spaces, and segmented lexemes are sorted based on their length and alphabetical order. To represent lexemes containing whitespace or punctuation, surround them with quotes:

```
SELECT $$the lexeme ' ' contains spaces$$::tsvector;
           tsvector
-----
' ' 'contains' 'lexeme' 'spaces' 'the'
(1 row)
```

Use double dollar signs (\$\$) to mark entries containing single quotation marks (').

```
SELECT $$the lexeme 'Joe's' contains a quote$$::tsvector;
           tsvector
-----
'Joe's' 'a' 'contains' 'lexeme' 'quote' 'the'
(1 row)
```

Optionally, integer positions can be attached to lexemes:

```
SELECT 'a:1 fat:2 cat:3 sat:4 on:5 a:6 mat:7 and:8 ate:9 a:10 fat:11 rat:12'::tsvector;
           tsvector
-----
'a':1,6,10 'and':8 'ate':9 'cat':3 'fat':2,11 'mat':7 'on':5 'rat':12 'sat':4
(1 row)
```

A position normally indicates the source word's location in the document. Positional information can be used for proximity ranking. Position values range from 1 to 16383. The default maximum value is **16383**. Duplicate positions for the same lexeme are discarded.

Lexemes that have positions can further be labeled with a weight, which can be **A**, **B**, **C**, or **D**. **D** is the default and hence is not shown on output:

```
SELECT 'a:1A fat:2B,4C cat:5D'::tsvector;
           tsvector
-----
'a':1A 'cat':5 'fat':2B,4C
(1 row)
```

Weights are typically used to reflect document structure, for example, by marking title words differently from body words. Text search ranking functions can assign different priorities to the different weight markers.

The following example is the standard usage of the **tsvector** type. For example:

```
SELECT 'The Fat Rats'::tsvector;
      tsvector
-----
'fat' 'rats' 'the'
(1 row)
```

For most English-text-searching applications the above words would be considered non-normalized, which should usually be passed through **to\_tsvector** to normalize the words appropriately for searching:

```
SELECT to_tsvector('english', 'The Fat Rats');
      to_tsvector
-----
'fat':2 'rat':3
(1 row)
```

## tsquery

The **tsquery** type represents a retrieval condition. A **tsquery** value stores lexemes that are to be searched for, and combines them honoring the **Boolean** operators **& (AND)**, **| (OR)**, and **! (NOT)**. Parentheses can be used to enforce grouping of the operators. The **to\_tsquery** and **plainto\_tsquery** functions will normalize lexemes before the lexemes are converted to the **tsquery** type.

```
SELECT 'fat & rat'::tsquery;
      tsquery
-----
'fat' & 'rat'
(1 row)

SELECT 'fat & (rat | cat)'::tsquery;
      tsquery
-----
'fat' & ( 'rat' | 'cat' )
(1 row)

SELECT 'fat & rat & ! cat'::tsquery;
      tsquery
-----
'fat' & 'rat' & '!cat'
(1 row)
```

In the absence of parentheses, **! (NOT)** binds most tightly, and **& (AND)** binds more tightly than **| (OR)**.

Lexemes in a **tsquery** can be labeled with one or more weight letters, which restrict them to match only **tsvector** lexemes with matching weights:

```
SELECT 'fat:ab & cat'::tsquery;
      tsquery
-----
'fat':AB & 'cat'
(1 row)
```

Also, lexemes in a **tsquery** can be labeled with **\*** to specify prefix matching:

```
SELECT 'super:*'::tsquery;
      tsquery
-----
```

```
'super':*  
(1 row)
```

This query will match any word in a **tsvector** that begins with "super".

Note that prefixes are first processed by text search configurations, which means the following example returns true:

```
SELECT to_tsvector( 'postgraduate' ) @@ to_tsquery( 'postgres:*' ) AS RESULT;  
result  
-----  
t  
(1 row)
```

because **postgres** gets stemmed to **postgr**:

```
SELECT to_tsquery('postgres:*');  
to_tsquery  
-----  
'postgr':*  
(1 row)
```

which then matches **postgraduate**.

'**Fat:ab & Cats**' is normalized to the **tsquery** type as follows:

```
SELECT to_tsquery('Fat:ab & Cats');  
to_tsquery  
-----  
'fat':AB & 'cat'  
(1 row)
```

## 4.11 UUID Type

The data type **UUID** stores Universally Unique Identifiers (UUID) as defined by RFC 4122, ISO/IEF 9834-8:2005, and related standards. This identifier is a 128-bit quantity that is generated by an algorithm chosen to make it very unlikely that the same identifier will be generated by anyone else in the known universe using the same algorithm.

Therefore, for distributed systems, these identifiers provide a better uniqueness guarantee than sequence generators, which are only unique within a single database.

A UUID is written as a sequence of lower-case hexadecimal digits, in several groups separated by hyphens, specifically a group of 8 digits followed by three groups of 4 digits followed by a group of 12 digits, for a total of 32 digits representing the 128 bits. An example of a UUID in this standard form is:

```
a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11
```

GaussDB(DWS) also accepts the following alternative forms for input: use of upper-case letters and digits, the standard format surrounded by braces, omitting some or all hyphens, and adding a hyphen after any group of four digits. For example:

```
A0EEBC99-9C0B-4EF8-BB6D-6BB9BD380A11  
{a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11}  
a0eebc999c0b4ef8bb6d6bb9bd380a11  
a0ee-bc99-9c0b-4ef8-bb6d-6bb9-bd38-0a11
```

Output is always in the standard form.

## 4.12 JSON Types

JSON data types are for storing JavaScript Object Notation (JSON) data. Such data can also be stored as TEXT, but the JSON data type has the advantage of checking that each stored value is a valid JSON value.

For functions that support the JSON data type, see [JSON Functions](#).

## 4.13 HLL Data Types

HyperLoglog (HLL) is an approximation algorithm for efficiently counting the number of distinct values in a data set. It features faster computing and lower space usage. You only need to store HLL data structures, instead of data sets. When new data is added to a data set, make hash calculation on the data and insert the result to an HLL. Then, you can obtain the final result based on the HLL.

[Table 4-19](#) compares HLL with other algorithms.

**Table 4-19** Comparison between HLL and other algorithms

Item	Sorting Algorithm	Hash Algorithm	HLL
Time complexity	$O(n \log n)$	$O(n)$	$O(n)$
Space complexity	$O(n)$	$O(n)$	1280 bytes
Error rate	0	0	$\approx 2\%$
Storage space requirement	Size of raw data	Size of raw data	1280 bytes

HLL has advantages over others in the computing speed and storage space requirement. In terms of time complexity, the sorting algorithm needs  $O(n \log n)$  time for sorting, and the hash algorithm and HLL need  $O(n)$  time for full table scanning. In terms of storage space requirements, the sorting algorithm and hash algorithm need to store raw data before collecting statistics, whereas the HLL algorithm needs to store only the HLL data structures rather than the raw data, and thereby occupying a fixed space of only 1280 bytes.

**NOTICE**

- In default specifications, the maximum number of distinct values is 1.6e plus 12, and the maximum error rate is only 2.3%. If a calculation result exceeds the maximum number, the error rate of the calculation result will increase, or the calculation will fail and an error will be reported.
- When using this feature for the first time, you need to evaluate the distinct values of the service, properly select configuration parameters, and perform verification to ensure that the accuracy meets requirements.
  - When default parameter configuration is used, the calculated number of distinct values is 1.6e plus 12. If the calculated result is **NaN**, you need to adjust **log2m** and **regwidth** to accommodate more distinct values.
  - The hash algorithm has an extremely low probability of collision. However, you are still advised to select 2 or 3 hash seeds for verification when using the hash algorithm for the first time. If there is only a small difference between the distinct values, you can select any one of the seeds as the hash seed.

**Table 4-20** describes main HLL data structures.

**Table 4-20** Main HLL data structures

Data Type	Description
hll	Its size is always 1280 bytes, which can be directly used to calculate the number of distinct values.

The following describes HLL application scenarios.

- Scenario 1: "Hello World"

The following example shows how to use the HLL data type:

```
-- Create a table with the HLL data type:
create table helloworld (id integer, set hll);

-- Insert an empty HLL to the table:
insert into helloworld(id, set) values (1, hll_empty());

-- Add a hashed integer to the HLL:
update helloworld set set = hll_add(set, hll_hash_integer(12345)) where id = 1;

-- Add a hashed string to the HLL:
update helloworld set set = hll_add(set, hll_hash_text('hello world')) where id = 1;

-- Obtain the number of distinct values of the HLL:
select hll_cardinality(set) from helloworld where id = 1;
 hll_cardinality
-----
                2
(1 row)
```

- Scenario 2: Collect statistics about website visitors.

The following example shows how an HLL collects statistics on the number of users visiting a website within a period of time:

```
-- Create a raw data table to show that a user has visited the website at a certain time:
create table facts (
```

```

        date      date,
        user_id   integer
    );

-- Construct data to show the users who have visited the website in a day:
insert into facts values ('2019-02-20', generate_series(1,100));
insert into facts values ('2019-02-21', generate_series(1,200));
insert into facts values ('2019-02-22', generate_series(1,300));
insert into facts values ('2019-02-23', generate_series(1,400));
insert into facts values ('2019-02-24', generate_series(1,500));
insert into facts values ('2019-02-25', generate_series(1,600));
insert into facts values ('2019-02-26', generate_series(1,700));
insert into facts values ('2019-02-27', generate_series(1,800));

-- Create another table and specify an HLL column:
create table daily_uniques (
    date      date UNIQUE,
    users     hll
);

-- Group data by date and insert the data into the HLL:
insert into daily_uniques(date, users)
select date, hll_add_agg(hll_hash_integer(user_id))
from facts
group by 1;

-- Calculate the numbers of users visiting the website every day:
select date, hll_cardinality(users) from daily_uniques order by date;
      date      | hll_cardinality
-----+-----
2019-02-20 00:00:00 |          100
2019-02-21 00:00:00 | 203.81335588808
2019-02-22 00:00:00 | 308.048239950384
2019-02-23 00:00:00 | 410.529188080374
2019-02-24 00:00:00 | 513.263875705319
2019-02-25 00:00:00 | 609.271181107416
2019-02-26 00:00:00 | 702.941844662509
2019-02-27 00:00:00 | 792.249946595237
(8 rows)

-- Calculate the number of users who had visited the website in the week from February 20, 2019 to
February 26, 2019:
select hll_cardinality(hll_union_agg(users)) from daily_uniques where date >= '2019-02-20'::date and
date <= '2019-02-26'::date;
      hll_cardinality
-----
702.941844662509
(1 row)

-- Calculate the number of users who had visited the website yesterday but have not visited the
website today:
SELECT date, (#hll_union_agg(users) OVER two_days) - #users AS lost_uniques FROM daily_uniques
WINDOW two_days AS (ORDER BY date ASC ROWS 1
PRECEDING);
      date      | lost_uniques
-----+-----
2019-02-20 00:00:00 |          0
2019-02-21 00:00:00 |          0
2019-02-22 00:00:00 |          0
2019-02-23 00:00:00 |          0
2019-02-24 00:00:00 |          0
2019-02-25 00:00:00 |          0
2019-02-26 00:00:00 |          0
2019-02-27 00:00:00 |          0
(8 rows)

```

- Scenario 3: The data to be inserted does not meet the requirements of the HLL data structure.

When inserting data into a column of the HLL type, ensure that the data meets the requirements of the HLL data structure. If the data does not meet the requirements after being parsed, an error will be reported. In the following example, `E\1234` to be inserted does not meet the requirements of the HLL data structure after being parsed. As a result, an error is reported.

```
create table test(id integer, set hll);
insert into test values(1, 'E\1234');
ERROR: unknown schema version 4
```

## 4.14 Object Identifier Types

Object identifiers (OIDs) are used internally by GaussDB(DWS) as primary keys for various system catalogs. OIDs are not added to user-created tables by the system. The **OID** type represents an object identifier.

The **OID** type is currently implemented as an unsigned four-byte integer. So, using a user-created table's **OID** column as a primary key is discouraged.

**Table 4-21** Object identifier types

Name	Reference	Description	Examples
OID	-	Numeric object identifier	564182
CID	-	A command identifier. This is the data type of the system columns <b>cmn</b> and <b>cmx</b> . Command identifiers are 32-bit quantities.	-
XID	-	A transaction identifier. This is the data type of the system columns <b>xmin</b> and <b>xmax</b> . Transaction identifiers are also 32-bit quantities.	-
TID	-	A row identifier. This is the data type of the system column <b>ctid</b> . A row ID is a pair (block number, tuple index within block) that identifies the physical location of the row within its table.	-
REGCONFIG	pg_ts_config	Text search configuration	english
REGDICTIONARY	pg_ts_dict	Text search dictionary	simple
REGOPER	pg_operator	Operator name	+



Name	Reference	Description	Examples
REGOPERATOR	pg_operator	Operator with argument types	*(integer,integer) or -(NONE,integer)
REGPROC	pg_proc	Indicates the name of the function.	sum
REGPROCEDURE	pg_proc	Function with argument types	sum(int4)
REGCLASS	pg_class	Relation name	pg_type
REGTYPE	pg_type	Data type name	integer

The **OID** type is used for a column in the database system catalog.

For example:

```
SELECT oid FROM pg_class WHERE relname = 'pg_type';
oid
-----
1247
(1 row)
```

The alias type for **OID** is **REGCLASS** which allows simplified search for **OID** values.

For example:

```
SELECT attrelid,attname,atttypid,attstattarget FROM pg_attribute WHERE attrelid = 'pg_type'::REGCLASS;
attrelid | attname | atttypid | attstattarget
-----+-----+-----+-----
1247 | xc_node_id | 23 | 0
1247 | tableoid | 26 | 0
1247 | cmax | 29 | 0
1247 | xmax | 28 | 0
1247 | cmin | 29 | 0
1247 | xmin | 28 | 0
1247 | oid | 26 | 0
1247 | ctid | 27 | 0
1247 | typname | 19 | -1
1247 | typnamespace | 26 | -1
1247 | typowner | 26 | -1
1247 | typplen | 21 | -1
1247 | typbyval | 16 | -1
1247 | typtype | 18 | -1
1247 | typcategory | 18 | -1
1247 | typispreferred | 16 | -1
1247 | typisdefined | 16 | -1
1247 | typdelim | 18 | -1
1247 | typrelid | 26 | -1
1247 | typelem | 26 | -1
1247 | typarray | 26 | -1
1247 | typinput | 24 | -1
1247 | typoutput | 24 | -1
1247 | typreceive | 24 | -1
1247 | typsend | 24 | -1
1247 | typmodin | 24 | -1
1247 | typmodout | 24 | -1
1247 | typanalyze | 24 | -1
1247 | typalign | 18 | -1
1247 | typstorage | 18 | -1
```

1247		typnotnull		16		-1
1247		typbasetype		26		-1
1247		tytypmod		23		-1
1247		typndims		23		-1
1247		typcollation		26		-1
1247		typdefaultbin		194		-1
1247		typdefault		25		-1
1247		typacl		1034		-1
(38 rows)						

## 4.15 Pseudo-Types

GaussDB(DWS) has a number of special-purpose entries that are collectively called pseudo-types. A pseudo-type cannot be used as a column data type, but it can be used to declare a function's argument or result type.

Each of the available pseudo-types is useful in situations where a function's behavior does not correspond to simply taking or returning a value of a specific SQL data type. [Table 4-22](#) lists all pseudo-types.

**Table 4-22** Pseudo-Types

Name	Description
any	Indicates that a function accepts any input data type.
anyelement	Indicates that a function accepts any data type.
anyarray	Indicates that a function accepts any array data type.
anynonarray	Indicates that a function accepts any non-array data type.
anyenum	Indicates that a function accepts any enum data type.
anyrange	Indicates that a function accepts any range data type.
cstring	Indicates that a function accepts or returns a null-terminated C string.
internal	Indicates that a function accepts or returns a server-internal data type.
language_handler	Indicates that a procedural language call handler is declared to return <b>language_handler</b> .
fdw_handler	Indicates that a foreign-data wrapper handler is declared to return <b>fdw_handler</b> .
record	Identifies a function returning an unspecified row type.
trigger	Indicates that a trigger function is declared to return <b>trigger</b> .
void	Indicates that a function returns no value.
opaque	Indicates an obsolete type name that formerly served all the above purposes.

Functions coded in C (whether built in or dynamically loaded) can be declared to accept or return any of these pseudo data types. It is up to the function author to ensure that the function will behave safely when a pseudo-type is used as an argument type.

Functions coded in procedural languages can use pseudo-types only as allowed by their implementation languages. At present the procedural languages all forbid use of a pseudo-type as argument type, and allow only **void** and **record** as a result type. Some also support polymorphic functions using the **anyelement**, **anyarray**, **anynonarray**, **anyenum**, and **anyrange** types.

The **internal** pseudo-type is used to declare functions that are meant only to be called internally by the database system, and not by direct call in an SQL query. If a function has at least one **internal**-type argument, it cannot be called from SQL. You are not advised to create any function that is declared to return **internal** unless the function has at least one **internal** argument.

For example:

```
-- Create the showall() function:
CREATE OR REPLACE FUNCTION showall() RETURNS SETOF record
AS $$ SELECT count(*) from tpchs.store_sales where ss_customer_sk = 9692; $$
LANGUAGE SQL;

-- Invoke the showall() function:
SELECT showall();
showall
-----
(35)
(1 row)

-- Delete the function:
DROP FUNCTION showall();
```

## 4.16 Data Types Supported by Column-Store Tables

[Table 4-23](#) lists the data types supported by column-store tables.

**Table 4-23** Data types supported by column-store tables

Category	Data Type	Length	Supported
Numeric types	smallint	2	Yes
	integer	4	Yes
	bigint	8	Yes
	decimal	Variable length	Yes
	numeric	Variable length	Yes
	real	4	Yes

Category	Data Type	Length	Supported
	double precision	8	Yes
	smallserial	2	Yes
	serial	4	Yes
	bigserial	8	Yes
Monetary types	money	8	Yes
Character types	character varying(n), varchar(n)	Variable length	Yes
	character(n), char(n)	n	Yes
	character, char	1	Yes
	text	Variable length	Yes
	nvarchar2	Variable length	Yes
	name	64	No
Date/time types	timestamp with time zone	8	Yes
	timestamp without time zone	8	Yes
	date	4	Yes
	time without time zone	8	Yes
	time with time zone	12	Yes
	interval	16	Yes
Large objects	clob	Variable length	Yes
	blob	Variable length	No
Others	...	...	No

# 5 Constant and Macro

**Table 5-1** lists the constants and macros that can be used in GaussDB(DWS).

**Table 5-1** Constants and macros

Parameter	Description	Examples
CURRENT_CATALOG	Specifies the current database.	SELECT CURRENT_CATALOG; current_database ----- postgres (1 row)
CURRENT_ROLE	Current user	SELECT CURRENT_ROLE; current_user ----- dbadmin (1 row)
CURRENT_SCHEMA	Current database model	SELECT CURRENT_SCHEMA; current_schema ----- public (1 row)
CURRENT_USER	Current user	SELECT CURRENT_USER; current_user ----- dbadmin (1 row)
LOCALTIMESTAMP	Current session time (without time zone)	SELECT LOCALTIMESTAMP; timestamp ----- 2015-10-10 15:37:30.968538 (1 row)
NULL	This parameter is left blank.	-
SESSION_USER	Current system user	SELECT SESSION_USER; session_user ----- dbadmin (1 row)

Parameter	Description	Examples
SYSDATE	Current system date	SELECT SYSDATE; sysdate ----- 2015-10-10 15:48:53 (1 row)
USER	Current user, also called <b>CURRENT_USER</b>	SELECT USER; current_user ----- dbadmin (1 row)

# 6 Functions and Operators

## 6.1 Logical Operators

The usual logical operators include AND, OR, and NOT. SQL uses a three-valued logical system with true, false, and null, which represents "unknown". Their priorities are NOT > AND > OR.

**Table 6-1** lists operation rules, where a and b represent logical expressions.

**Table 6-1** Operation rules

a	b	a AND b Result	a OR b Result	NOT a Result
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE
TRUE	NULL	NULL	TRUE	FALSE
FALSE	FALSE	FALSE	FALSE	TRUE
FALSE	NULL	FALSE	NULL	TRUE
NULL	NULL	NULL	NULL	NULL

 **NOTE**

The operators AND and OR are commutative, that is, you can switch the left and right operand without affecting the result.

## 6.2 Comparison Operators

Comparison operators are available for all data types and return Boolean values.

All comparison operators are binary operators. Only data types that are the same or can be implicitly converted can be compared using comparison operators.

**Table 6-2** describes comparison operators provided by GaussDB(DWS).

**Table 6-2** Comparison Operators

Operators	Description
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
=	Equality
<> or !=	Inequality

Comparison operators are available for all relevant data types. All comparison operators are binary operators that returned values of Boolean type. Expressions like **1 < 2 < 3** are invalid. (Because there is no comparison operator to compare a Boolean value with **3**.)

## 6.3 Character Processing Functions and Operators

String functions and operators provided by GaussDB(DWS) are for concatenating strings with each other, concatenating strings with non-strings, and matching the patterns of strings.

- `bit_length(string)`

Description: Specifies the number of bits occupied by a string.

Return type: int

For example:

```
SELECT bit_length('world');
bit_length
-----
         40
(1 row)
```

- `btrim(string text [, characters text])`

Description: Removes the longest string consisting only of characters in **characters** (a space by default) from the start and end of **string**.

Return type: text

For example:

```
SELECT btrim('string', 'ing');
btrim
-----
sr
(1 row)
```

- `char_length(string)` or `character_length(string)`

Description: Number of characters in a string

Return type: int



For example:

```
SELECT char_length('hello');
char_length
-----
5
(1 row)
```

- instr(text,text,int,int)

Description: **FROM int** indicates the start position of the replacement in the first string. **for int** indicates the number of characters replaced in the first string.

Return type: int

For example:

```
SELECT instr( 'abcdabcdabcd', 'bcd', 2, 2 );
instr
-----
6
(1 row)
```

- lengthb(text/bpchar)

Description: Obtains the number of bytes of a specified string.

Return type: int

For example:

```
SELECT lengthb('hello');
lengthb
-----
5
(1 row)
```

- left(str text, n int)

Description: Returns first **n** characters in the string. When **n** is negative, return all but the last **|n|** characters.

Return type: text

For example:

```
SELECT left('abcde', 2);
left
-----
ab
(1 row)
```

- length(string bytea, encoding name )

Description: Number of characters in **string** in the given **encoding**. The **string** must be valid in this encoding.

Return type: int

For example:

```
SELECT length('jose', 'UTF8');
length
-----
4
(1 row)
```

- lpad(string text, length int [, fill text])

Description: Fills up the string to the specified length by appending the characters **fill** (a space by default). If the **string** is already longer than **length** then it is truncated (on the right).

Return type: text

For example:

```
SELECT lpad('hi', 5, 'xyza');
lpad
-----
xyzhi
(1 row)
```

- `octet_length(string)`

Description: Number of bytes in a string

Return type: int

For example:

```
SELECT octet_length('jose');
octet_length
-----
4
(1 row)
```

- `overlay(string placing string FROM int [for int])`

Description: Replaces substring. **FROM int** indicates the start position of the replacement in the first string. **for int** indicates the number of characters replaced in the first string.

Return type: text

For example:

```
SELECT overlay('hello' placing 'world' from 2 for 3 );
overlay
-----
hworldo
(1 row)
```

- `position(substring in string)`

Description: Location of specified substring

Return type: int

For example:

```
SELECT position('ing' in 'string');
position
-----
4
(1 row)
```

- `pg_client_encoding()`

Description: Current client encoding name

Return type: name

For example:

```
SELECT pg_client_encoding();
pg_client_encoding
-----
UTF8
(1 row)
```

- `quote_ident(string text)`

Description: Returns the given string suitably quoted to be used as an identifier in an SQL statement string (quotation marks are used as required). Quotes are added only if necessary (that is, if the string contains non-identifier characters or would be case-folded). Embedded quotes are properly doubled.

Return type: text

For example:

```
SELECT quote_ident('hello world');
quote_ident
-----
"hello world"
(1 row)
```

- `quote_literal(string text)`

Description: Returns the given string suitably quoted to be used as a string literal in an SQL statement string (quotation marks are used as required).

Return type: text

For example:

```
SELECT quote_literal('hello');
quote_literal
-----
'hello'
(1 row)
```

If command similar to the following exists, text will be escaped.

```
SELECT quote_literal(E'O\hello');
quote_literal
-----
'O"hello'
(1 row)
```

If command similar to the following exists, backslash will be properly doubled.

```
SELECT quote_literal('O\hello');
quote_literal
-----
E'O\\hello'
(1 row)
```

If the parameter is null, return **NULL**. If the parameter may be null, you are advised to use **quote\_nullable**.

```
SELECT quote_literal(NULL);
quote_literal
-----
(1 row)
```

- `quote_literal(value anyelement)`

Description: Coerces the given value to text and then quotes it as a literal.

Return type: text

For example:

```
SELECT quote_literal(42.5);
quote_literal
-----
'42.5'
(1 row)
```

If command similar to the following exists, the given value will be escaped.

```
SELECT quote_literal(E'O\42.5');
quote_literal
-----
'O"42.5'
(1 row)
```

If command similar to the following exists, backslash will be properly doubled.

```
SELECT quote_literal('O\42.5');
quote_literal
-----
E'O\\42.5'
(1 row)
```

- `quote_nullable(string text)`

Description: Returns the given string suitably quoted to be used as a string literal in an SQL statement string (quotation marks are used as required).

Return type: text

For example:

```
SELECT quote_nullable('hello');
quote_nullable
-----
'hello'
(1 row)
```

If command similar to the following exists, text will be escaped.

```
SELECT quote_nullable(E'O'hello);
quote_nullable
-----
'O"hello'
(1 row)
```

If command similar to the following exists, backslash will be properly doubled.

```
SELECT quote_nullable('O\hello');
quote_nullable
-----
E'O\\hello'
(1 row)
```

If the parameter is null, return **NULL**.

```
SELECT quote_nullable(NULL);
quote_nullable
-----
NULL
(1 row)
```

- `quote_nullable(value anyelement)`

Description: Converts the given value to text and then quotes it as a literal.

Return type: text

For example:

```
SELECT quote_nullable(42.5);
quote_nullable
-----
'42.5'
(1 row)
```

If command similar to the following exists, the given value will be escaped.

```
SELECT quote_nullable(E'O'42.5);
quote_nullable
-----
'O"42.5'
(1 row)
```

If command similar to the following exists, backslash will be properly doubled.

```
SELECT quote_nullable('O\42.5');
quote_nullable
-----
E'O\\42.5'
(1 row)
```

If the parameter is null, return **NULL**.

```
SELECT quote_nullable(NULL);
quote_nullable
-----
NULL
(1 row)
```

- `substring(string [from int] [for int])`

Description: Extracts a substring. **from int** indicates the start position of the truncation. **for int** indicates the number of characters truncated.

Return type: text

For example:

```
SELECT substring('Thomas' from 2 for 3);
substring
-----
hom
(1 row)
```

- `substring(string from pattern)`

Description: Extracts substring matching POSIX regular expression. It returns the text that matches the pattern. If no match record is found, a null value is returned.

Return type: text

For example:

```
SELECT substring('Thomas' from '...$');
substring
-----
mas
(1 row)
SELECT substring('foobar' from 'o(.)b');
result
-----
o
(1 row)
SELECT substring('foobar' from '(o(.)b)');
result
-----
oob
(1 row)
```

#### NOTE

If the POSIX pattern contains any parentheses, the portion of the text that matched the first parenthesized sub-expression (the one whose left parenthesis comes first) is returned. You can put parentheses around the whole expression if you want to use parentheses within it without triggering this exception.

- `substring(string from pattern for escape)`

Description: Extracts substring matching SQL regular expression. The specified pattern must match the entire data string, or else the function fails and returns null. To indicate the part of the pattern that should be returned on success, the pattern must contain two occurrences of the escape character followed by a double quote ("). The text matching the portion of the pattern between these markers is returned.

Return type: text

For example:

```
SELECT substring('Thomas' from '%#"o_a#"_' for '#');
substring
-----
oma
(1 row)
```

- `rawcat(raw,raw)`

Description: Indicates the string concatenation functions.

Return type: raw

For example:

```
SELECT rawcat('ab','cd');
rawcat
-----
ABCD
(1 row)
```

- `regexp_like(text,text,text)`

Description: Indicates the mode matching function of a regular expression.

Return type: bool

For example:

```
SELECT regexp_like('str','[ac]');
regexp_like
-----
f
(1 row)
```

- `regexp_substr(text,text)`

Description: Extracts substrings from a regular expression. Its function is similar to **substr**. When a regular expression contains multiple parallel brackets, it also needs to be processed.

Return type: text

For example:

```
SELECT regexp_substr('str','[ac]');
regexp_substr
-----
(1 row)
```

- `regexp_matches(string text, pattern text [, flags text])`

Description: Returns all captured substrings resulting from matching a POSIX regular expression against the **string**. If the pattern does not match, the function returns no rows. If the pattern contains no parenthesized sub-expressions, then each row returned is a single-element text array containing the substring matching the whole pattern. If the pattern contains parenthesized sub-expressions, the function returns a text array whose *n*th element is the substring matching the *n*th parenthesized sub-expression of the pattern.

The optional **flags** argument contains zero or multiple single-letter flags that change function behavior. **i** indicates that the matching is not related to uppercase and lowercase. **g** indicates that each matching substring is replaced, instead of replacing only the first one.

#### NOTICE

If the last parameter is provided but the parameter value is an empty string (") and the SQL compatibility mode of the database is set to ORA, the returned result is an empty set. This is because the ORA compatible mode treats the empty string (") as **NULL**. To resolve this problem, you can:

- Change the database SQL compatibility mode to TD.
- Do not provide the last parameter or do not set the last parameter to an empty string.

Return type: setof text[]

For example:

```

SELECT regexp_matches('foobarbequebaz', '(bar)(beque)');
regexp_matches
-----
{bar,beque}
(1 row)
SELECT regexp_matches('foobarbequebaz', 'barbeque');
regexp_matches
-----
{barbeque}
(1 row)
SELECT regexp_matches('foobarbequebazilbarfbonk', '(b[^b]+)(b[^b]+)', 'g');
result
-----
{bar,beque}
{bazil,barf}
(2 rows)

```

- `regexp_split_to_array(string text, pattern text [, flags text ])`

Description: Splits **string** using a POSIX regular expression as the delimiter. The `regexp_split_to_array` function behaves the same as `regexp_split_to_table`, except that `regexp_split_to_array` returns its result as an array of text.

Return type: text[]

For example:

```

SELECT regexp_split_to_array('hello world', E'\\s+');
regexp_split_to_array
-----
{hello,world}
(1 row)

```

- `regexp_split_to_table(string text, pattern text [, flags text])`

Description: Splits **string** using a POSIX regular expression as the delimiter. If there is no match to the pattern, the function returns the string. If there is at least one match, for each match it returns the text from the end of the last match (or the beginning of the string) to the beginning of the match. When there are no more matches, it returns the text from the end of the last match to the end of the string.

The **flags** parameter is a text string containing zero or more single-letter flags that change the function's behavior. **i** indicates that the matching is not related to uppercase and lowercase. **g** indicates that each matching substring is replaced, instead of replacing only the first one.

Return type: setof text

For example:

```

SELECT regexp_split_to_table('hello world', E'\\s+');
regexp_split_to_table
-----
hello
world
(2 rows)

```

- `repeat(string text, number int )`

Description: text

Return type: string repeated for *number* times

For example:

```

SELECT repeat('Pg', 4);
repeat
-----
PgPgPgPg
(1 row)

```

- `replace(string text, from text, to text)`  
 Description: Replaces all occurrences in **string** of substring **from** with substring **to**.  
 Return type: text  
 For example:  

```
SELECT replace('abcdefabcdef', 'cd', 'XXX');
replace
-----
abXXXefabXXXef
(1 row)
```
- `reverse(str)`  
 Description: Returns reversed string.  
 Return type: text  
 For example:  

```
SELECT reverse('abcde');
reverse
-----
edcba
(1 row)
```
- `right(str text, n int)`  
 Description: Returns the last **n** characters in the string. When **n** is negative, return all but first **|n|** characters.  
 Return type: text  
 For example:  

```
SELECT right('abcde', 2);
right
-----
de
(1 row)

SELECT right('abcde', -2);
right
-----
cde
(1 row)
```
- `rpad(string text, length int [, fill text])`  
 Description: Fills up the string to length by appending the characters fill (a space by default). If the string is already longer than length then it is truncated.  
 Return type: text  
 For example:  

```
SELECT rpad('hi', 5, 'xy');
rpad
-----
hixyx
(1 row)
```
- `rtrim(string text [, characters text])`  
 Description: Removes the longest string containing only characters from characters (a space by default) from the end of string.  
 Return type: text  
 For example:  

```
SELECT rtrim('trimxxx', 'x');
rtrim
```



- ```
-----
trim
(1 row)
```
- `sys_context ( 'namespace' , 'parameter')`

Description: Obtains and returns the parameter values of a specified **namespace**.

Return type: text

For example:

```
SELECT SYS_CONTEXT ( 'postgres' , 'archive_mode');
sys_context
-----
(1 row)
```
  - `substrb(text,int,int)`

Description: Extracts a substring. The first **int** indicates the start position of the subtraction. The second **int** indicates the number of characters subtracted.

Return type: text

For example:

```
SELECT substrb('string',2,3);
substrb
-----
tri
(1 row)
```
  - `substrb(text,int)`

Description: Extracts a substring. **int** indicates the start position of the subtraction.

Return type: text

For example:

```
SELECT substrb('string',2);
substrb
-----
tring
(1 row)
```
  - `substr(bytea,from,count)`

Description: Extracts a substring from **bytea**. **from** specifies the position where the extraction starts. **count** specifies the length of the extracted substring.

Return type: text

For example:

```
SELECT substr('string',2,3);
substr
-----
tri
(1 row)
```
  - `string || string`

Description: Concatenates strings.

Return type: text

For example:

```
SELECT 'MPP'||'DB' AS RESULT;
result
-----
```

- ```
MPPDB
(1 row)
```
- string || non-string or non-string || string**

Description: Concatenates strings and non-strings.

Return type: text

For example:

```
SELECT 'Value: '||42 AS RESULT;
result
-----
Value: 42
(1 row)
```
  - split\_part(string text, delimiter text, field int)**

Description: Splits **string** on **delimiter** and returns the **fieldth** column (counting from text of the first appeared delimiter).

Return type: text

For example:

```
SELECT split_part('abc~@~def~@~ghi', '~@~', 2);
split_part
-----
def
(1 row)
```
  - strpos(string, substring)**

Description: Specifies the position of a substring. It is the same as **position(substring in string)**. However, the parameter sequences of them are reversed.

Return type: int

For example:

```
SELECT strpos('source', 'rc');
strpos
-----
4
(1 row)
```
  - to\_hex(number int or bigint)**

Description: Converts number to a hexadecimal expression.

Return type: text

For example:

```
SELECT to_hex(2147483647);
to_hex
-----
7fffffff
(1 row)
```
  - translate(string text, from text, to text)**

Description: Any character in **string** that matches a character in the **from** set is replaced by the corresponding character in the **to** set. If **from** is longer than **to**, extra characters occurred in **from** are removed.

Return type: text

For example:

```
SELECT translate('12345', '143', 'ax');
translate
-----
a2x5
(1 row)
```

- length(string)

Description: Obtains the number of characters in a string.

Return type: integer

For example:

```
SELECT length('abcd');
length
-----
      4
(1 row)
```

- lengthb(string)

Description: Obtains the number of characters in a string. The value depends on character sets (GBK and UTF8).

Return type: integer

For example:

```
SELECT lengthb('Chinese');
lengthb
-----
       7
(1 row)
```

- substr(string,from)

Description:

Extracts substrings from a string.

**from** indicates the start position of the extraction.

- If **from** starts at 0, the value **1** is used.
- If the value of **from** is positive, all characters from **from** to the end are extracted.
- If the value of **from** is negative, the last n characters in the string are extracted, in which n indicates the absolute value of **from**.

Return type: varchar

For example:

If the value of **from** is positive:

```
SELECT substr('ABCDEF',2);
substr
-----
BCDEF
(1 row)
```

If the value of **from** is negative:

```
SELECT substr('ABCDEF',-2);
substr
-----
EF
(1 row)
```

- substr(string,from,count)

Description:

Extracts substrings from a string.

**from** indicates the start position of the extraction.

"count" indicates the length of the extracted substring.

- If **from** starts at 0, the value **1** is used.

- If the value of **from** is positive, extract **count** characters starting from **from**.
- If the value of **from** is negative, extract the last **n count** characters in the string, in which **n** indicates the absolute value of **from**.
- If the value of "count" is smaller than 1, null is returned.

Return type: varchar

For example:

If the value of **from** is positive:

```
SELECT substr('ABCDEF',2,2);
substr
-----
BC
(1 row)
```

If the value of **from** is negative:

```
SELECT substr('ABCDEF',-3,2);
substr
-----
DE
(1 row)
```

- **substrb(string,from)**

Description: The functionality of this function is the same as that of **SUBSTR(string,from)**. However, the calculation unit is byte.

Return type: bytea

For example:

```
SELECT substrb('ABCDEF',-2);
substrb
-----
EF
(1 row)
```

- **substrb(string,from,count)**

Description: The functionality of this function is the same as that of **SUBSTR(string,from,count)**. However, the calculation unit is byte.

Return type: bytea

For example:

```
SELECT substrb('ABCDEF',2,2);
substrb
-----
BC
(1 row)
```

- **trim([leading |trailing |both] [characters] from string)**

Description: Removes the longest string containing only the characters (a space by default) from the start/end/both ends of the string.

Return type: varchar

For example:

```
SELECT trim(BOTH 'x' FROM 'xTomxx');
btrim
-----
Tom
(1 row)
SELECT trim(LEADING 'x' FROM 'xTomxx');
ltrim
-----
```

```
Tomxx
(1 row)
SELECT trim(TRAILING 'x' FROM 'xTomxx');
rtrim
-----
xTom
(1 row)
```

- rtrim(string [, characters])

Description: Removes the longest string containing only characters from characters (a space by default) from the end of string.

Return type: varchar

For example:

```
SELECT rtrim('TRIMxxxx','x');
rtrim
-----
TRIM
(1 row)
```

- ltrim(string [, characters])

Description: Removes the longest string containing only characters from characters (a space by default) from the start of string.

Return type: varchar

For example:

```
SELECT ltrim('xxxTRIM','x');
ltrim
-----
TRIM
(1 row)
```

- upper(string)

Description: Converts the string into the uppercase.

Return type: varchar

For example:

```
SELECT upper('tom');
upper
-----
TOM
(1 row)
```

- lower(string)

Description: Converts the string into the lowercase.

Return type: varchar

For example:

```
SELECT lower('TOM');
lower
-----
tom
(1 row)
```

- rpad(string varchar, length int [, fill varchar])

Description: Fills up the string to length by appending the characters fill (a space by default). If the string is already longer than length then it is truncated.

**length** in GaussDB(DWS) indicates the character length. One Chinese character is counted as one character.

Return type: varchar

For example:

```
SELECT rpad('hi',5,'xyza');
rpad
-----
hixyx
(1 row)
SELECT rpad('hi',5,'abcdefg');
rpad
-----
hiabc
(1 row)
```

- `instr(string,substring[,position,occurrence])`

Description: Queries and returns the value of the substring position that occurs the occurrence (first by default) times from the position (1 by default) in the string.

- If the value of "position" is 0, 0 is returned.
- If the value of position is negative, searches backwards from the last nth character in the string, in which **n** indicates the absolute value of position.

In this function, the calculation unit is character. One Chinese character is one character.

Return type: integer

For example:

```
SELECT instr('corporate floor','or', 3);
instr
-----
5
(1 row)
SELECT instr('corporate floor','or',-3,2);
instr
-----
2
(1 row)
```

- `initcap(string)`

Description: The first letter of each word in the string is converted into the uppercase and the other letters are converted into the lowercase.

Return type: text

For example:

```
SELECT initcap('hi THOMAS');
initcap
-----
Hi Thomas
(1 row)
```

- `ascii(string)`

Description: Indicates the ASCII code of the first character in the string.

Return type: integer

For example:

```
SELECT ascii('xyz');
ascii
-----
120
(1 row)
```

- `replace(string varchar, search_string varchar, replacement_string varchar)`

Description: Replaces all **search-string** in the string with **replacement\_string**.

Return type: varchar

For example:

```
SELECT replace('jack and jue','j','bl');
replace
-----
black and blue
(1 row)
```

- `lpad(string varchar, length int[, repeat_string varchar])`

Description: Adds a series of **repeat\_string** (a space by default) on the left of the string to generate a new string with the total length of n.

If the length of the string is longer than the specified length, the function truncates the string and returns the substrings with the specified length.

Return type: varchar

For example:

```
SELECT lpad('PAGE 1',15,'*');
lpad
-----
*****PAGE 1
(1 row)
SELECT lpad('hello world',5,'abcd');
lpad
-----
hello
(1 row)
```

- `concat(str1,str2)`

Description: Connects str1 and str2 and returns the string.

Return type: varchar

For example:

```
SELECT concat('Hello', ' World!');
concat
-----
Hello World!
(1 row)
```

- `chr(integer)`

Description: Specifies the character of the ASCII code.

Return type: varchar

For example:

```
SELECT chr(65);
chr
-----
A
(1 row)
```

- `regexp_substr(source_char, pattern)`

Description: Extracts substrings from a regular expression.

Return type: varchar

For example:

```
SELECT regexp_substr('500 Hello World, Redwood Shores, CA', '[^,]+') "REGEXPR_SUBSTR";
REGEXPR_SUBSTR
-----
, Redwood Shores,
(1 row)
```

- `regexp_replace(string, pattern, replacement [,flags ])`

Description: Replaces substring matching POSIX regular expression. The source string is returned unchanged if there is no match to the pattern. If there is a match, the source string is returned with the replacement string substituted for the matching substring.

The replacement string can contain `\n`, where `n` is 1 through 9, to indicate that the source substring matching the `n`th parenthesized sub-expression of the pattern should be inserted, and it can contain `\&` to indicate that the substring matching the entire pattern should be inserted.

The optional **flags** argument contains zero or multiple single-letter flags that change function behavior. **i** indicates that the matching is not related to uppercase and lowercase. **g** indicates that each matching substring is replaced, instead of replacing only the first one.

Return type: varchar

For example:

```
SELECT regexp_replace('Thomas', '[mN]a.', 'M');
regexp_replace
-----
ThM
(1 row)
SELECT regexp_replace('foobarbaz','b(..)', E'X\1Y', 'g') AS RESULT;
result
-----
fooXarYXazY
(1 row)
```

- `concat_ws(sep text, str"any" [, str"any" [, ...] ])`

Description: The first parameter is used as the separator, which is associated with all following parameters.

Return type: text

For example:

```
SELECT concat_ws(',', 'ABCDE', 2, NULL, 22);
concat_ws
-----
ABCDE,2,22
(1 row)
```

- `convert(string bytea, src_encoding name, dest_encoding name)`

Description: Converts the bytea string to **dest\_encoding**. **src\_encoding** specifies the source code encoding. The string must be valid in this encoding.

Return type: bytea

For example:

```
SELECT convert('text_in_utf8', 'UTF8', 'GBK');
convert
-----
\x746578745f696e5f75746638
(1 row)
```



 NOTE

If the rule for converting between source to target encoding (for example, GBK and LATIN1) does not exist, the string is returned without conversion. See the **pg\_conversion** system catalog for details.

For example:

```
show server_encoding;
server_encoding
-----
LATIN1
(1 row)

SELECT convert_from('some text', 'GBK');
convert_from
-----
some text
(1 row)

db_latin1=# SELECT convert_to('some text', 'GBK');
convert_to
-----
\x736f6d652074657874
(1 row)

db_latin1=# SELECT convert('some text', 'GBK', 'LATIN1');
convert
-----
\x736f6d652074657874
(1 row)
```

- **convert\_from**(string bytea, src\_encoding name)

Description: Converts the long bytea using the coding mode of the database. **src\_encoding** specifies the source code encoding. The string must be valid in this encoding.

Return type: text

For example:

```
SELECT convert_from('text_in_utf8', 'UTF8');
convert_from
-----
text_in_utf8
(1 row)
```

- **convert\_to**(string text, dest\_encoding name)

Description: Converts string to **dest\_encoding**.

Return type: bytea

For example:

```
SELECT convert_to('some text', 'UTF8');
convert_to
-----
\x736f6d652074657874
(1 row)
```

- **string** [NOT] LIKE pattern [ESCAPE escape-character]

Description: Pattern matching function

If the pattern does not include a percentage sign (%) or an underscore (\_), this mode represents itself only. In this case, the behavior of LIKE is the same as the equal operator. The underscore (\_) in the pattern matches any single character while one percentage sign (%) matches no or multiple characters.

To match with underscores (\_) or percent signs (%), corresponding characters in pattern must lead escape characters. The default escape character is a

backward slash (\) and can be specified using the **ESCAPE** clause. To match with escape characters, enter two escape characters.

Return type: Boolean

For example:

```
SELECT 'AA_BBCC' LIKE '%A@_B%' ESCAPE '@' AS RESULT;
result
-----
t
(1 row)
SELECT 'AA_BBCC' LIKE '%A@_B%' AS RESULT;
result
-----
f
(1 row)
SELECT 'AA@_BBCC' LIKE '%A@_B%' AS RESULT;
result
-----
t
(1 row)
```

- **REGEXP\_LIKE**(source\_string, pattern [, match\_parameter])

Description: Indicates the mode matching function of a regular expression.

**source\_string** indicates the source string and **pattern** indicates the matching pattern of the regular expression. **match\_parameter** indicates the matching items and the values are as follows:

- "i": case-insensitive
- "c": case-sensitive
- "n": allowing the metacharacter "." in a regular expression to be matched with a linefeed.
- "m": allows **source\_string** to be regarded as multiple rows.

If **match\_parameter** is ignored, **case-sensitive** is enabled by default, "." is not matched with a linefeed, and **source\_string** is regarded as a single row.

Return type: Boolean

For example:

```
SELECT regexp_like('ABC', '[A-Z]');
regexp_like
-----
t
(1 row)
SELECT regexp_like('ABC', '[D-Z]');
regexp_like
-----
f
(1 row)
SELECT regexp_like('ABC', '[A-Z]', 'i');
regexp_like
-----
t
(1 row)
SELECT regexp_like('ABC', '[A-Z]');
regexp_like
-----
t
(1 row)
```

- **format**(formatstr text [, str"any" [, ...] ])

Description: Formats a string.

Return type: text

For example:

```
SELECT format('Hello %s, %1$s', 'World');
       format
-----
Hello World, World
(1 row)
```

- md5(string)

Description: Encrypts a string in MD5 mode and returns a value in hexadecimal form.

 **NOTE**

MD5 is insecure and is not recommended.

Return type: text

For example:

```
SELECT md5('ABC');
       md5
-----
902fbbd2b1df0c4f70b4a5d23525e932
(1 row)
```

- decode(string text, format text)

Description: Decodes binary data from textual representation.

Return type: bytea

For example:

```
SELECT decode('MTIzAAE=', 'base64');
       decode
-----
\x3132330001
(1 row)
```

- encode(data bytea, format text)

Description: Encodes binary data into a textual representation.

Return type: text

For example:

```
SELECT encode(E'123\000\001', 'base64');
       encode
-----
MTIzAAE=
(1 row)
```

 **NOTE**

- For a string containing newline characters, for example, a string consisting of a newline character and a space, the value of **length** and **lengthb** in GaussDB(DWS) is 2.
- In GaussDB(DWS), *n* of the CHAR(*n*) type indicates the number of characters. Therefore, for multiple-octet coded character sets, the length returned by the LENGTHB function may be longer than *n*.

## 6.4 Binary String Functions and Operators

### String operators

SQL defines some string functions that use keywords, rather than commas, to separate arguments.

- `octet_length(string)`

Description: Number of bytes in binary string

Return type: int

For example:

```
SELECT octet_length(E'jo\000se'::bytea) AS RESULT;
result
-----
      5
(1 row)
```

- `overlay(string placing string from int [for int])`

Description: Replaces substring.

Return type: bytea

For example:

```
SELECT overlay(E'Th\000omas'::bytea placing E'\002\003'::bytea from 2 for 3) AS RESULT;
result
-----
\x5402036d6173
(1 row)
```

- `position(substring in string)`

Description: Location of specified substring

Return type: int

For example:

```
SELECT position(E'\000om'::bytea in E'Th\000omas'::bytea) AS RESULT;
result
-----
      3
(1 row)
```

- `substring(string [from int] [for int])`

Description: Truncates substring.

Return type: bytea

For example:

```
SELECT substring(E'Th\000omas'::bytea from 2 for 3) AS RESULT;
result
-----
\x68006f
(1 row)
```

- `trim([both] bytes from string)`

Description: Removes the longest string containing only bytes from **bytes** from the start and end of **string**.

Return type: bytea

For example:

```
SELECT trim(E'\000'::bytea from E'\000Tom\000'::bytea) AS RESULT;
result
-----
\x546f6d
(1 row)
```

## Other Binary String Functions

GaussDB(DWS) also provides the common syntax used for invoking functions.

- btrim(string bytea, bytes bytea)**  
 Description: Removes the longest string containing only bytes from **bytes** from the start and end of **string**.  
 Return type: bytea  
 For example:

```
SELECT btrim(E'\000trim\000':bytea, E'\000':bytea) AS RESULT;
result
-----
\x7472696d
(1 row)
```
- get\_bit(string, offset)**  
 Description: Extracts bit from string.  
 Return type: int  
 For example:

```
SELECT get_bit(E'Th\000omas':bytea, 45) AS RESULT;
result
-----
1
(1 row)
```
- get\_byte(string, offset)**  
 Description: Extracts byte from string.  
 Return type: int  
 For example:

```
SELECT get_byte(E'Th\000omas':bytea, 4) AS RESULT;
result
-----
109
(1 row)
```
- set\_bit(string, offset, newvalue)**  
 Description: Sets bit in string.  
 Return type: bytea  
 For example:

```
SELECT set_bit(E'Th\000omas':bytea, 45, 0) AS RESULT;
result
-----
\x5468006f6d4173
(1 row)
```
- set\_byte(string, offset, newvalue)**  
 Description: Sets byte in string.  
 Return type: bytea  
 For example:

```
SELECT set_byte(E'Th\000omas':bytea, 4, 64) AS RESULT;
result
-----
\x5468006f406173
(1 row)
```

## 6.5 Bit String Functions and Operators

### Bit string operators

Aside from the usual comparison operators, the following operators can be used. Bit string operands of **&**, **|**, and **#** must be of equal length. When bit shifting, the original length of the string is preserved by zero padding (if necessary).

- **||**

Description: Connects bit strings.

For example:

```
SELECT B'10001' || B'011' AS RESULT;
result
-----
10001011
(1 row)
```

- **&**

Description: AND operation between bit strings

For example:

```
SELECT B'10001' & B'01101' AS RESULT;
result
-----
00001
(1 row)
```

- **|**

Description: OR operation between bit strings

For example:

```
SELECT B'10001' | B'01101' AS RESULT;
result
-----
11101
(1 row)
```

- **#**

Description: OR operation between bit strings if they are inconsistent. If the same positions in the two bit strings are both 1 or 0, the position returns **0**.

For example:

```
SELECT B'10001' # B'01101' AS RESULT;
result
-----
11100
(1 row)
```

- **~**

Description: NOT operation between bit strings

For example:

```
SELECT ~B'10001' AS RESULT;
result
-----
01110
(1 row)
```

- **<<**

Description: binary left shift

```
For example:
SELECT B'10001' << 3 AS RESULT;
result
-----
01000
(1 row)
```

- >>  
Description: binary right shift

```
For example:
SELECT B'10001' >> 2 AS RESULT;
result
-----
00100
(1 row)
```

The following SQL-standard functions work on bit strings as well as character strings: **length**, **bit\_length**, **octet\_length**, **position**, **substring**, and **overlay**.

The following functions work on bit strings as well as binary strings: **get\_bit** and **set\_bit**. When working with a bit string, these functions number the first (leftmost) bit of the string as bit 0.

In addition, it is possible to convert between integral values and type **bit**. For example:

```
SELECT 44::bit(10) AS RESULT;
result
-----
0000101100
(1 row)

SELECT 44::bit(3) AS RESULT;
result
-----
100
(1 row)

SELECT cast(-44 as bit(12)) AS RESULT;
result
-----
111111010100
(1 row)

SELECT '1110'::bit(4)::integer AS RESULT;
result
-----
14
(1 row)
```

#### NOTE

Casting to just "bit" means casting to bit(1), and so will deliver only the least significant bit of the integer.

## 6.6 Pattern Matching Operators

There are three separate approaches to pattern matching provided by the database: the traditional SQL LIKE operator, the more recent SIMILAR TO operator, and POSIX-style regular expressions. Besides these basic operators, functions can be used to extract or replace matching substrings and to split a string at matching locations.

- LIKE

Description: checks whether the string matches the mode string following LIKE. The LIKE expression returns true if the string matches the supplied pattern. (As expected, the NOT LIKE expression returns false if LIKE returns true, and vice versa.)

Matching rule:

- This operator can succeed only when its pattern matches the entire string. If you want to match a sequence in any position within the string, the pattern must begin and end with a percent sign.
- The underscore (\_) represents (matching) any single character. Percentage (%) indicates the wildcard character of any string.
- To match a literal underscore or percent sign without matching other characters, the respective character in pattern must be preceded by the escape character. The default escape character is the backslash but a different one can be selected by using the ESCAPE clause.
- To match the escape character itself, write two escape characters. For example: To write a **pattern** constant containing a backslash (\), you need to enter two backslashes in SQL statements.

 NOTE

When **standard\_conforming\_strings** is set to **off**, any backslashes you write in literal string constants will need to be doubled. Therefore, writing a pattern matching a single backslash is actually going to write four backslashes in the statement. You can avoid this by selecting a different escape character by using ESCAPE, so that the backslash is no longer a special character of LIKE. But the backslash is still the special character of the character text analyzer, so you still need two backslashes.) You can also select no escape character by writing **ESCAPE ''**. This effectively disables the escape mechanism, which makes it impossible to turn off the special meaning of underscore and percent signs in the pattern.

- The keyword ILIKE can be used instead of LIKE to make the match case-insensitive.
- Operator ~~ is equivalent to LIKE, and operator ~~\* corresponds to ILIKE.

For example:

```
SELECT 'abc' LIKE 'abc' AS RESULT;
result
-----
t
(1 row)
SELECT 'abc' LIKE 'a%' AS RESULT;
result
-----
t
(1 row)
SELECT 'abc' LIKE '_b_' AS RESULT;
result
-----
t
(1 row)
SELECT 'abc' LIKE 'c' AS RESULT;
result
-----
f
(1 row)
```

- SIMILAR TO



Description: The SIMILAR TO operator returns true or false depending on whether the pattern matches the given string. It is similar to LIKE, except that it interprets the pattern using the SQL standard's definition of a regular expression.

Matching rule:

- a. Like LIKE, this operator succeeds only when its pattern matches the entire string. If you want to match a sequence in any position within the string, the pattern must begin and end with a percent sign.
- b. The underscore (\_) represents (matching) any single character. Percentage (%) indicates the wildcard character of any string.
- c. SIMILAR TO supports these pattern-matching metacharacters borrowed from POSIX regular expressions:

Metacharacter	Description
	Specifies alternation (either of two alternatives).
*	Specifies repetition of the previous item zero or more times.
+	Specifies repetition of the previous item one or more times.
?	Specifies repetition of the previous item zero or one time.
{m}	Specifies repetition of the previous item exactly <i>m</i> times.
{m,}	Specifies repetition of the previous item <i>m</i> or more times.
{m,n}	Specifies repetition of the previous item at least <i>m</i> times and does not exceed <i>n</i> times.
()	Specifies that parentheses () can be used to group items into a single logical item.
[...]	Specifies a character class, just as in POSIX regular expressions.

- d. A preamble escape character disables the special meaning of any of these metacharacters. The rules for using escape characters are the same as those for LIKE.

Regular expressions:

The substring function with three parameters, **substring(string from pattern for escape)**, provides extraction of a substring that matches an SQL regular expression pattern.

For example:

```

SELECT 'abc' SIMILAR TO 'abc' AS RESULT;
result
-----
t
(1 row)
SELECT 'abc' SIMILAR TO 'a' AS RESULT;
result
-----
f
(1 row)
SELECT 'abc' SIMILAR TO '%(b|d)%' AS RESULT;
result
-----
t
(1 row)
SELECT 'abc' SIMILAR TO '(b|c)%' AS RESULT;
result
-----
f
(1 row)

```

- **POSIX regular expressions**

Description: A regular expression is a character sequence that is an abbreviated definition of a set of strings (a regular set). If a string is a member of a regular expression described by a regular expression, the string matches the regular expression. POSIX regular expressions provide a more powerful means for pattern matching than the LIKE and SIMILAR TO operators. [Table 1 Regular expression match operators](#) lists all available operators for pattern matching using POSIX regular expressions.

**Table 6-3** Regular expression match operators

Operator	Description	Example
~	Matches regular expression, which is case-sensitive.	'thomas' ~ '.*thomas.*'
~*	Matches regular expression, which is case-insensitive.	'thomas' ~* '.*Thomas.*'
!~	Does not match regular expression, which is case-sensitive.	'thomas' !~ '.*Thomas.*'
!~*	Does not match regular expression, which is case-insensitive.	'thomas' !~* '.*vadim.*'

Matching rule:

- Unlike LIKE patterns, a regular expression is allowed to match anywhere within a string, unless the regular expression is explicitly anchored to the beginning or end of the string.
- Besides the metacharacters mentioned above, POSIX regular expressions also support the following pattern matching metacharacters:

Metacharacter	Description
^	Specifies the match starting with a string.
\$	Specifies the match at the end of a string.
.	Matches any single character.

Regular expressions:

POSIX regular expressions support the following functions:

- The **substring(string from pattern)** function provides a method for extracting a substring that matches the POSIX regular expression pattern.
- The **regexp\_replace** function provides the function of replacing the substring matching the POSIX regular expression pattern with the new text.
- The **regexp\_matches** function returns a text array consisting of all captured substrings that match a POSIX regular expression pattern.
- The **regexp\_split\_to\_table** function splits a string using a POSIX regular expression pattern as a delimiter.
- The **regexp\_split\_to\_array** function behaves the same as **regexp\_split\_to\_table**, except that **regexp\_split\_to\_array** returns its result as an array of text.

 NOTE

The regular expression split functions ignore zero-length matches, which occur at the beginning or end of a string or after the previous match. This is contrary to the strict definition of regular expression matching. The latter is implemented by **regexp\_matches**, but the former is usually the most commonly used behavior in practice.

For example:

```
SELECT 'abc' ~ 'Abc' AS RESULT;
result
-----
f
(1 row)
SELECT 'abc' ~* 'Abc' AS RESULT;
result
-----
t
(1 row)
SELECT 'abc' !~ 'Abc' AS RESULT;
result
-----
t
(1 row)
SELECT 'abc' !~* 'Abc' AS RESULT;
result
-----
f
(1 row)
SELECT 'abc' ~ '^a' AS RESULT;
result
-----
t
(1 row)
```

```
SELECT 'abc' ~ '(b|d)' AS RESULT;
result
-----
t
(1 row)
SELECT 'abc' ~ '^ (b|c)' AS RESULT;
result
-----
f
(1 row)
```

Although most regular expression searches can be executed quickly, regular expressions can still be artificially made up of memory that takes a long time and any amount of memory. It is not recommended that you accept the regular expression search mode from the non-security mode source. If you must do this, you are advised to add the statement timeout limit. The search with the SIMILAR TO mode has the same security risks as the SIMILAR TO provides many capabilities that are the same as those of the POSIX- style regular expression. The LIKE search is much simpler than the other two options. Therefore, it is more secure to accept the non-secure mode source search.

## 6.7 Mathematical Functions and Operators

### Numeric operators

- +

Description: Addition

For example:

```
SELECT 2+3 AS RESULT;
result
-----
5
(1 row)
```

- -

Description: Subtraction

For example:

```
SELECT 2-3 AS RESULT;
result
-----
-1
(1 row)
```

- \*

Description: multiply

For example:

```
SELECT 2*3 AS RESULT;
result
-----
6
(1 row)
```

- /

Description: Division (The result is not rounded.)

For example:

```
SELECT 4/2 AS RESULT;
result
```

```
-----  
2  
(1 row)  
SELECT 4/3 AS RESULT;  
result  
-----  
1.3333333333333333  
(1 row)
```

- +/-  
Description: Positive/negative

For example:

```
SELECT -2 AS RESULT;  
result  
-----  
-2  
(1 row)
```

- %  
Description: Model (to obtain the remainder)

For example:

```
SELECT 5%4 AS RESULT;  
result  
-----  
1  
(1 row)
```

- @  
Description: Absolute value

For example:

```
SELECT @ -5.0 AS RESULT;  
result  
-----  
5.0  
(1 row)
```

- ^  
Description: Power (exponent calculation)

For example:

```
SELECT 2.0^3.0 AS RESULT;  
result  
-----  
8.0000000000000000  
(1 row)
```

- |/  
Description: Square root

For example:

```
SELECT |/ 25.0 AS RESULT;  
result  
-----  
5  
(1 row)
```

- ||/  
Description: Cubic root

For example:

```
SELECT ||/ 27.0 AS RESULT;  
result  
-----
```

- ```
      3
(1 row)
```

!

Description: Factorial

For example:

```
SELECT 5! AS RESULT;
result
-----
      120
(1 row)
```
- ```
!!
```

Description: Factorial (prefix operator)

For example:

```
SELECT !!5 AS RESULT;
result
-----
      120
(1 row)
```
- ```
&
```

Description: Binary AND

For example:

```
SELECT 91&15 AS RESULT;
result
-----
       11
(1 row)
```
- ```
|
```

Description: Binary OR

For example:

```
SELECT 32|3 AS RESULT;
result
-----
       35
(1 row)
```
- ```
#
```

Description: Binary XOR

For example:

```
SELECT 17#5 AS RESULT;
result
-----
       20
(1 row)
```
- ```
~
```

Description: Binary NOT

For example:

```
SELECT ~1 AS RESULT;
result
-----
      -2
(1 row)
```
- ```
<<
```

Description: Binary shift left

For example:

```
SELECT 1<<4 AS RESULT;  
result  
-----  
16  
(1 row)
```

- >>  
Description: Binary shift right

For example:

```
SELECT 8>>2 AS RESULT;  
result  
-----  
2  
(1 row)
```

## Numeric operation functions

- abs(x)  
Description: Absolute value  
Return type: same as the input

For example:

```
SELECT abs(-17.4);  
abs  
-----  
17.4  
(1 row)
```

- acos(x)  
Description: Arc cosine  
Return type: double precision

For example:

```
SELECT acos(-1);  
acos  
-----  
3.14159265358979  
(1 row)
```

- asin(x)  
Description: Arc sine  
Return type: double precision

For example:

```
SELECT asin(0.5);  
asin  
-----  
.523598775598299  
(1 row)
```

- atan(x)  
Description: Arc tangent  
Return type: double precision

For example:

```
SELECT atan(1);  
atan  
-----  
.785398163397448  
(1 row)
```

- atan2(y, x)

Description: Arc tangent of  $y/x$

Return type: double precision

For example:

```
SELECT atan2(2, 1);
      atan2
-----
1.10714871779409
(1 row)
```

- **bitand(integer, integer)**

Description: Performs AND (&) operation on two integers.

Return type: bigint

For example:

```
SELECT bitand(127, 63);
      bitand
-----
          63
(1 row)
```

- **cbrt(dp)**

Description: Cubic root

Return type: double precision

For example:

```
SELECT cbrt(27.0);
      cbrt
-----
          3
(1 row)
```

- **ceil(x)**

Description: Minimum integer greater than or equal to the parameter

Return type: integer

For example:

```
SELECT ceil(-42.8);
      ceil
-----
      -42
(1 row)
```

- **ceiling(dp or numeric)**

Description: Minimum integer (alias of ceil) greater than or equal to the parameter

Return type: same as the input

For example:

```
SELECT ceiling(-95.3);
      ceiling
-----
      -95
(1 row)
```

- **cos(x)**

Description: Cosine

Return type: double precision

For example:

```
SELECT cos(-3.1415927);
      cos
```



```
-----  
-.9999999999999999  
(1 row)
```

- **cot(x)**

Description: Cotangent

Return type: double precision

For example:

```
SELECT cot(1);  
cot  
-----  
.642092615934331  
(1 row)
```

- **degrees(dp)**

Description: Converts radians to angles.

Return type: double precision

For example:

```
SELECT degrees(0.5);  
degrees  
-----  
28.6478897565412  
(1 row)
```

- **div(y numeric, x numeric)**

Description: Integer part of  $y/x$

Return type: numeric

For example:

```
SELECT div(9,4);  
div  
-----  
2  
(1 row)
```

- **exp(x)**

Description: Natural exponent

Return type: same as the input

For example:

```
SELECT exp(1.0);  
exp  
-----  
2.7182818284590452  
(1 row)
```

- **floor(x)**

Description: Not larger than the maximum integer of the parameter

Return type: same as the input

For example:

```
SELECT floor(-42.8);  
floor  
-----  
-43  
(1 row)
```

- **radians(dp)**

Description: Converts angles to radians.

Return type: double precision

For example:

```
SELECT radians(45.0);
      radians
-----
.785398163397448
(1 row)
```

- random()

Description: Random number between 0.0 and 1.0

Return type: double precision

For example:

```
SELECT random();
      random
-----
.824823560658842
(1 row)
```

- ln(x)

Description: Natural logarithm

Return type: same as the input

For example:

```
SELECT ln(2.0);
      ln
-----
.6931471805599453
(1 row)
```

- log(x)

Description: Logarithm with 10 as the base

Return type: same as the input

For example:

```
SELECT log(100.0);
      log
-----
2.0000000000000000
(1 row)
```

- log(b numeric, x numeric)

Description: Logarithm with b as the base

Return type: numeric

For example:

```
SELECT log(2.0, 64.0);
      log
-----
6.0000000000000000
(1 row)
```

- mod(x,y)

Description:

Remainder of x/y (model)

If x equals to 0, y is returned.

Return type: same as the parameter type

For example:

```
SELECT mod(9,4);
      mod
-----
```

```
1
(1 row)
SELECT mod(9,0);
mod
-----
9
(1 row)
```

- pi()

Description:  $\pi$  constant value

Return type: double precision

For example:

```
SELECT pi();
pi
-----
3.14159265358979
(1 row)
```

- power(a double precision, b double precision)

Description: b power of a

Return type: double precision

For example:

```
SELECT power(9.0, 3.0);
power
-----
729.0000000000000000
(1 row)
```

- round(x)

Description: Integer closest to the input parameter

Return type: same as the input

For example:

```
SELECT round(42.4);
round
-----
42
(1 row)

SELECT round(42.6);
round
-----
43
(1 row)
```

- round(v numeric, s int)

Description: s digits are kept after the decimal point.

Return type: numeric

For example:

```
SELECT round(42.4382, 2);
round
-----
42.44
(1 row)
```

- setseed(dp)

Description: Sets seed for the following random() invoking (between -1.0 and 1.0, inclusive).

Return type: void

For example:

```
SELECT setseed(0.54823);
setseed
-----
(1 row)
```

- **sign(x)**

Description: returns symbols of this parameter.

The return value type:-1 indicates negative. 0 indicates 0, and 1 indicates a positive number.

For example:

```
SELECT sign(-8.4);
sign
-----
-1
(1 row)
```

- **sin(x)**

Description: Sine

Return type: double precision

For example:

```
SELECT sin(1.57079);
sin
-----
.999999999979986
(1 row)
```

- **sqrt(x)**

Description: Square root

Return type: same as the input

For example:

```
SELECT sqrt(2.0);
sqrt
-----
1.414213562373095
(1 row)
```

- **tan(x)**

Description: Tangent

Return type: double precision

For example:

```
SELECT tan(20);
tan
-----
2.23716094422474
(1 row)
```

- **trunc(x)**

Description: truncates (the integral part).

Return type: same as the input

For example:

```
SELECT trunc(42.8);
trunc
-----
42
(1 row)
```

- **trunc(v numeric, s int)**

Description: Truncates a number with **s** digits after the decimal point.

Return type: numeric

For example:

```
SELECT trunc(42.4382, 2);
trunc
-----
42.43
(1 row)
```

- `width_bucket(op numeric, b1 numeric, b2 numeric, count int)`

Description: Returns a bucket to which the operand will be assigned in an equidepth histogram with **count** buckets, ranging from **b1** to **b2**.

Return type: int

For example:

```
SELECT width_bucket(5.35, 0.024, 10.06, 5);
width_bucket
-----
3
(1 row)
```

- `width_bucket(op dp, b1 dp, b2 dp, count int)`

Description: Returns a bucket to which the operand will be assigned in an equidepth histogram with **count** buckets, ranging from **b1** to **b2**.

Return type: int

For example:

```
SELECT width_bucket(5.35, 0.024, 10.06, 5);
width_bucket
-----
3
(1 row)
```

## 6.8 Date and Time Processing Functions and Operators

### Date and Time Operators



When the user uses date/time operators, explicit type prefixes are modified for corresponding operands to ensure that the operands parsed by the database are consistent with what the user expects, and no unexpected results occur.

For example, abnormal mistakes will occur in the following example without an explicit data type.

```
SELECT date '2001-10-01' - '7' AS RESULT;
```

---

**Table 6-4** Time and date operators

| Operators | Examples                                                                                                                       |
|-----------|--------------------------------------------------------------------------------------------------------------------------------|
| +         | <pre>SELECT date '2001-09-28' + integer '7' AS RESULT;       result ----- 2001-10-05 00:00:00 (1 row)</pre>                    |
|           | <pre>SELECT date '2001-09-28' + interval '1 hour' AS RESULT;       result ----- 2001-09-28 01:00:00 (1 row)</pre>              |
|           | <pre>SELECT date '2001-09-28' + time '03:00' AS RESULT;       result ----- 2001-09-28 03:00:00 (1 row)</pre>                   |
|           | <pre>SELECT interval '1 day' + interval '1 hour' AS RESULT;       result ----- 1 day 01:00:00 (1 row)</pre>                    |
|           | <pre>SELECT timestamp '2001-09-28 01:00' + interval '23 hours' AS RESULT;       result ----- 2001-09-29 00:00:00 (1 row)</pre> |
|           | <pre>SELECT time '01:00' + interval '3 hours' AS RESULT;       result ----- 04:00:00 (1 row)</pre>                             |
| -         | <pre>SELECT date '2001-10-01' - date '2001-09-28' AS RESULT;       result ----- 3 days (1 row)</pre>                           |
|           | <pre>SELECT date '2001-10-01' - integer '7' AS RESULT;       result ----- 2001-09-24 00:00:00 (1 row)</pre>                    |
|           | <pre>SELECT date '2001-09-28' - interval '1 hour' AS RESULT;       result ----- 2001-09-27 23:00:00 (1 row)</pre>              |
|           | <pre>SELECT time '05:00' - time '03:00' AS RESULT;       result ----- 02:00:00 (1 row)</pre>                                   |

| Operators | Examples                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|-----------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|           | <pre>SELECT time '05:00' - interval '2 hours' AS RESULT; result ----- 03:00:00 (1 row)</pre> <pre>SELECT timestamp '2001-09-28 23:00' - interval '23 hours' AS RESULT; result ----- 2001-09-28 00:00:00 (1 row)</pre> <pre>SELECT interval '1 day' - interval '1 hour' AS RESULT; result ----- 23:00:00 (1 row)</pre> <pre>SELECT timestamp '2001-09-29 03:00' - timestamp '2001-09-27 12:00' AS RESULT; result ----- 1 day 15:00:00 (1 row)</pre> |
| *         | <pre>SELECT 900 * interval '1 second' AS RESULT; result ----- 00:15:00 (1 row)</pre> <pre>SELECT 21 * interval '1 day' AS RESULT; result ----- 21 days (1 row)</pre> <pre>SELECT double precision '3.5' * interval '1 hour' AS RESULT; result ----- 03:30:00 (1 row)</pre>                                                                                                                                                                         |
| /         | <pre>SELECT interval '1 hour' / double precision '1.5' AS RESULT; result ----- 00:40:00 (1 row)</pre>                                                                                                                                                                                                                                                                                                                                              |

## Time/Date functions

- age(timestamp, timestamp)**  
 Description: Subtracts arguments, producing a result in YYYY-MM-DD format. If the result is negative, the returned result is also negative.  
 Return type: interval  
 For example:  

```
SELECT age(timestamp '2001-04-10', timestamp '1957-06-13');
age
-----
```

- 43 years 9 mons 27 days  
(1 row)
- **age(timestamp)**  
Description: Subtracts from **current\_date**  
Return type: interval  
For example:  

```
SELECT age(timestamp '1957-06-13');
      age
-----
60 years 2 mons 18 days
(1 row)
```
- **clock\_timestamp()**  
Description: Specifies the current timestamp of the real-time clock.  
Return type: timestamp with time zone  
For example:  

```
SELECT clock_timestamp();
      clock_timestamp
-----
2017-09-01 16:57:36.636205+08
(1 row)
```
- **current\_date**  
Description: Current date  
Return type: date  
For example:  

```
SELECT current_date;
      date
-----
2017-09-01
(1 row)
```
- **current\_time**  
Description: Current time  
Return type: time with time zone  
For example:  

```
SELECT current_time;
      timetz
-----
16:58:07.086215+08
(1 row)
```
- **current\_timestamp**  
Description: Specifies the current date and time.  
Return type: timestamp with time zone  
For example:  

```
SELECT current_timestamp;
      pg_systimestamp
-----
2017-09-01 16:58:19.22173+08
(1 row)
```
- **date\_part(text, timestamp)**  
Description:  
Description: Obtains the hour.  
Equivalent to **extract(field from timestamp)**.



Return type: double precision

For example:

```
SELECT date_part('hour', timestamp '2001-02-16 20:38:40');
date_part
-----
      20
(1 row)
```

- `date_part(text, interval)`

Description:

Obtains the month. If the value is greater than 12, obtain the remainder after it is divided by 12.

Equivalent to **`extract(field from timestamp)`**.

Return type: double precision

For example:

```
SELECT date_part('month', interval '2 years 3 months');
date_part
-----
       3
(1 row)
```

- `date_trunc(text, timestamp)`

Description: Truncates to the precision specified by **text**.

Return type: timestamp

For example:

```
SELECT date_trunc('hour', timestamp '2001-02-16 20:38:40');
date_trunc
-----
2001-02-16 20:00:00
(1 row)
```

- `trunc(timestamp)`

Description: By default, the data is intercepted by day.

For example:

```
SELECT trunc(timestamp '2001-02-16
20:38:40');
trunc
-----
2001-02-16 00:00:00
(1 row)
```

- `extract(field from timestamp)`

Description: Obtains the hour.

Return type: double precision

For example:

```
SELECT extract(hour from timestamp '2001-02-16 20:38:40');
date_part
-----
      20
(1 row)
```

- `extract(field from interval)`

Description: Obtains the month. If the value is greater than 12, obtain the remainder after it is divided by 12.

Return type: double precision

For example:

```
SELECT extract(month from interval '2 years 3 months');
date_part
-----
3
(1 row)
```

- **isfinite(date)**

Description: Tests for valid date.

Return type: Boolean

For example:

```
SELECT isfinite(date '2001-02-16');
isfinite
-----
t
(1 row)
```

- **isfinite(timestamp)**

Description: Tests for valid timestamp.

Return type: Boolean

For example:

```
SELECT isfinite(timestamp '2001-02-16 21:28:30');
isfinite
-----
t
(1 row)
```

- **isfinite(interval)**

Description: Tests for valid interval.

Return type: Boolean

For example:

```
SELECT isfinite(interval '4 hours');
isfinite
-----
t
(1 row)
```

- **justify\_days(interval)**

Description: Adjusts interval to 30-day time periods are represented as months

Return type: interval

For example:

```
SELECT justify_days(interval '35 days');
justify_days
-----
1 mon 5 days
(1 row)
```

- **justify\_hours(interval)**

Description: Adjusts interval to 24-hour time periods are represented as days

Return type: interval

For example:

```
SELECT JUSTIFY_HOURS(INTERVAL '27 HOURS');
justify_hours
-----
1 day 03:00:00
(1 row)
```

- **justify\_interval(interval)**

Description: Adjusts **interval** using **justify\_days** and **justify\_hours**.

Return type: interval

For example:

```
SELECT JUSTIFY_INTERVAL(INTERVAL '1 MON -1 HOUR');
justify_interval
-----
29 days 23:00:00
(1 row)
```

- localtime

Description: Current time

Return type: time

For example:

```
SELECT localtime AS RESULT;
result
-----
16:05:55.664681
(1 row)
```

- localtimestamp

Description: Specifies the current date and time.

Return type: timestamp

For example:

```
SELECT localtimestamp;
timestamp
-----
2017-09-01 17:03:30.781902
(1 row)
```

- now()

Description: Specifies the current date and time.

Return type: timestamp with time zone

For example:

```
SELECT now();
now
-----
2017-09-01 17:03:42.549426+08
(1 row)
```

- numtodsinterval(num, interval\_unit)

Description: Converts a number to the interval type. **num** is a numeric-typed number. **interval\_unit** is a string in the following format: 'DAY' | 'HOUR' | 'MINUTE' | 'SECOND'

You can set the **IntervalStyle** parameter to **oracle** to be compatible with the interval output format of the function in the Oracle database.

For example:

```
SELECT numtodsinterval(100, 'HOUR');
numtodsinterval
-----
100:00:00
(1 row)

SET intervalstyle = oracle;
SET
SELECT numtodsinterval(100, 'HOUR');
numtodsinterval
-----
```

```
+000000004 04:00:00.000000000
(1 row)
```

- `pg_sleep(seconds)`

Description: Specifies the delay time of the server thread in unit of second.

Return type: void

For example:

```
SELECT pg_sleep(10);
pg_sleep
-----
(1 row)
```

- `statement_timestamp()`

Description: Specifies the current date and time.

Return type: timestamp with time zone

For example:

```
SELECT statement_timestamp();
statement_timestamp
-----
2017-09-01 17:04:39.119267+08
(1 row)
```

- `sysdate`

Description: Specifies the current date and time.

Return type: timestamp

For example:

```
SELECT sysdate;
sysdate
-----
2017-09-01 17:04:49
(1 row)
```

- `timeofday()`

Description: Current date and time (like `clock_timestamp`, but returned as a **text** string)

Return type: text

For example:

```
SELECT timeofday();
timeofday
-----
Fri Sep 01 17:05:01.167506 2017 CST
(1 row)
```

- `transaction_timestamp()`

Description: Current date and time (equivalent to `current_timestamp`)

Return type: timestamp with time zone

For example:

```
SELECT transaction_timestamp();
transaction_timestamp
-----
2017-09-01 17:05:13.534454+08
(1 row)
```

- `add_months(d,n)`

Description: Calculates the time point day and time of nth months.

Return type: timestamp

For example:

```
SELECT add_months(to_date('2017-5-29', 'yyyy-mm-dd'), 11) FROM dual;
add_months
-----
2018-04-29 00:00:00
(1 row)
```

- last\_day(d)

Description: Calculates the time of the last day in the month.

Return type: timestamp

For example:

```
select last_day(to_date('2017-01-01', 'YYYY-MM-DD')) AS cal_result;
cal_result
-----
2017-01-31 00:00:00
(1 row)
```

- next\_day(x,y)

Description: Calculates the time of the next week y started from x

Return type: timestamp

For example:

```
postgres=# select next_day(timestamp '2017-05-25 00:00:00','Sunday')AS cal_result;
cal_result
-----
2017-05-28 00:00:00
(1 row)
```

## EXTRACT

### EXTRACT(*field* FROM *source*)

The **extract** function retrieves subcolumns such as year or hour from date/time values. **source** must be a value expression of type **timestamp**, **time**, or **interval**. (Expressions of type **date** are cast to **timestamp** and can therefore be used as well.) **field** is an identifier or string that selects what column to extract from the source value. The **extract** function returns values of type **double precision**. The following are valid field names:

- century

Century

The first century starts at 0001-01-01 00:00:00 AD. This definition applies to all Gregorian calendar countries. There is no century number 0. You go from **-1** century to **1** century.

For example:

```
SELECT EXTRACT(CENTURY FROM TIMESTAMP '2000-12-16 12:21:13');
date_part
-----
20
(1 row)
```

- day

– For **timestamp** values, the day (of the month) column (1–31)

```
SELECT EXTRACT(DAY FROM TIMESTAMP '2001-02-16 20:38:40');
date_part
-----
16
(1 row)
```

- For **interval** values, the number of days  

```
SELECT EXTRACT(DAY FROM INTERVAL '40 days 1 minute');
date_part
-----
      40
(1 row)
```

- decade

Year column divided by 10

```
SELECT EXTRACT(DECADE FROM TIMESTAMP '2001-02-16 20:38:40');
date_part
-----
      200
(1 row)
```

- dow

Day of the week as Sunday(0) to Saturday (6)

```
SELECT EXTRACT(DOW FROM TIMESTAMP '2001-02-16 20:38:40');
date_part
-----
        5
(1 row)
```

- doy

Day of the year (1–365 or 366)

```
SELECT EXTRACT(DOY FROM TIMESTAMP '2001-02-16 20:38:40');
date_part
-----
       47
(1 row)
```

- epoch

- For **timestamp with time zone** values, the number of seconds since 1970-01-01 00:00:00 UTC (can be negative);

for **date** and **timestamp** values, the number of seconds since 1970-01-01 00:00:00 local time;

for **interval** values, the total number of seconds in the interval.

```
SELECT EXTRACT(EPOCH FROM TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40.12-08');
date_part
-----
982384720.12
(1 row)
```

```
SELECT EXTRACT(EPOCH FROM INTERVAL '5 days 3 hours');
date_part
-----
      442800
(1 row)
```

- Way to convert an epoch value back to a timestamp

```
SELECT TIMESTAMP WITH TIME ZONE 'epoch' + 982384720.12 * INTERVAL '1 second' AS
RESULT;
result
-----
2001-02-17 12:38:40.12+08
(1 row)
```

- hour

Hour column (0–23)

```
SELECT EXTRACT(HOUR FROM TIMESTAMP '2001-02-16 20:38:40');
date_part
-----
       20
(1 row)
```

- **isodow**  
Day of the week (1–7)  
Monday is 1 and Sunday is 7.

 **NOTE**

This is identical to **dow** except for Sunday.

```
SELECT EXTRACT(ISODOW FROM TIMESTAMP '2001-02-18 20:38:40');
date_part
-----
       7
(1 row)
```

- **isoyear**  
The ISO 8601 year that the date falls in (not applicable to intervals).  
Each ISO year begins with the Monday of the week containing the 4th of January, so in early January or late December the ISO year may be different from the Gregorian year. See the **week** column for more information.

```
SELECT EXTRACT(ISOYEAR FROM DATE '2006-01-01');
date_part
-----
      2005
(1 row)
SELECT EXTRACT(ISOYEAR FROM DATE '2006-01-02');
date_part
-----
      2006
(1 row)
```

- **microseconds**  
The seconds column, including fractional parts, multiplied by 1,000,000

```
SELECT EXTRACT(MICROSECONDS FROM TIME '17:12:28.5');
date_part
-----
28500000
(1 row)
```

- **millennium**  
Millennium  
Years in the 1900s are in the second millennium. The third millennium started from January 1, 2001.

```
SELECT EXTRACT(MILLENNIUM FROM TIMESTAMP '2001-02-16 20:38:40');
date_part
-----
       3
(1 row)
```

- **milliseconds**  
The seconds column, including fractional parts, multiplied by 1000. Note that this includes full seconds.

```
SELECT EXTRACT(MILLISECONDS FROM TIME '17:12:28.5');
date_part
-----
      28500
(1 row)
```

- **minute**  
Minutes column (0–59)

```
SELECT EXTRACT(MINUTE FROM TIMESTAMP '2001-02-16 20:38:40');
date_part
```

```
-----
      38
(1 row)
```

- month

For **timestamp** values, the number of the month within the year (1–12);

```
SELECT EXTRACT(MONTH FROM TIMESTAMP '2001-02-16 20:38:40');
date_part
```

```
-----
      2
(1 row)
```

For **interval** values, the number of months, modulo 12 (0–11)

```
SELECT EXTRACT(MONTH FROM INTERVAL '2 years 13 months');
date_part
```

```
-----
      1
(1 row)
```

- quarter

Quarter of the year (1–4) that the date is in

```
SELECT EXTRACT(QUARTER FROM TIMESTAMP '2001-02-16 20:38:40');
date_part
```

```
-----
      1
(1 row)
```

- second

Seconds column, including fractional parts (0–59)

```
SELECT EXTRACT(SECOND FROM TIME '17:12:28.5');
date_part
```

```
-----
    28.5
(1 row)
```

- timezone

The time zone offset from UTC, measured in seconds. Positive values correspond to time zones east of UTC, negative values to zones west of UTC.

- timezone\_hour

The hour component of the time zone offset

- timezone\_minute

The minute component of the time zone offset

- week

The number of the week of the year that the day is in. By definition (ISO 8601), the first week of a year contains January 4 of that year. (The ISO-8601 week starts on Monday.) In other words, the first Thursday of a year is in week 1 of that year.

Because of this, it is possible for early January dates to be part of the 52nd or 53rd week of the previous year, and late December dates to be part of the 1st week of the next year. For example, **2005-01-01** is part of the 53rd week of year 2004, **2006-01-01** is part of the 52nd week of year 2005, and **2012-12-31** is part of the 1st week of year 2013. You are advised to use the columns **isoyear** and **week** together to ensure consistency.

```
SELECT EXTRACT(WEEK FROM TIMESTAMP '2001-02-16 20:38:40');
date_part
```

```
-----
      7
(1 row)
```



- year

Year column

```
SELECT EXTRACT(YEAR FROM TIMESTAMP '2001-02-16 20:38:40');
date_part
-----
2001
(1 row)
```

## date\_part

The **date\_part** function is modeled on the traditional Ingres equivalent to the SQL-standard function **extract**:

**date\_part**('field', source)

Note that here the **field** parameter needs to be a string value, not a name. The valid field names for **date\_part** are the same as for **extract**. For details, see [EXTRACT](#).

For example:

```
SELECT date_part('day', TIMESTAMP '2001-02-16 20:38:40');
date_part
-----
16
(1 row)
SELECT date_part('hour', INTERVAL '4 hours 3 minutes');
date_part
-----
4
(1 row)
```

**Table 6-5** specifies the schema for formatting date and time values.

**Table 6-5** Date/time formats

| Category              | Format     | Description                                |
|-----------------------|------------|--------------------------------------------|
| Hours                 | HH         | Number of hours in one day (01-12)         |
|                       | HH12       | Number of hours in one day (01-12)         |
|                       | HH24       | Number of hours in one day (00-23)         |
| Minute                | MI         | Minute (00-59)                             |
| Seconds               | SS         | Second (00-59)                             |
|                       | FF         | Microsecond (000000-999999)                |
|                       | SSSSS      | Second after midnight (0-86399)            |
| Morning and afternoon | AM or A.M. | Morning identifier                         |
|                       | PM or P.M. | Afternoon identifier                       |
| Year                  | Y,YYY      | Year with comma (with four digits or more) |
|                       | SYYYY      | Year with four digits BC                   |
|                       | YYYY       | Year (with four digits or more)            |

| Category | Format                                                                               | Description                                                                                                                                                                                 |
|----------|--------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|          | YYY                                                                                  | Last three digits of a year                                                                                                                                                                 |
|          | YY                                                                                   | Last two digits of a year                                                                                                                                                                   |
|          | Y                                                                                    | Last one digit of a year                                                                                                                                                                    |
|          | IYYY                                                                                 | ISO year (with four digits or more)                                                                                                                                                         |
|          | IYY                                                                                  | Last three digits of an ISO year                                                                                                                                                            |
|          | IY                                                                                   | Last two digits of an ISO year                                                                                                                                                              |
|          | I                                                                                    | Last one digit of a year                                                                                                                                                                    |
|          | RR                                                                                   | Last two digits of a year (A year of the 20th century can be stored in the 21st century.)                                                                                                   |
|          | RRRR                                                                                 | Capable of receiving a year with four digits or two digits. If there are 2 digits, the value is the same as the returned value of RR. If there are 4 digits, the value is the same as YYYY. |
|          | <ul style="list-style-type: none"> <li>• BC or B.C.</li> <li>• AD or A.D.</li> </ul> | Era indicator Before Christ (BC) and After Christ (AD)                                                                                                                                      |
| Month    | MONTH                                                                                | Full spelling of a month in uppercase (9 characters are filled in if the value is empty.)                                                                                                   |
|          | MON                                                                                  | Month in abbreviated format in uppercase (with three characters)                                                                                                                            |
|          | MM                                                                                   | Month (01-12)                                                                                                                                                                               |
|          | RM                                                                                   | Month in Roman numerals (I-XII; I=JAN) and uppercase                                                                                                                                        |
| Day      | DAY                                                                                  | Full spelling of a date in uppercase (9 characters are filled in if the value is empty.)                                                                                                    |
|          | DY                                                                                   | Day in abbreviated format in uppercase (with three characters)                                                                                                                              |
|          | DDD                                                                                  | Day in a year (001-366)                                                                                                                                                                     |
|          | DD                                                                                   | Day in a month (01-31)                                                                                                                                                                      |
|          | D                                                                                    | Day in a week (1-7.)                                                                                                                                                                        |
| Week     | W                                                                                    | Week in a month (1-5) (The first week starts from the first day of the month.)                                                                                                              |
|          | WW                                                                                   | Week in a year (1-53) (The first week starts from the first day of the year.)                                                                                                               |
|          | IW                                                                                   | Week in an ISO year (The first Thursday is in the first week.)                                                                                                                              |

| Category    | Format | Description                                                          |
|-------------|--------|----------------------------------------------------------------------|
| Century     | CC     | Century (with two digits) (The 21st century starts from 2001-01-01.) |
| Julian date | J      | Julian date (starting from January 1 of 4712 BC)                     |
| Quarter     | Q      | Quarter                                                              |

 **NOTE**

In the table, the rules for RR to calculate years are as follows:

- If the range of the input two-digit year is between 00 and 49:
  - If the last two digits of the current year are between 00 and 49, the first two digits of the returned year are the same as the first two digits of the current year.
  - If the last two digits of the current year are between 50 and 99, the first two digits of the returned year equal to the first two digits of the current year plus 1.
- If the range of the input two-digit year is between 50 and 99:
  - If the last two digits of the current year are between 00 and 49, the first two digits of the returned year equal to the first two digits of the current year minus 1.
  - If the last two digits of the current year are between 50 and 99, the first two digits of the returned year are the same as the first two digits of the current year.

## 6.9 Type Conversion Functions

### Type Conversion Functions

- `cast(x as y)`  
Description: Converts x into the type specified by y.  
For example:  

```
SELECT cast('22-oct-1997' as timestamp);
timestamp
-----
1997-10-22 00:00:00
(1 row)
```
- `hextoraw(string)`  
Description: Converts a string in hexadecimal format into binary format.  
Return type: raw  
For example:  

```
SELECT hextoraw('7D');
hextoraw
-----
7D
(1 row)
```
- `numtoday(numeric)`  
Description: Converts values of the number type into the timestamp of the specified type.  
Return type: timestamp

For example:

```
SELECT numtoday(2);
 numtoday
-----
 2 days
(1 row)
```

- `pg_systimestamp()`

Description: Obtains the system timestamp.

Return type: timestamp with time zone

For example:

```
SELECT pg_systimestamp();
 pg_systimestamp
-----
2015-10-14 11:21:28.317367+08
(1 row)
```

- `rawtohex(string)`

Description: Converts a string in binary format into hexadecimal format.

The result is the ACSII code of the input characters in hexadecimal format.

Return type: varchar

For example:

```
SELECT rawtohex('1234567');
 rawtohex
-----
31323334353637
(1 row)
```

- `to_char (datetime/interval [, fmt])`

Description: Converts a DATETIME or INTERVAL value of the DATE/TIMESTAMP/TIMESTAMP WITH TIME ZONE/TIMESTAMP WITH LOCAL TIME ZONE type into the VARCHAR type according to the format specified by **fmt**.

- The optional parameters **fmt** include the following types: date, time, week, quarter, and century. Each type has a unique template. The templates can be combined together. Common templates include: HH, MM, SS, YYYY, MM, and DD.
- A template may have a modification word. FM is a common modification word and is used to suppress the preceding zero or the following blank spaces.

Return type: varchar

For example:

```
SELECT to_char(current_timestamp,'HH12:MI:SS');
 to_char
-----
10:19:26
(1 row)
SELECT to_char(current_timestamp,'FMHH12:FMMI:FMSS');
 to_char
-----
10:19:46
(1 row)
```

- `to_char(double precision, text)`

Description: Converts the values of the double-precision type into the strings in the specified format.

Return type: text

For example:

```
SELECT to_char(125.8::real, '999D99');
to_char
-----
125.80
(1 row)
```

- `to_char (integer/number[, fmt])`

Descriptions: Converts an integer or a value in floating point format into a string in specified format.

- Optional parameters `fmt` include the following types: decimal characters, grouping characters, positive/negative sign and currency sign. Each type has a unique template. The templates can be combined together. Common templates include: 9, 0, millesimal sign (.), and decimal point (.).
- A template can have a modification word, similar to FM. However, FM does not suppress 0 which is output according to the template.
- Use the template X or x to convert an integer value into a string in hexadecimal format.

Return type: `varchar`

For example:

```
SELECT to_char(1485,'9,999');
to_char
-----
1,485
(1 row)
SELECT to_char( 1148.5,'9,999.999');
to_char
-----
1,148.500
(1 row)
SELECT to_char(148.5,'990999.909');
to_char
-----
0148.500
(1 row)
SELECT to_char(123,'XXX');
to_char
-----
7B
(1 row)
```

- `to_char(interval, text)`

Description: Converts the values of the time interval type into the strings in the specified format.

Return type: `text`

For example:

```
SELECT to_char(interval '15h 2m 12s', 'HH24:MI:SS');
to_char
-----
15:02:12
(1 row)
```

- `to_char(int, text)`

Description: Converts the values of the integer type into the strings in the specified format.

Return type: `text`

For example:

```
SELECT to_char(125, '999');
to_char
-----
125
(1 row)
```

- `to_char(numeric, text)`

Description: Converts the values of the numeric type into the strings in the specified format.

Return type: text

For example:

```
SELECT to_char(-125.8, '999D99S');
to_char
-----
125.80-
(1 row)
```

- `to_char (string)`

Description: Converts the CHAR/VARCHAR/VARCHAR2/CLOB type into the VARCHAR type.

If this function is used to convert data of the CLOB type, and the value to be converted exceeds the value range of the target type, an error is returned.

Return type: varchar

For example:

```
SELECT to_char('01110');
to_char
-----
01110
(1 row)
```

- `to_char(timestamp, text)`

Description: Converts the values of the timestamp type into the strings in the specified format.

Return type: text

For example:

```
SELECT to_char(current_timestamp, 'HH12:MI:SS');
to_char
-----
10:55:59
(1 row)
```

- `to_clob(char/nchar/varchar/nvarchar/varchar2/nvarchar2/text/raw)`

Description: Convert the RAW type or text character set type CHAR/NCHAR/VARCHAR/VARCHAR2/NVARCHAR2/TEXT into the CLOB type.

Return type: clob

For example:

```
SELECT to_clob('ABCDEF'::RAW(10));
to_clob
-----
ABCDEF
(1 row)
SELECT to_clob('hello111'::CHAR(15));
to_clob
-----
hello111
(1 row)
SELECT to_clob('gauss123'::NCHAR(10));
to_clob
```

```

-----
gauss123
(1 row)
SELECT to_clob('gauss234'::VARCHAR(10));
to_clob
-----
gauss234
(1 row)
SELECT to_clob('gauss345'::VARCHAR2(10));
to_clob
-----
gauss345
(1 row)
SELECT to_clob('gauss456'::NVARCHAR2(10));
to_clob
-----
gauss456
(1 row)
SELECT to_clob('World222!'::TEXT);
to_clob
-----
World222!
(1 row)

```

- **to\_date(text)**

Description: Converts values of the text type into the timestamp in the specified format.

Return type: timestamp

For example:

```

SELECT to_date('2015-08-14');
to_date
-----
2015-08-14 00:00:00
(1 row)

```

- **to\_date(text, text)**

Description: Converts the values of the string type into the dates in the specified format.

Return type: timestamp

For example:

```

SELECT to_date('05 Dec 2000', 'DD Mon YYYY');
to_date
-----
2000-12-05 00:00:00
(1 row)

```

- **to\_date(string, fmt)**

Description:

Converts a string into a value of the DATE type according to the format specified by fmt.

This function cannot support the CLOB type directly. However, a parameter of the CLOB type can be converted using implicit conversion.

Return type: date

For example:

```

SELECT TO_DATE('05 Dec 2010','DD Mon YYYY');
to_date
-----
2010-12-05 00:00:00
(1 row)

```

- `to_number ( expr [, fmt])`

Description: Converts **expr** into a value of the NUMBER type according to the specified format.

For details about the type conversion formats, see [Table 6-6](#).

If a hexadecimal string is converted into a decimal number, the hexadecimal string can include a maximum of 16 bytes if it is to be converted into a sign-free number.

During the conversion from a hexadecimal string to a decimal digit, the format string cannot have a character other than x or X. Otherwise, an error is reported.

Return type: number

For example:

```
SELECT to_number('12,454.8-', '99G999D9S');
to_number
-----
-12454.8
(1 row)
```

- `to_number(text, text)`

Description: Converts the values of the string type into the numbers in the specified format.

Return type: numeric

For example:

```
SELECT to_number('12,454.8-', '99G999D9S');
to_number
-----
-12454.8
(1 row)
```

- `to_timestamp(double precision)`

Description: Converts a UNIX century into a timestamp.

Return type: timestamp with time zone

For example:

```
SELECT to_timestamp(1284352323);
to_timestamp
-----
2010-09-13 12:32:03+08
(1 row)
```

- `to_timestamp(string [,fmt])`

Description: Converts a string into a value of the timestamp type according to the format specified by **fmt**. When **fmt** is not specified, perform the conversion according to the format specified by **nls\_timestamp\_format**.

In **to\_timestamp** in GaussDB(DWS):

- If the input year YYYY = 0, an error is reported.
- If the input year YYYY < 0 to specify SYYYY in **fmt**, the year with the value of n (an absolute value) BC is output correctly.

Characters in the **fmt** must match the schema for formatting the data and time. Otherwise, an error is reported.

Return type: timestamp without time zone

For example:

```
SHOW nls_timestamp_format;
nls_timestamp_format
```



```

-----
DD-Mon-YYYY HH:MI:SS.FF AM
(1 row)

SELECT to_timestamp('12-sep-2014');
       to_timestamp
-----
2014-09-12 00:00:00
(1 row)
SELECT to_timestamp('12-Sep-10 14:10:10.123000','DD-Mon-YY HH24:MI:SS.FF');
       to_timestamp
-----
2010-09-12 14:10:10.123
(1 row)
SELECT to_timestamp('-1','SYYYY');
       to_timestamp
-----
0001-01-01 00:00:00 BC
(1 row)
SELECT to_timestamp('98','RR');
       to_timestamp
-----
1998-01-01 00:00:00
(1 row)
SELECT to_timestamp('01','RR');
       to_timestamp
-----
2001-01-01 00:00:00
(1 row)

```

- `to_timestamp(text, text)`  
Description: Converts values of the string type into the timestamp of the specified type.  
Return type: timestamp

For example:

```

SELECT to_timestamp('05 Dec 2000', 'DD Mon YYYY');
       to_timestamp
-----
2000-12-05 00:00:00
(1 row)

```

**Table 6-6** Template patterns for numeric formatting

| Schema     | Description                           |
|------------|---------------------------------------|
| 9          | Value with specified digits           |
| 0          | Values with leading zeros             |
| Period (.) | Decimal point                         |
| Comma (,)  | Group (thousand) separator            |
| PR         | Negative values in angle brackets     |
| S          | Sign anchored to number (uses locale) |
| L          | Currency symbol (uses locale)         |
| D          | Decimal point (uses locale)           |
| G          | Group separator (uses locale)         |

| Schema   | Description                                                           |
|----------|-----------------------------------------------------------------------|
| MI       | Minus sign in the specified position (if the number is less than 0)   |
| PL       | Plus sign in the specified position (if the number is greater than 0) |
| SG       | Plus or minus sign in the specified position                          |
| RN       | Roman numerals (the input values are between 1 and 3999)              |
| TH or th | Ordinal number suffix                                                 |
| V        | Shifts specified number of digits (decimal)                           |

## 6.10 Geometric Functions and Operators

### Geometric Operators

- +

Description: Translation

For example:

```
SELECT box '((0,0),(1,1))' + point '(2.0,0)' AS RESULT;
result
-----
(3,1),(2,0)
(1 row)
```

- -

Description: Translation

For example:

```
SELECT box '((0,0),(1,1))' - point '(2.0,0)' AS RESULT;
result
-----
(-1,1),(-2,0)
(1 row)
```

- \*

Description: Scaling out/rotation

For example:

```
SELECT box '((0,0),(1,1))' * point '(2.0,0)' AS RESULT;
result
-----
(2,2),(0,0)
(1 row)
```

- /

Description: Scaling in/rotation

For example:

```
SELECT box '((0,0),(2,2))' / point '(2.0,0)' AS RESULT;
result
-----
```

- ```
(1,1),(0,0)
(1 row)
```

  - #

Description: Point or box of intersection

For example:

```
SELECT box'((1,-1),(-1,1))' # box'((1,1),(-1,-1))' AS RESULT;
result
-----
(1,1),(-1,-1)
(1 row)
```
  - #

Description: Number of paths or polygon vertexs

For example:

```
SELECT # path'((1,0),(0,1),(-1,0))' AS RESULT;
result
-----
3
(1 row)
```
  - @-@

Description: Length or circumference

For example:

```
SELECT @-@ path'((0,0),(1,0))' AS RESULT;
result
-----
2
(1 row)
```
  - @@

Description: Center of box

For example:

```
SELECT @@ circle'((0,0),10)' AS RESULT;
result
-----
(0,0)
(1 row)
```
  - ##

Description: Closest point to first figure on second figure.

For example:

```
SELECT point'(0,0)' ## box'((2,0),(0,2))' AS RESULT;
result
-----
(0,0)
(1 row)
```
  - <->

Description: Distance between the two figures.

For example:

```
SELECT circle'((0,0),1)' <-> circle'((5,0),1)' AS RESULT;
result
-----
3
(1 row)
```
  - &&

Description: Overlaps? (One point in common makes this true.)

For example:

```
SELECT box '((0,0),(1,1))' && box '((0,0),(2,2))' AS RESULT;
result
-----
t
(1 row)
```

- <<

Description: Is strictly left of (no common horizontal coordinate)?

For example:

```
SELECT circle '((5,0),1)' << circle '((0,0),1)' AS RESULT;
result
-----
t
(1 row)
```

- >>

Description: Is strictly right of (no common horizontal coordinate)?

For example:

```
SELECT circle '((5,0),1)' >> circle '((0,0),1)' AS RESULT;
result
-----
t
(1 row)
```

- &<

Description: Does not extend to the right of?

For example:

```
SELECT box '((0,0),(1,1))' &< box '((0,0),(2,2))' AS RESULT;
result
-----
t
(1 row)
```

- &>

Description: Does not extend to the left of?

For example:

```
SELECT box '((0,0),(3,3))' &> box '((0,0),(2,2))' AS RESULT;
result
-----
t
(1 row)
```

- <<|

Description: Is strictly below (no common horizontal coordinate)?

For example:

```
SELECT box '((0,0),(3,3))' <<| box '((3,4),(5,5))' AS RESULT;
result
-----
t
(1 row)
```

- |>>

Description: Is strictly above (no common horizontal coordinate)?

For example:

```
SELECT box '((3,4),(5,5))' |>> box '((0,0),(3,3))' AS RESULT;
result
-----
t
(1 row)
```

- **&<|**

Description: Does not extend above?

For example:

```
SELECT box '((0,0),(1,1))' &<| box '((0,0),(2,2))' AS RESULT;
result
-----
t
(1 row)
```
- **|&>**

Description: Does not extend below?

For example:

```
SELECT box '((0,0),(3,3))' |&> box '((0,0),(2,2))' AS RESULT;
result
-----
t
(1 row)
```
- **<^**

Description: Is below (allows touching)?

For example:

```
SELECT box '((0,0),(-3,-3))' <^ box '((0,0),(2,2))' AS RESULT;
result
-----
t
(1 row)
```
- **>^**

Description: Is above (allows touching)?

For example:

```
SELECT box '((0,0),(2,2))' >^ box '((0,0),(-3,-3))' AS RESULT;
result
-----
t
(1 row)
```
- **?#**

Description: Intersect?

For example:

```
SELECT lseg '((-1,0),(1,0))' ?# box '((-2,-2),(2,2))' AS RESULT;
result
-----
t
(1 row)
```
- **?-**

Description: Is horizontal?

For example:

```
SELECT ?- lseg '((-1,0),(1,0))' AS RESULT;
result
-----
t
(1 row)
```
- **?-**

Description: Are horizontally aligned?

For example:

```
SELECT point '(1,0)' ?- point '(0,0)' AS RESULT;
result
```

- ```
-----
t
(1 row)
```

  - ?|

Description: Is vertical?

For example:

```
SELECT ?| lseg '((-1,0),(1,0))' AS RESULT;
result
-----
f
(1 row)
```
  - ?|

Description: Are vertically aligned?

For example:

```
SELECT point '(0,1)' ?| point '(0,0)' AS RESULT;
result
-----
t
(1 row)
```
  - ?-|

Description: Are perpendicular?

For example:

```
SELECT lseg '((0,0),(0,1))' ?-| lseg '((0,0),(1,0))' AS RESULT;
result
-----
t
(1 row)
```
  - ?||

Description: Are parallel?

For example:

```
SELECT lseg '((-1,0),(1,0))' ?|| lseg '((-1,2),(1,2))' AS RESULT;
result
-----
t
(1 row)
```
  - @>

Description: Contains?

For example:

```
SELECT circle '((0,0),2)' @> point '(1,1)' AS RESULT;
result
-----
t
(1 row)
```
  - <@

Description: Contained in or on?

For example:

```
SELECT point '(1,1)' <@ circle '((0,0),2)' AS RESULT;
result
-----
t
(1 row)
```
  - ~=

Description: Same as?

For example:

```
SELECT polygon '((0,0),(1,1))' ~= polygon '((1,1),(0,0))' AS RESULT;  
result  
-----  
t  
(1 row)
```

## Geometric Functions

- **area(object)**

Description: Area calculation

Return type: double precision

For example:

```
SELECT area(box '((0,0),(1,1))') AS RESULT;  
result  
-----  
1  
(1 row)
```

- **center(object)**

Description: Figure center calculation

Return type: point

For example:

```
SELECT center(box '((0,0),(1,2))') AS RESULT;  
result  
-----  
(0.5,1)  
(1 row)
```

- **diameter(circle)**

Description: Circle diameter calculation

Return type: double precision

For example:

```
SELECT diameter(circle '((0,0),2.0)') AS RESULT;  
result  
-----  
4  
(1 row)
```

- **height(box)**

Description: Vertical size of box

Return type: double precision

For example:

```
SELECT height(box '((0,0),(1,1))') AS RESULT;  
result  
-----  
1  
(1 row)
```

- **isclosed(path)**

Description: A closed path?

Return type: Boolean

For example:

```
SELECT isclosed(path '((0,0),(1,1),(2,0))') AS RESULT;  
result  
-----
```

- t  
(1 row)

  - **isopen(path)**  
Description: An open path?  
Return type: Boolean  
For example:  
SELECT isopen(path '((0,0),(1,1),(2,0))) AS RESULT;  
result  
-----  
t  
(1 row)
- **length(object)**  
Description: Length calculation  
Return type: double precision  
For example:  
SELECT length(path '((1,0),(1,0))) AS RESULT;  
result  
-----  
4  
(1 row)
- **npoints(path)**  
Description: Number of points in path  
Return type: int  
For example:  
SELECT npoints(path '((0,0),(1,1),(2,0))) AS RESULT;  
result  
-----  
3  
(1 row)
- **npoints(polygon)**  
Description: Number of points in polygon  
Return type: int  
For example:  
SELECT npoints(polygon '((1,1),(0,0))) AS RESULT;  
result  
-----  
2  
(1 row)
- **pclose(path)**  
Description: Converts path to closed.  
Return type: path  
For example:  
SELECT pclose(path '((0,0),(1,1),(2,0))) AS RESULT;  
result  
-----  
((0,0),(1,1),(2,0))  
(1 row)
- **popen(path)**  
Description: Converts path to open.  
Return type: path  
For example:



```
SELECT popen(path '((0,0),(1,1),(2,0)')) AS RESULT;
      result
-----
[(0,0),(1,1),(2,0)]
(1 row)
```

- **radius(circle)**  
Description: Circle diameter calculation  
Return type: double precision  
For example:

```
SELECT radius(circle '((0,0),2.0)') AS RESULT;
      result
-----
                2
(1 row)
```

- **width(box)**  
Description: Horizontal size of box  
Return type: double precision  
For example:

```
SELECT width(box '((0,0),(1,1)')) AS RESULT;
      result
-----
                1
(1 row)
```

## Geometric Type Conversion Functions

- **box(circle)**  
Description: Circle to box  
Return type: box  
For example:

```
SELECT box(circle '((0,0),2.0)') AS RESULT;
      result
-----
(1.41421356237309,1.41421356237309),(-1.41421356237309,-1.41421356237309)
(1 row)
```

- **box(point, point)**  
Description: Points to box  
Return type: box  
For example:

```
SELECT box(point '(0,0)', point '(1,1)') AS RESULT;
      result
-----
(1,1),(0,0)
(1 row)
```

- **box(polygon)**  
Description: Polygon to box  
Return type: box  
For example:

```
SELECT box(polygon '((0,0),(1,1),(2,0)')) AS RESULT;
      result
-----
(2,1),(0,0)
(1 row)
```

- **circle(box)**  
 Description: Box to circle  
 Return type: circle  
 For example:  

```
SELECT circle(box '((0,0),(1,1)))' AS RESULT;
      result
-----
<(0.5,0.5),0.707106781186548>
(1 row)
```
- **circle(point, double precision)**  
 Description: Center and radius to circle  
 Return type: circle  
 For example:  

```
SELECT circle(point '(0,0)', 2.0) AS RESULT;
      result
-----
<(0,0),2>
(1 row)
```
- **circle(polygon)**  
 Description: Polygon to circle  
 Return type: circle  
 For example:  

```
SELECT circle(polygon '((0,0),(1,1),(2,0)))' AS RESULT;
      result
-----
<(1,0.3333333333333333),0.924950591148529>
(1 row)
```
- **lseg(box)**  
 Description: Box diagonal to line segment  
 Return type: lseg  
 For example:  

```
SELECT lseg(box '((-1,0),(1,0))' AS RESULT;
      result
-----
[(1,0),(-1,0)]
(1 row)
```
- **lseg(point, point)**  
 Description: Points to line segment  
 Return type: lseg  
 For example:  

```
SELECT lseg(point '(-1,0)', point '(1,0)') AS RESULT;
      result
-----
[(-1,0),(1,0)]
(1 row)
```
- **path(polygon)**  
 Description: Polygon to path  
 Return type: path  
 For example:  

```
SELECT path(polygon '((0,0),(1,1),(2,0)))' AS RESULT;
      result
```

```
-----  
(0,0),(1,1),(2,0)  
(1 row)
```

- `point(double precision, double precision)`

Description: Points

Return type: point

For example:

```
SELECT point(23.4, -44.5) AS RESULT;  
result
```

```
-----  
(23.4,-44.5)  
(1 row)
```

- `point(box)`

Description: Center of box

Return type: point

For example:

```
SELECT point(box '((-1,0),(1,0))') AS RESULT;  
result
```

```
-----  
(0,0)  
(1 row)
```

- `point(circle)`

Description: Center of circle

Return type: point

For example:

```
SELECT point(circle '((0,0),2.0)') AS RESULT;  
result
```

```
-----  
(0,0)  
(1 row)
```

- `point(lseg)`

Description: Center of line segment

Return type: point

For example:

```
SELECT point(lseg '((-1,0),(1,0))') AS RESULT;  
result
```

```
-----  
(0,0)  
(1 row)
```

- `point(polygon)`

Description: Center of polygon

Return type: point

For example:

```
SELECT point(polygon '((0,0),(1,1),(2,0))') AS RESULT;  
result
```

```
-----  
(1,0.3333333333333333)  
(1 row)
```

- `polygon(box)`

Description: Box to 4-point polygon

Return type: polygon

For example:

```
SELECT polygon(box '((0,0),(1,1)')) AS RESULT;
result
-----
((0,0),(0,1),(1,1),(1,0))
(1 row)
```

- **polygon(circle)**

Description: Circle to 12-point polygon

Return type: polygon

For example:

```
SELECT polygon(circle '((0,0),2.0)') AS RESULT;
result
-----
((-2,0),(-1.73205080756888,1),(-1,1.73205080756888),(-1.22464679914735e-16,2),
(1,1.73205080756888),(1.73205080756888,1),(2,2.44929359829471e-16),
(1.73205080756888,-0.9999999999999999),(1,-1.73205080756888),(3.67394039744206e-16,-2),
(-0.9999999999999999,-1.73205080756888),(-1.73205080756888,-1))
(1 row)
```

- **polygon(npts, circle)**

Description: Circle to **npts**-point polygon

Return type: polygon

For example:

```
SELECT polygon(12, circle '((0,0),2.0)') AS RESULT;
result
-----
((-2,0),(-1.73205080756888,1),(-1,1.73205080756888),(-1.22464679914735e-16,2),
(1,1.73205080756888),(1.73205080756888,1),(2,2.44929359829471e-16),
(1.73205080756888,-0.9999999999999999),(1,-1.73205080756888),(3.67394039744206e-16,-2),
(-0.9999999999999999,-1.73205080756888),(-1.73205080756888,-1))
(1 row)
```

- **polygon(path)**

Description: Path to polygon

Return type: polygon

For example:

```
SELECT polygon(path '((0,0),(1,1),(2,0)')) AS RESULT;
result
-----
((0,0),(1,1),(2,0))
(1 row)
```

## 6.11 Network Address Functions and Operators

### cidr and inet Operators

The operators <<, <<=, >>, and >>= test for subnet inclusion. They consider only the network parts of the two addresses (ignoring any host part) and determine whether one network is identical to or a subnet of the other.

- <  
Description: Is less than  
For example:  

```
SELECT inet '192.168.1.5' < inet '192.168.1.6' AS RESULT;  
result  
-----  
t  
(1 row)
```
- <=  
Description: Is less than or equals  
For example:  

```
SELECT inet '192.168.1.5' <= inet '192.168.1.5' AS RESULT;  
result  
-----  
t  
(1 row)
```
- =  
Description: Equals  
For example:  

```
SELECT inet '192.168.1.5' = inet '192.168.1.5' AS RESULT;  
result  
-----  
t  
(1 row)
```
- >=  
Description: Is greater than or equals  
For example:  

```
SELECT inet '192.168.1.5' >= inet '192.168.1.5' AS RESULT;  
result  
-----  
t  
(1 row)
```
- >  
Description: Is greater than  
For example:  

```
SELECT inet '192.168.1.5' > inet '192.168.1.4' AS RESULT;  
result  
-----  
t  
(1 row)
```
- <>  
Description: Does not equal to  
For example:  

```
SELECT inet '192.168.1.5' <> inet '192.168.1.4' AS RESULT;  
result  
-----  
t  
(1 row)
```
- <<  
Description: Is contained in  
For example:  

```
SELECT inet '192.168.1.5' << inet '192.168.1/24' AS RESULT;  
result
```

- ```
-----  
t  
(1 row)
```

• <<=

Description: Is contained in or equals

For example:

```
SELECT inet '192.168.1/24' <<= inet '192.168.1/24' AS RESULT;  
result  
-----  
t  
(1 row)
```
- >>

Description: Contains

For example:

```
SELECT inet '192.168.1/24' >> inet '192.168.1.5' AS RESULT;  
result  
-----  
t  
(1 row)
```
- >>=

Description: Contains or equals

For example:

```
SELECT inet '192.168.1/24' >>= inet '192.168.1/24' AS RESULT;  
result  
-----  
t  
(1 row)
```
- ~

Description: Bitwise NOT

For example:

```
SELECT ~ inet '192.168.1.6' AS RESULT;  
result  
-----  
10.100.10.29  
(1 row)
```
- &

Description: The AND operation is performed on each bit of the two network addresses.

For example:

```
SELECT inet '192.168.1.6' & inet '10.0.0.0' AS RESULT;  
result  
-----  
0.0.0.0  
(1 row)
```
- |

Description: The OR operation is performed on each bit of the two network addresses.

For example:

```
SELECT inet '192.168.1.6' | inet '10.0.0.0' AS RESULT;  
result  
-----  
192.168.1.6  
(1 row)
```

- +

Description: Addition

For example:

```
SELECT inet '192.168.1.6' + 25 AS RESULT;
result
-----
192.168.1.31
(1 row)
```

- -

Description: Subtraction

For example:

```
SELECT inet '192.168.1.43' - 36 AS RESULT;
result
-----
192.168.1.7
(1 row)
```

- -

Description: Subtraction

For example:

```
SELECT inet '192.168.1.43' - inet '192.168.1.19' AS RESULT;
result
-----
24
(1 row)
```

## cidr and inet Functions

The **abbrev**, **host**, and **text** functions are primarily intended to offer alternative display formats.

- abbrev(inet)

Description: Abbreviated display format as text

Return type: text

For example:

```
SELECT abbrev(inet '10.1.0.0/16') AS RESULT;
result
-----
10.1.0.0/16
(1 row)
```

- abbrev(cidr)

Description: Abbreviated display format as text

Return type: text

For example:

```
SELECT abbrev(cidr '10.1.0.0/16') AS RESULT;
result
-----
10.1/16
(1 row)
```

- broadcast(inet)

Description: Broadcast address for network

Return type: inet

For example:

```
SELECT broadcast('192.168.1.5/24') AS RESULT;  
result  
-----  
192.168.1.255/24  
(1 row)
```

- **family(inet)**  
Description: Extracts family of address; **4** for IPv4, **6** for IPv6  
Return type: int  
For example:

```
SELECT family('::1') AS RESULT;  
result  
-----  
6  
(1 row)
```

- **host(inet)**  
Description: Extracts IP address as text.  
Return type: text  
For example:

```
SELECT host('192.168.1.5/24') AS RESULT;  
result  
-----  
192.168.1.5  
(1 row)
```

- **hostmask(inet)**  
Description: Constructs host mask for network.  
Return type: inet  
For example:

```
SELECT hostmask('192.168.23.20/30') AS RESULT;  
result  
-----  
10.0.0.3  
(1 row)
```

- **masklen(inet)**  
Description: Extracts subnet mask length.  
Return type: int  
For example:

```
SELECT masklen('192.168.1.5/24') AS RESULT;  
result  
-----  
24  
(1 row)
```

- **netmask(inet)**  
Description: Constructs a subnet mask for the network.  
Return type: inet  
For example:

```
SELECT netmask('192.168.1.5/24') AS RESULT;  
result  
-----  
255.255.255.0  
(1 row)
```

- **network(inet)**  
Description: Extracts network part of address.



Return type: `cidr`

For example:

```
SELECT network('192.168.1.5/24') AS RESULT;  
result  
-----  
192.168.1.0/24  
(1 row)
```

- `set_masklen(inet, int)`

Description: Sets subnet mask length for **inet** value.

Return type: `inet`

For example:

```
SELECT set_masklen('192.168.1.5/24', 16) AS RESULT;  
result  
-----  
192.168.1.5/16  
(1 row)
```

- `set_masklen(cidr, int)`

Description: Sets subnet mask length for **cidr** value.

Return type: `cidr`

For example:

```
SELECT set_masklen('192.168.1.0/24'::cidr, 16) AS RESULT;  
result  
-----  
192.168.0.0/16  
(1 row)
```

- `text(inet)`

Description: Extracts IP address and subnet mask length as text.

Return type: `text`

For example:

```
SELECT text(inet '192.168.1.5') AS RESULT;  
result  
-----  
192.168.1.5/32  
(1 row)
```

Any **cidr** value can be cast to **inet** implicitly or explicitly; therefore, the functions shown above as operating on **inet** also work on **cidr** values. An **inet** value can be cast to **cidr**. After the conversion, any bits to the right of the subnet mask are silently zeroed to create a valid **cidr** value. In addition, you can cast a text string to **inet** or **cidr** using normal casting syntax. For example, **inet(expression)** or **colname::cidr**.

## macaddr Functions

The function **trunc(macaddr)** returns a MAC address with the last 3 bytes set to zero.

`trunc(macaddr)`

Description: Sets last 3 bytes to zero.

Return type: `macaddr`

For example:

```
SELECT trunc(macaddr '12:34:56:78:90:ab') AS RESULT;  
result  
-----  
12:34:56:00:00:00  
(1 row)
```

The **macaddr** type also supports the standard relational operators (such as > and <=) for lexicographical ordering, and the bitwise arithmetic operators (~, & and |) for NOT, AND and OR.

## 6.12 Text Search Functions and Operators

### Text Search Operators

- @@

Description: Specifies whether the **tsvector**-typed words match the **tsquery**-typed words.

For example:

```
SELECT to_tsvector('fat cats ate rats') @@ to_tsquery('cat & rat') AS RESULT;  
result  
-----  
t  
(1 row)
```

- @@@

Description: Synonym for @@

For example:

```
SELECT to_tsvector('fat cats ate rats') @@@ to_tsquery('cat & rat') AS RESULT;  
result  
-----  
t  
(1 row)
```

- ||

Description: Connects two **tsvector**-typed words.

For example:

```
SELECT 'a:1 b:2'::tsvector || 'c:1 d:2 b:3'::tsvector AS RESULT;  
result  
-----  
'a':1 'b':2,5 'c':3 'd':4  
(1 row)
```

- &&

Description: Performs the AND operation on two **tsquery**-typed words.

For example:

```
SELECT 'fat | rat'::tsquery && 'cat'::tsquery AS RESULT;  
result  
-----  
( 'fat' | 'rat' ) & 'cat'  
(1 row)
```

- ||

Description: Performs the OR operation on two **tsquery**-typed words.

For example:

```
SELECT 'fat | rat'::tsquery || 'cat'::tsquery AS RESULT;  
result  
-----
```

```
( 'fat' | 'rat' ) | 'cat'
(1 row)
```

- !!

Description: **NOT** a **tsquery**

For example:

```
SELECT !! 'cat'::tsquery AS RESULT;
result
-----
!'cat'
(1 row)
```

- @>

Description: Specifies whether a **tsquery**-typed word contains another **tsquery**-typed word.

For example:

```
SELECT 'cat'::tsquery @> 'cat & rat'::tsquery AS RESULT;
result
-----
f
(1 row)
```

- <@

Description: Specifies whether a **tsquery**-typed word is contained in another **tsquery**-typed word.

For example:

```
SELECT 'cat'::tsquery <@ 'cat & rat'::tsquery AS RESULT;
result
-----
t
(1 row)
```

In addition to the preceding operators, the ordinary B-tree comparison operators (including = and <) are defined for types **tsvector** and **tsquery**.

## Text search functions

- get\_current\_ts\_config()

Description: Gets default text search configuration.

Return type: regconfig

For example:

```
SELECT get_current_ts_config();
get_current_ts_config
-----
english
(1 row)
```

- length(tsvector)

Description: Number of lexemes in a **tsvector**-typed word.

Return type: integer

For example:

```
SELECT length('fat:2,4 cat:3 rat:5A'::tsvector);
length
-----
3
(1 row)
```

- numnode(tsquery)

Description: Number of lexemes plus **tsquery** operators

Return type: integer

For example:

```
SELECT numnode('(fat & rat) | cat::tsquery);
numnode
-----
      5
(1 row)
```

- **plainto\_tsquery**([ config regconfig , ] query text)

Description: Generates **tsquery** lexemes without punctuation.

Return type: tsquery

For example:

```
SELECT plainto_tsquery('english', 'The Fat Rats');
plainto_tsquery
-----
'fat' & 'rat'
(1 row)
```

- **querytree**(query tsquery)

Description: Gets indexable part of a **tsquery**.

Return type: text

For example:

```
SELECT querytree('foo & ! bar::tsquery);
querytree
-----
'foo'
(1 row)
```

- **setweight**(tsvector, "char")

Description: Assigns weight to each element of **tsvector**.

Return type: tsvector

For example:

```
SELECT setweight('fat:2,4 cat:3 rat:5B::tsvector, 'A');
setweight
-----
'cat':3A 'fat':2A,4A 'rat':5A
(1 row)
```

- **strip**(tsvector)

Description: Removes positions and weights from **tsvector**.

Return type: tsvector

For example:

```
SELECT strip('fat:2,4 cat:3 rat:5A::tsvector);
strip
-----
'cat' 'fat' 'rat'
(1 row)
```

- **to\_tsquery**([ config regconfig , ] query text)

Description: Normalizes words and converts them to **tsquery**.

Return type: tsquery

For example:

```
SELECT to_tsquery('english', 'The & Fat & Rats');
to_tsquery
-----
```

```
'fat' & 'rat'
(1 row)
```

- `to_tsvector([ config regconfig , ] document text)`

Description: Reduces document text to **tsvector**.

Return type: `tsvector`

For example:

```
SELECT to_tsvector('english', 'The Fat Rats');
 to_tsvector
-----
'fat':2 'rat':3
(1 row)
```

- `ts_headline([ config regconfig, ] document text, query tsquery [, options text ])`

Description: Highlights a query match.

Return type: `text`

For example:

```
SELECT ts_headline('x y z', 'z'::tsquery);
 ts_headline
-----
x y <b>z</b>
(1 row)
```

- `ts_rank([ weights float4[], ] vector tsvector, query tsquery [, normalization integer ])`

Description: Ranks document for query.

Return type: `float4`

For example:

```
SELECT ts_rank('hello world'::tsvector, 'world'::tsquery);
 ts_rank
-----
.0607927
(1 row)
```

- `ts_rank_cd([ weights float4[], ] vector tsvector, query tsquery [, normalization integer ])`

Description: Ranks document for query using cover density.

Return type: `float4`

For example:

```
SELECT ts_rank_cd('hello world'::tsvector, 'world'::tsquery);
 ts_rank_cd
-----
0
(1 row)
```

- `ts_rewrite(query tsquery, target tsquery, substitute tsquery)`

Description: Replaces **tsquery**-typed word.

Return type: `tsquery`

For example:

```
SELECT ts_rewrite('a & b'::tsquery, 'a'::tsquery, 'foo|bar'::tsquery);
 ts_rewrite
-----
'b' & ( 'foo' | 'bar' )
(1 row)
```

- `ts_rewrite(query tsquery, select text)`

Description: Replaces **tsquery** data in the target with the result of a **SELECT** command.

Return type: tsquery

For example:

```
SELECT ts_rewrite('world'::tsquery, 'select "world"::tsquery, "hello"::tsquery');
ts_rewrite
-----
'hello'
(1 row)
```

## Text Search Debugging Functions

- `ts_debug([ config regconfig, ] document text, OUT alias text, OUT description text, OUT token text, OUT dictionaries regdictionary[], OUT dictionary regdictionary, OUT lexemes text[])`

Description: Tests a configuration.

Return type: setof record

For example:

```
SELECT ts_debug('english', 'The Brightest supernovae');
ts_debug
-----
(asciiword,"Word, all ASCII",The,{english_stem},english_stem,{})
(blank,"Space symbols", " ",{},{,})
(asciiword,"Word, all ASCII",Brightest,{english_stem},english_stem,{brightest})
(blank,"Space symbols", " ",{},{,})
(asciiword,"Word, all ASCII",supernovae,{english_stem},english_stem,{supernova})
(5 rows)
```

- `ts_lexize(dict regdictionary, token text)`

Description: Tests a data dictionary.

Return type: text[]

For example:

```
SELECT ts_lexize('english_stem', 'stars');
ts_lexize
-----
{star}
(1 row)
```

- `ts_parse(parser_name text, document text, OUT tokid integer, OUT token text)`

Description: Tests a parser.

Return type: setof record

For example:

```
SELECT ts_parse('default', 'foo - bar');
ts_parse
-----
(1,foo)
(12," ")
(12,"- ")
(1,bar)
(4 rows)
```

- `ts_parse(parser_oid oid, document text, OUT tokid integer, OUT token text)`

Description: Tests a parser.

Return type: setof record

For example:

```
SELECT ts_parse(3722, 'foo - bar');
ts_parse
-----
(1,foo)
(12," ")
(12,"- ")
(1,bar)
(4 rows)
```

- `ts_token_type(parser_name text, OUT tokid integer, OUT alias text, OUT description text)`

Description: Gets token types defined by parser.

Return type: setof record

For example:

```
SELECT ts_token_type('default');
ts_token_type
-----
(1,asciiword,"Word, all ASCII")
(2,word,"Word, all letters")
(3,numword,"Word, letters and digits")
(4,email,"Email address")
(5,url,URL)
(6,host,Host)
(7,sfloat,"Scientific notation")
(8,version,"Version number")
(9,hword_numpart,"Hyphenated word part, letters and digits")
(10,hword_part,"Hyphenated word part, all letters")
(11,hword_asciipart,"Hyphenated word part, all ASCII")
(12,blank,"Space symbols")
(13,tag,"XML tag")
(14,protocol,"Protocol head")
(15,numhword,"Hyphenated word, letters and digits")
(16,asciihword,"Hyphenated word, all ASCII")
(17,hword,"Hyphenated word, all letters")
(18,url_path,"URL path")
(19,file,"File or path name")
(20,float,"Decimal notation")
(21,int,"Signed integer")
(22,uint,"Unsigned integer")
(23,entity,"XML entity")
(23 rows)
```

- `ts_token_type(parser_oid oid, OUT tokid integer, OUT alias text, OUT description text)`

Description: Gets token types defined by parser.

Return type: setof record

For example:

```
SELECT ts_token_type(3722);
ts_token_type
-----
(1,asciiword,"Word, all ASCII")
(2,word,"Word, all letters")
(3,numword,"Word, letters and digits")
(4,email,"Email address")
(5,url,URL)
(6,host,Host)
(7,sfloat,"Scientific notation")
(8,version,"Version number")
(9,hword_numpart,"Hyphenated word part, letters and digits")
(10,hword_part,"Hyphenated word part, all letters")
(11,hword_asciipart,"Hyphenated word part, all ASCII")
(12,blank,"Space symbols")
(13,tag,"XML tag")
(14,protocol,"Protocol head")
(15,numhword,"Hyphenated word, letters and digits")
```

```
(16,asciihword,"Hyphenated word, all ASCII")
(17,hword,"Hyphenated word, all letters")
(18,url_path,"URL path")
(19,file,"File or path name")
(20,float,"Decimal notation")
(21,int,"Signed integer")
(22,uint,"Unsigned integer")
(23,entity,"XML entity")
(23 rows)
```

- `ts_stat(sqlquery text, [ weights text, ] OUT word text, OUT ndoc integer, OUT nentry integer)`

Description: Gets statistics of a **tsvector** column.

Return type: setof record

For example:

```
SELECT ts_stat('select "hello world"::tsvector');
 ts_stat
-----
(world,1,1)
(hello,1,1)
(2 rows)
```

## 6.13 UUID Functions

UUID functions are used to generate UUID data (see [UUID Type](#)).

- `uuid_generate_v1()`

Description: Generates a UUID sequence number.

Return type: UUID

Example:

```
SELECT uuid_generate_v1();
 uuid_generate_v1
-----
c71ceaca-a175-11e9-a920-797ff7000001
(1 row)
```

### NOTE

The `uuid_generate_v1` function generates UUIDs based on the time information, cluster node ID, and thread ID that generates the sequence. Each UUID is globally unique in a cluster, but there is a low probability that a UUID is duplicated among multiple clusters.

- `sys_guid()`

Description: Generate a sequence number that is the same as the sequence number generated by the Oracle `sys_guid` method.

Return type: text

Example:

```
SELECT sys_guid();
 sys_guid
-----
4EBD3C74A17A11E9A1BF797FF7000001
(1 row)
```

### NOTE

The data generation principle of the `sys_guid` function is the same as that of the `uuid_generate_v1` function.



## 6.14 JSON Functions

JSON functions are used to generate JSON data (see [JSON Types](#)).

- `array_to_json(anyarray [, pretty_bool])`

Description: Returns the array as JSON. A multi-dimensional array becomes a JSON array of arrays. Line feeds will be added between dimension-1 elements if **pretty\_bool** is **true**.

Return type: json

For example:

```
SELECT array_to_json('{{1,5},{99,100}}::int[]);
array_to_json
-----
[[1,5],[99,100]]
(1 row)
```

- `row_to_json(record [, pretty_bool])`

Description: Returns the row as JSON. Line feeds will be added between level-1 elements if **pretty\_bool** is **true**.

Return type: json

For example:

```
SELECT row_to_json(row(1,'foo'));
row_to_json
-----
{"f1":1,"f2":"foo"}
(1 row)
```

## 6.15 HLL Functions and Operators

### Hash Functions

- `hll_hash_boolean(bool)`

Description: Hashes data of the bool type.

Return type: hll\_hashval

For example:

```
SELECT hll_hash_boolean(FALSE);
hll_hash_boolean
-----
5048724184180415669
(1 row)
```

- `hll_hash_boolean(bool, int32)`

Description: Configures a hash seed (that is, change the hash policy) and hashes data of the bool type.

Return type: hll\_hashval

For example:

```
SELECT hll_hash_boolean(FALSE, 10);
hll_hash_boolean
-----
391264977436098630
(1 row)
```

- `hll_hash_smallint(smallint)`  
Description: Hashes data of the `smallint` type.  
Return type: `hll_hashval`

For example:

```
SELECT hll_hash_smallint(100::smallint);
hll_hash_smallint
-----
4631120266694327276
(1 row)
```

 **NOTE**

If parameters with the same numeric value are hashed using different data types, the data will differ, because hash functions select different calculation policies for each type.

- `hll_hash_smallint(smallint, int32)`  
Description: Configures a hash seed (that is, change the hash policy) and hashes data of the `smallint` type.  
Return type: `hll_hashval`

For example:

```
SELECT hll_hash_smallint(100::smallint, 10);
hll_hash_smallint
-----
8349353095166695771
(1 row)
```

- `hll_hash_integer(integer)`  
Description: Hashes data of the `integer` type.  
Return type: `hll_hashval`

For example:

```
SELECT hll_hash_integer(0);
hll_hash_integer
-----
-3485513579396041028
(1 row)
```

- `hll_hash_integer(integer, int32)`  
Description: Hashes data of the `integer` type and configures a hash seed (that is, change the hash policy).  
Return type: `hll_hashval`

For example:

```
SELECT hll_hash_integer(0, 10);
hll_hash_integer
-----
183371090322255134
(1 row)
```

- `hll_hash_bigint(bigint)`  
Description: Hashes data of the `bigint` type.  
Return type: `hll_hashval`

For example:

```
SELECT hll_hash_bigint(100::bigint);
hll_hash_bigint
-----
8349353095166695771
(1 row)
```

- **hll\_hash\_bigint(bigint, int32)**  
Description: Hashes data of the bigint type and configures a hash seed (that is, change the hash policy).  
Return type: hll\_hashval  
For example:  

```
SELECT hll_hash_bigint(100::bigint, 10);
 hll_hash_bigint
-----
4631120266694327276
(1 row)
```
- **hll\_hash\_bytea(bytea)**  
Description: Hashes data of the bytea type.  
Return type: hll\_hashval  
For example:  

```
SELECT hll_hash_bytea(E'\x');
 hll_hash_bytea
-----
0
(1 row)
```
- **hll\_hash\_bytea(bytea, int32)**  
Description: Hashes data of the bytea type and configures a hash seed (that is, change the hash policy).  
Return type: hll\_hashval  
For example:  

```
SELECT hll_hash_bytea(E'\x', 10);
 hll_hash_bytea
-----
6574525721897061910
(1 row)
```
- **hll\_hash\_text(text)**  
Description: Hashes data of the text type.  
Return type: hll\_hashval  
For example:  

```
SELECT hll_hash_text('AB');
 hll_hash_text
-----
5365230931951287672
(1 row)
```
- **hll\_hash\_text(text, int32)**  
Description: Hashes data of the text type and configures a hash seed (that is, change the hash policy).  
Return type: hll\_hashval  
For example:  

```
SELECT hll_hash_text('AB', 10);
 hll_hash_text
-----
7680762839921155903
(1 row)
```
- **hll\_hash\_any(anytype)**  
Description: Hashes data of any type.  
Return type: hll\_hashval

For example:

```
select hll_hash_any(1);
      hll_hash_any
-----
-8604791237420463362
(1 row)

select hll_hash_any('08:00:2b:01:02:03'::macaddr);
      hll_hash_any
-----
-4883882473551067169
(1 row)
```

- `hll_hash_any(anytype, int32)`

Description: Hashes data of any type and configures a hash seed (that is, change the hash policy).

Return type: `hll_hashval`

For example:

```
select hll_hash_any(1, 10);
      hll_hash_any
-----
-1478847531811254870
(1 row)
```

- `hll_hashval_eq(hll_hashval, hll_hashval)`

Description: Compares two pieces of data of the `hll_hashval` type to check whether they are the same.

Return type: `bool`

For example:

```
select hll_hashval_eq(hll_hash_integer(1), hll_hash_integer(1));
      hll_hashval_eq
-----
t
(1 row)
```

- `hll_hashval_ne(hll_hashval, hll_hashval)`

Description: Compares two pieces of data of the `hll_hashval` type to check whether they are different.

Return type: `bool`

For example:

```
select hll_hashval_ne(hll_hash_integer(1), hll_hash_integer(1));
      hll_hashval_ne
-----
f
(1 row)
```

## Precision Functions

HLL supports **explicit**, **sparse**, and **full** modes. **explicit** and **sparse** excel when the data scale is small, and barely produce errors in calculation results. When the number of distinct values increases, **full** becomes more suitable, but produces some errors. The following functions are used to view precision parameters in HLLs.

- `hll_schema_version(hll)`

Description: Checks the schema version in the current HLL.

For example:

```
select hll_schema_version(hll_empty());
hll_schema_version
-----
1
(1 row)
```

- **hll\_type(hll)**

Description: Checks the type of the current HLL.

For example:

```
select hll_type(hll_empty());
hll_type
-----
1
(1 row)
```

- **hll\_log2m(hll)**

Description: Check the value of log2m of the current HLL. This value affects the error rate in calculating the number of distinct values by the HLL. The formula for calculating the error rate is as follows:

$$\pm 1.04 / \sqrt{2 \wedge \log 2m}$$

For example:

```
select hll_log2m(hll_empty());
hll_log2m
-----
11
(1 row)
```

- **hll\_regwidth(hll)**

Description: Checks the number of bits of buckets in a hll data structure.

For example:

```
select hll_regwidth(hll_empty());
hll_regwidth
-----
5
(1 row)
```

- **hll\_expthresh(hll)**

Description: Obtains the size of **expthresh** in the current HLL. An HLL usually switches from the **explicit** mode to the **sparse** mode and then to the **full** mode. This process is called the promotion hierarchy policy. You can change the value of **expthresh** to change the policy. For example, if **expthresh** is **0**, an HLL will skip the **explicit** mode and directly enter the **sparse** mode. If the value of **expthresh** is explicitly set to a value ranging from 1 to 7, this function returns  $2^{\text{expthresh}}$ .

For example:

```
select hll_expthresh(hll_empty());
hll_expthresh
-----
(-1,160)
(1 row)

select hll_expthresh(hll_empty(11,5,3));
hll_expthresh
-----
(8,8)
(1 row)
```

- **hll\_sparseon(hll)**

Description: Specifies whether to enable the **sparse** mode. **0** indicates **off** and **1** indicates **on**.

For example:

```
select hll_sparseon(hll_empty());
hll_sparseon
-----
      1
(1 row)
```

## Aggregation Functions

- `hll_add_agg(hll_hashval)`

Description: Groups hashed data into HLL.

Return type: `hll`

For example:

```
-- Prepare data:
create table t_id(id int);
insert into t_id values(generate_series(1,500));
create table t_data(a int, c text);
insert into t_data select mod(id,2), id from t_id;

-- Create another table and specify an HLL column:
create table t_a_c_hll(a int, c hll);

-- Use GROUP BY on column a to group data, and insert the data to the HLL:
insert into t_a_c_hll select a, hll_add_agg(hll_hash_text(c)) from t_data group by a;

-- Calculate the number of distinct values for each group in the HLL:
select a, #c as cardinality from t_a_c_hll order by a;
a | cardinality
--+-----
0 | 250.741759091658
1 | 250.741759091658
(2 rows)
```

- `hll_add_agg(hll_hashval, int32 log2m)`

Description: Groups hashed data into HLL and sets the **log2m** parameter. The parameter value ranges from 10 to 16.

Return type: `hll`

For example:

```
Select hll_cardinality(hll_add_agg(hll_hash_text(c), 10)) from t_data;
hll_cardinality
-----
503.932348927339
(1 row)
```

- `hll_add_agg(hll_hashval, int32 log2m, int32 regwidth)`

Description: Groups hashed data into HLL and sets the **log2m** and **regwidth** parameters in sequence. The value of **regwidth** ranges from 1 to 5.

Return type: `hll`

For example:

```
Select hll_cardinality(hll_add_agg(hll_hash_text(c), NULL, 1)) from t_data;
hll_cardinality
-----
496.628982624022
(1 row)
```

- `hll_add_agg(hll_hashval, int32 log2m, int32 regwidth, int64 expthresh)`

Description: Groups hashed data into HLL and sets the parameters **log2m**, **regwidth**, and **expthresh** in sequence. The value of **expthresh** is an integer ranging from -1 to 7. **expthresh** is used to specify the threshold for switching

from the **explicit** mode to the **sparse** mode. **-1** indicates the auto mode; **0** indicates that the **explicit** mode is skipped; a value from 1 to 7 indicates that the mode is switched when the number of distinct values reaches  $2^{\text{expthresh}}$ .

Return type: hll

For example:

```
Select hll_cardinality(hll_add_agg(hll_hash_text(c), NULL, 1, 4)) from t_data;
hll_cardinality
-----
496.628982624022
(1 row)
```

- `hll_add_agg(hll_hashval, int32 log2m, int32 regwidth, int64 expthresh, int32 sparseon)`

Description: Groups hashed data into HLL and sets the **log2m**, **regwidth**, **expthresh**, and **sparseon** parameters in sequence. The value of **sparseon** is **0** or **1**.

Return type: hll

For example:

```
Select hll_cardinality(hll_add_agg(hll_hash_text(c), NULL, 1, 4, 0)) from t_data;
hll_cardinality
-----
496.628982624022
(1 row)
```

- `hll_union_agg(hll)`

Description: Perform the **UNION** operation on multiple pieces of data of the hll type to obtain one HLL.

Return type: hll

For example:

```
-- Perform the UNION operation on data of the hll type in each group to obtain one HLL, and
calculate the number of distinct values:
select #hll_union_agg(c) as cardinality from t_a_c_hll;
cardinality
-----
496.628982624022
(1 row)
```

#### NOTE

To perform **UNION** on data in multiple HLLs, ensure that the HLLs have the same precision. Otherwise, **UNION** cannot be performed. This restriction also applies to the `hll_union(hll, hll)` function.

## Functional Functions

- `hll_print(hll)`

Description: Prints some debugging parameters of an HLL.

For example:

```
select hll_print(hll_empty());
hll_print
-----
EMPTY, nregs=2048, nbits=5, expthresh=-1(160), sparseon=1gongne
(1 row)
```

- `hll_empty()`

Description: Creates an empty HLL.

Return type: hll

For example:

```
select hll_empty();
hll_empty
-----
\x118b7f
(1 row)
```

- `hll_empty(int32 log2m)`

Description: Creates an empty HLL and sets the **log2m** parameter. The parameter value ranges from 10 to 16.

Return type: hll

For example:

```
select hll_empty(10);
hll_empty
-----
\x118a7f
(1 row)
```

- `hll_empty(int32 log2m, int32 regwidth)`

Description: Creates an empty HLL and sets the **log2m** and **regwidth** parameters in sequence. The value of **regwidth** ranges from 1 to 5.

Return type: hll

For example:

```
select hll_empty(10, 4);
hll_empty
-----
\x116a7f
(1 row)
```

- `hll_empty(int32 log2m, int32 regwidth, int64 expthresh)`

Description: Creates an empty HLL and sets the **log2m**, **regwidth**, and **expthresh** parameters. The value of **expthresh** is an integer ranging from -1 to 7. This parameter specifies the threshold for switching from the **explicit** mode to the **sparse** mode. -1 indicates the auto mode; 0 indicates that the **explicit** mode is skipped; a value from 1 to 7 indicates that the mode is switched when the number of distinct values reaches  $2^{\text{expthresh}}$ .

Return type: hll

For example:

```
select hll_empty(10, 4, 7);
hll_empty
-----
\x116a48
(1 row)
```

- `hll_empty(int32 log2m, int32 regwidth, int64 expthresh, int32 sparseon)`

Description: Creates an empty HLL and sets the **log2m**, **regwidth**, **expthresh**, and **sparseon** parameters. The value of **sparseon** is 0 or 1.

Return type: hll

For example:

```
select hll_empty(10,4,7,0);
hll_empty
-----
\x116a08
(1 row)
```

- `hll_add(hll, hll_hashval)`

Description: Adds `hll_hashval` to an HLL.



Return type: hll

For example:

```
select hll_add(hll_empty(), hll_hash_integer(1));
       hll_add
-----
\x128b7f8895a3f5af28cafe
(1 row)
```

- **hll\_add\_rev(hll\_hashval, hll)**

Description: Adds hll\_hashval to an HLL. This function works the same as hll\_add, except that the positions of parameters are switched.

Return type: hll

For example:

```
select hll_add_rev(hll_hash_integer(1), hll_empty());
       hll_add_rev
-----
\x128b7f8895a3f5af28cafe
(1 row)
```

- **hll\_eq(hll, hll)**

Description: Compares two HLLs to check whether they are the same.

Return type: bool

For example:

```
select hll_eq(hll_add(hll_empty(), hll_hash_integer(1)), hll_add(hll_empty(), hll_hash_integer(2)));
       hll_eq
-----
f
(1 row)
```

- **hll\_ne(hll, hll)**

Description: Compares two HLLs to check whether they are different.

Return type: bool

For example:

```
select hll_ne(hll_add(hll_empty(), hll_hash_integer(1)), hll_add(hll_empty(), hll_hash_integer(2)));
       hll_ne
-----
t
(1 row)
```

- **hll\_cardinality(hll)**

Description: Calculates the number of distinct values of an HLL.

Return type: int

For example:

```
select hll_cardinality(hll_empty() || hll_hash_integer(1));
       hll_cardinality
-----
1
(1 row)
```

- **hll\_union(hll, hll)**

Description: Performs the **UNION** operation on two HLL data structures to obtain one HLL.

Return type: hll

For example:

```
select hll_union(hll_add(hll_empty(), hll_hash_integer(1)), hll_add(hll_empty(), hll_hash_integer(2)));
       hll_union
```

```
-----
\x128b7f8895a3f5af28cafeda0ce907e4355b60
(1 row)
```

## Built-in Functions

HLL has a series of built-in functions for internal data processing. Generally, users do not need to know how to use these functions. For details, see [Table 6-7](#).

**Table 6-7** Built-in functions

Function	Description
hll_in	Receives hll data in string format.
hll_out	Sends hll data in string format.
hll_recv	Receives hll data in bytea format.
hll_send	Sends hll data in bytea format.
hll_trans_in	Receives hll_trans_type data in string format.
hll_trans_out	Sends hll_trans_type data in string format.
hll_trans_recv	Receives hll_trans_type data in bytea format.
hll_trans_send	Sends hll_trans_type data in bytea format.
hll_typmod_in	Receives typmod data.
hll_typmod_out	Sends typmod data.
hll_hashval_in	Receives hll_hashval data.
hll_hashval_out	Sends hll_hashval data.
hll_add_trans0	Works similar to hll_add, and is used on the first phase of DNs in distributed aggregation operations.
hll_union_trans	Works similar to hll_union, and is used on the first phase of DNs in distributed aggregation operations.
hll_union_collect	Works similar to hll_union, and is used on the second phase of CNs in distributed aggregation operations to summarize the results of each DN.
hll_pack	Is used on the third phase of CNs in distributed aggregation operations to convert a user-defined type hll_trans_type to the hll type.
hll	Converts a hll type to another hll type. Input parameters can be specified.
hll_hashval	Converts the bigint type to the hll_hashval type.
hll_hashval_int4	Converts the int4 type to the hll_hashval type.

## Operators

- =**

Description: Compares the values of hll and hll\_hashval types to check whether they are the same.

Return type: bool

For example:

```
--hll
select (hll_empty() || hll_hash_integer(1)) = (hll_empty() || hll_hash_integer(1));
column
-----
t
(1 row)

--hll_hashval
select hll_hash_integer(1) = hll_hash_integer(1);
?column?
-----
t
(1 row)
```
- <> or !=**

Description: Compares the values of hll and hll\_hashval types to check whether they are different.

Return type: bool

For example:

```
--hll
select (hll_empty() || hll_hash_integer(1)) <> (hll_empty() || hll_hash_integer(2));
?column?
-----
t
(1 row)

--hll_hashval
select hll_hash_integer(1) <> hll_hash_integer(2);
?column?
-----
t
(1 row)
```
- ||**

Description: Represents the functions of hll\_add, hll\_union, and hll\_add\_rev.

Return type: hll

For example:

```
--hll_add
select hll_empty() || hll_hash_integer(1);
?column?
-----
\x128b7f8895a3f5af28cafe
(1 row)

--hll_add_rev
select hll_hash_integer(1) || hll_empty();
?column?
-----
\x128b7f8895a3f5af28cafe
(1 row)

--hll_union
select (hll_empty() || hll_hash_integer(1)) || (hll_empty() || hll_hash_integer(2));
?column?
```

```
-----
\x128b7f8895a3f5af28cafeda0ce907e4355b60
(1 row)
```

- **#**  
Description: Calculates the number of distinct values of an HLL. It works the same as the `hll_cardinality` function.

Return type: int

For example:

```
select #(hll_empty() || hll_hash_integer(1));
?column?
-----
1
(1 row)
```

## 6.16 SEQUENCE Functions

The sequence functions provide a simple method to ensure security of multiple users for users to obtain sequence values from sequence objects.

- `nextval(regclass)`  
Specifies an increasing sequence and returns a new value.

### NOTE

- To avoid blocking of concurrent transactions that obtain numbers from the same sequence, a `nextval` operation is never rolled back; that is, once a value has been fetched it is considered used, even if the transaction that did the `nextval` later aborts. This means that aborted transactions may leave unused "holes" in the sequence of assigned values. Therefore, sequences in GaussDB(DWS) cannot be used to obtain sequence without gaps.
- If the `nextval` function is pushed to DNs, each DN will automatically connect to the GTM and requests the next value. For example, in the **`insert into t1 select xxx`** statement, a column in table **`t1`** needs to invoke the `nextval` function. If maximum number of connections on the GTM is 8192, this type of pushed statements occupies too many GTM connections. Therefore, the number of concurrent connections for these statements is limited to 7000 divided by the number of cluster DNs. The other 1192 connections are reserved for other statements.

Return type: bigint

The **`nextval`** function can be invoked in either of the following ways: (In example 2, the Oracle syntax is supported. Currently, the sequence name cannot contain a dot.)

Example 1:

```
select nextval('seqDemo');
nextval
-----
2
(1 row)
```

Example 2:

```
select seqDemo.nextval;
nextval
-----
2
(1 row)
```

- `currval(regclass)`

Returns the last value of **nextval** for a specified sequence in the current session. If **nextval** has not been invoked for the specified sequence in the current session, an error is reported when **currval** is invoked. By default, **currval** is disabled. To enable it, set **enable\_beta\_features** to **true**. After **currval** is enabled, **nextval** will not be pushed down.

Return type: bigint

The **currval** function can be invoked in either of the following ways: (In example 2, the Oracle syntax is supported. Currently, the sequence name cannot contain a dot.)

Example 1:

```
select currval('seq1');
currval
-----
      2
(1 row)
```

Example 2:

```
select seq1.currval seq1;
currval
-----
      2
(1 row)
```

- **lastval()**

Returns the last value of **nextval** in the current session. This function is equivalent to **currval**, but **lastval** does not have a parameter. If **nextval** has not been invoked in the current session, an error is reported when **lastval** is invoked.

By default, **lastval** is disabled. To enable it, set **enable\_beta\_features** or **lastval\_supported** to **true**. After **lastval** is enabled, **nextval** will not be pushed down.

Return type: bigint

For example:

```
select lastval();
lastval
-----
      2
(1 row)
```

- **setval(regclass, bigint)**

Sets the current value of a sequence.

Return type: bigint

For example:

```
select setval('seqDemo',1);
setval
-----
      1
(1 row)
```

- **setval(regclass, bigint, Boolean)**

Sets the current value of a sequence and the `is_called` sign.

Return type: bigint

For example:

```
select setval('seqDemo',1,true);
setval
-----
```

```
1  
(1 row)
```

**NOTE**

The current session and GTM will take effect immediately after **setval** is performed. If other sessions have buffered sequence values, **setval** will take effect only after the values are used up. Therefore, to prevent sequence value conflicts, you are advised to use **setval** with caution.

Because the sequence is non-transactional, changes made by **setval** will not be canceled when a transaction rolled back.

## 6.17 Array Functions and Operators

### Array Operators

- =  
Description: Specifies whether two arrays are equal.  
For example:  

```
SELECT ARRAY[1.1,2.1,3.1]::int[] = ARRAY[1,2,3] AS RESULT ;  
result  
-----  
t  
(1 row)
```
- <>  
Description: Specifies whether two arrays are not equal.  
For example:  

```
SELECT ARRAY[1,2,3] <> ARRAY[1,2,4] AS RESULT;  
result  
-----  
t  
(1 row)
```
- <  
Description: Specifies whether an array is less than another.  
For example:  

```
SELECT ARRAY[1,2,3] < ARRAY[1,2,4] AS RESULT;  
result  
-----  
t  
(1 row)
```
- >  
Description: Specifies whether an array is greater than another.  
For example:  

```
SELECT ARRAY[1,4,3] > ARRAY[1,2,4] AS RESULT;  
result  
-----  
t  
(1 row)
```
- <=  
Description: Specifies whether an array is less than another.  
For example:  

```
SELECT ARRAY[1,2,3] <= ARRAY[1,2,3] AS RESULT;  
result
```

- ```
-----
t
(1 row)
```
- **>=**

Description: Specifies whether an array is greater than or equal to another.

For example:

```
SELECT ARRAY[1,4,3] >= ARRAY[1,4,3] AS RESULT;
result
-----
t
(1 row)
```
- **@>**

Description: Specifies whether an array contains another.

For example:

```
SELECT ARRAY[1,4,3] @> ARRAY[3,1] AS RESULT;
result
-----
t
(1 row)
```
- **<@**

Description: Specifies whether an array is contained in another.

For example:

```
SELECT ARRAY[2,7] <@ ARRAY[1,7,4,2,6] AS RESULT;
result
-----
t
(1 row)
```
- **&&**

Description: Specifies whether an array overlaps another (have common elements).

For example:

```
SELECT ARRAY[1,4,3] && ARRAY[2,1] AS RESULT;
result
-----
t
(1 row)
```
- **||**

Description: Array-to-array concatenation

For example:

```
SELECT ARRAY[1,2,3] || ARRAY[4,5,6] AS RESULT;
result
-----
{1,2,3,4,5,6}
(1 row)
SELECT ARRAY[1,2,3] || ARRAY[[4,5,6],[7,8,9]] AS RESULT;
result
-----
{{1,2,3},{4,5,6},{7,8,9}}
(1 row)
```
- **||**

Description: Element-to-array concatenation

For example:

```
SELECT 3 || ARRAY[4,5,6] AS RESULT;
result
```

```
-----
{3,4,5,6}
(1 row)
```

- ||  
Description: Array-to-element concatenation  
For example:  
SELECT ARRAY[4,5,6] || 7 AS RESULT;  
result  
-----  
{4,5,6,7}  
(1 row)

Array comparisons compare the array contents element-by-element, using the default B-tree comparison function for the element data type. In multidimensional arrays, the elements are accessed in row-major order. If the contents of two arrays are equal but the dimensionality is different, the first difference in the dimensionality information determines the sort order.

## Array Functions

- array\_append(anyarray, anyelement)  
Description: Appends an element to the end of an array, and only supports dimension-1 arrays.  
Return type: anyarray  
For example:  
SELECT array\_append(ARRAY[1,2], 3) AS RESULT;  
result  
-----  
{1,2,3}  
(1 row)
- array\_prepend(anyelement, anyarray)  
Description: Appends an element to the beginning of an array, and only supports dimension-1 arrays.  
Return type: anyarray  
For example:  
SELECT array\_prepend(1, ARRAY[2,3]) AS RESULT;  
result  
-----  
{1,2,3}  
(1 row)
- array\_cat(anyarray, anyarray)  
Description: Concatenates two arrays, and supports multi-dimensional arrays.  
Return type: anyarray  
For example:  
SELECT array\_cat(ARRAY[1,2,3], ARRAY[4,5]) AS RESULT;  
result  
-----  
{1,2,3,4,5}  
(1 row)  
SELECT array\_cat(ARRAY[[1,2],[4,5]], ARRAY[6,7]) AS RESULT;  
result  
-----  
{{1,2},{4,5},{6,7}}  
(1 row)



- `array_ndims(anyarray)`  
 Description: Returns the number of dimensions of the array.  
 Return type: int  
 For example:  

```
SELECT array_ndims(ARRAY[[1,2,3], [4,5,6]]) AS RESULT;
result
-----
      2
(1 row)
```
- `array_dims(anyarray)`  
 Description: Returns a text representation of array's dimensions.  
 Return type: text  
 For example:  

```
SELECT array_dims(ARRAY[[1,2,3], [4,5,6]]) AS RESULT;
result
-----
[1:2][1:3]
(1 row)
```
- `array_length(anyarray, int)`  
 Description: Returns the length of the requested array dimension.  
 Return type: int  
 For example:  

```
SELECT array_length(array[1,2,3], 1) AS RESULT;
result
-----
      3
(1 row)
```
- `array_lower(anyarray, int)`  
 Description: Returns lower bound of the requested array dimension.  
 Return type: int  
 For example:  

```
SELECT array_lower('[0:2]={1,2,3}::int[]', 1) AS RESULT;
result
-----
      0
(1 row)
```
- `array_upper(anyarray, int)`  
 Description: Returns upper bound of the requested array dimension.  
 Return type: int  
 For example:  

```
SELECT array_upper(ARRAY[1,8,3,7], 1) AS RESULT;
result
-----
      4
(1 row)
```
- `array_to_string(anyarray, text [, text])`  
 Description: Uses the first **text** as the new delimiter and the second **text** to replace **NULL** values.  
 Return type: text  
 For example:

```
SELECT array_to_string(ARRAY[1, 2, 3, NULL, 5], ',', '**') AS RESULT;
result
-----
1,2,3*,5
(1 row)
```

- `string_to_array(text, text [, text])`

Description: Uses the second **text** as the new delimiter and the third **text** as the substring to be replaced by **NULL** values. A substring can be replaced by **NULL** values only when it is the same as the third **text**.

Return type: `text[]`

For example:

```
SELECT string_to_array('xx~^~yy~^~zz', '~^~', 'yy') AS RESULT;
result
-----
{xx,NULL,zz}
(1 row)
SELECT string_to_array('xx~^~yy~^~zz', '~^~', 'y') AS RESULT;
result
-----
{xx,y,zz}
(1 row)
```

- `unnest(anyarray)`

Description: Expands an array to a set of rows.

Return type: `setof anyelement`

For example:

```
SELECT unnest(ARRAY[1,2]) AS RESULT;
result
-----
1
2
(2 rows)
```

In **string\_to\_array**, if the delimiter parameter is `NULL`, each character in the input string will become a separate element in the resulting array. If the delimiter is an empty string, then the entire input string is returned as a one-element array. Otherwise the input string is split at each occurrence of the delimiter string.

In **string\_to\_array**, if the null-string parameter is omitted or `NULL`, none of the substrings of the input will be replaced by `NULL`.

In **array\_to\_string**, if the null-string parameter is omitted or `NULL`, any null elements in the array are simply skipped and not represented in the output string.

## 6.18 Range Functions and Operators

### Range Operators

- `=`

Description: Equals

For example:

```
SELECT int4range(1,5) = '[1,4]':int4range AS RESULT;
result
-----
t
(1 row)
```

- <>

Description: Does not equal to

For example:

```
SELECT numrange(1.1,2.2) <> numrange(1.1,2.3) AS RESULT;  
result  
-----  
t  
(1 row)
```

- <

Description: Is less than

For example:

```
SELECT int4range(1,10) < int4range(2,3) AS RESULT;  
result  
-----  
t  
(1 row)
```

- >

Description: Is greater than

For example:

```
SELECT int4range(1,10) > int4range(1,5) AS RESULT;  
result  
-----  
t  
(1 row)
```

- <=

Description: Is less than or equals

For example:

```
SELECT numrange(1.1,2.2) <= numrange(1.1,2.2) AS RESULT;  
result  
-----  
t  
(1 row)
```

- >=

Description: Is greater than or equals

For example:

```
SELECT numrange(1.1,2.2) >= numrange(1.1,2.0) AS RESULT;  
result  
-----  
t  
(1 row)
```

- @>

Description: Contains range

For example:

```
SELECT int4range(2,4) @> int4range(2,3) AS RESULT;  
result  
-----  
t  
(1 row)
```

- @>

Description: Contains element

For example:

```
SELECT '[2011-01-01,2011-03-01]':tsrange @> '2011-01-10':timestamp AS RESULT;  
result
```

- ```
-----
t
(1 row)
```
- **<@**

Description: Range is contained by

For example:

```
SELECT int4range(2,4) <@ int4range(1,7) AS RESULT;
result
-----
t
(1 row)
```
  - **<@**

Description: Element is contained by

For example:

```
SELECT 42 <@ int4range(1,7) AS RESULT;
result
-----
f
(1 row)
```
  - **&&**

Description: Overlap (have points in common)

For example:

```
SELECT int8range(3,7) && int8range(4,12) AS RESULT;
result
-----
t
(1 row)
```
  - **<<**

Description: Strictly left of

For example:

```
SELECT int8range(1,10) << int8range(100,110) AS RESULT;
result
-----
t
(1 row)
```
  - **>>**

Description: Strictly right of

For example:

```
SELECT int8range(50,60) >> int8range(20,30) AS RESULT;
result
-----
t
(1 row)
```
  - **&<**

Description: Does not extend to the right of

For example:

```
SELECT int8range(1,20) &< int8range(18,20) AS RESULT;
result
-----
t
(1 row)
```
  - **&>**

Description: Does not extend to the left of

For example:

```
SELECT int8range(7,20) &> int8range(5,10) AS RESULT;
result
-----
t
(1 row)
```

- -|-

Description: Is adjacent to

For example:

```
SELECT numrange(1.1,2.2) -|- numrange(2.2,3.3) AS RESULT;
result
-----
t
(1 row)
```

- +

Description: Union

For example:

```
SELECT numrange(5,15) + numrange(10,20) AS RESULT;
result
-----
[5,20)
(1 row)
```

- \*

Description: Intersection

For example:

```
SELECT int8range(5,15) * int8range(10,20) AS RESULT;
result
-----
[10,15)
(1 row)
```

- -

Description: Difference

For example:

```
SELECT int8range(5,15) - int8range(10,20) AS RESULT;
result
-----
[5,10)
(1 row)
```

The simple comparison operators `<`, `>`, `<=`, and `>=` compare the lower bounds first, and only if those are equal, compare the upper bounds.

The `<<`, `>>`, and `-|-` operators always return false when an empty range is involved; that is, an empty range is not considered to be either before or after any other range.

The union and difference operators will fail if the resulting range would need to contain two disjoint sub-ranges.

## Range Functions

- lower(anyrange)

Description: Lower bound of range

Return type: Range's element type

For example:

```
SELECT lower(numrange(1.1,2.2)) AS RESULT;  
result  
-----  
1.1  
(1 row)
```

- **upper(anyrange)**

Description: Upper bound of range

Return type: Range's element type

For example:

```
SELECT upper(numrange(1.1,2.2)) AS RESULT;  
result  
-----  
2.2  
(1 row)
```

- **isempty(anyrange)**

Description: Is the range empty?

Return type: Boolean

For example:

```
SELECT isempty(numrange(1.1,2.2)) AS RESULT;  
result  
-----  
f  
(1 row)
```

- **lower\_inc(anyrange)**

Description: Is the lower bound inclusive?

Return type: Boolean

For example:

```
SELECT lower_inc(numrange(1.1,2.2)) AS RESULT;  
result  
-----  
t  
(1 row)
```

- **upper\_inc(anyrange)**

Description: Is the upper bound inclusive?

Return type: Boolean

For example:

```
SELECT upper_inc(numrange(1.1,2.2)) AS RESULT;  
result  
-----  
f  
(1 row)
```

- **lower\_inf(anyrange)**

Description: Is the lower bound infinite?

Return type: Boolean

For example:

```
SELECT lower_inf('()'::daterange) AS RESULT;  
result  
-----  
t  
(1 row)
```

- `upper_inf(anyrange)`  
Description: Is the upper bound infinite?  
Return type: Boolean  
For example:

```
SELECT upper_inf('('::daterange) AS RESULT;  
result  
-----  
t  
(1 row)
```

The **lower** and **upper** functions return null if the range is empty or the requested bound is infinite. The **lower\_inc**, **upper\_inc**, **lower\_inf**, and **upper\_inf** functions all return false for an empty range.

## 6.19 Aggregate Functions

### Aggregate Functions

- `sum(expression)`  
Description: Sum of expression across all input values  
Return type:  
Generally, same as the argument data type. In the following cases, type conversion occurs:
  - **BIGINT** for **SMALLINT** or **INT** arguments
  - **NUMBER** for **BIGINT** arguments
  - **DOUBLE PRECISION** for floating-point arguments

For example:

```
SELECT SUM(ss_ext_tax) FROM tpceds.STORE_SALES;  
sum  
-----  
213267594.69  
(1 row)
```

- `max(expression)`  
Description:  
Maximum value of expression across all input values  
Argument types: any array, numeric, string, or date/time type  
Return type: same as the argument type  
For example:  
SELECT MAX(inv\_quantity\_on\_hand) FROM tpceds.inventory;
- `min(expression)`  
Description:  
Minimum value of expression across all input values  
Argument types: any array, numeric, string, or date/time type  
Return type: same as the argument type  
For example:

```
SELECT MIN(inv_quantity_on_hand) FROM tpceds.inventory;  
min  
-----
```

- ```

0
(1 row)

```
- avg(expression)**  
Description: Average (arithmetic mean) of all input values  
Return type:  
**NUMBER** for any integer-type argument.  
**DOUBLE PRECISION** for floating-point parameters.  
otherwise the same as the argument data type.  
For example:  
SELECT AVG(inv\_quantity\_on\_hand) FROM tpcds.inventory;  
avg  
-----  
500.0387129084044604  
(1 row)
  - count(expression)**  
Description: Number of input rows for which the value of expression is not null  
Return type: bigint  
For example:  
SELECT COUNT(inv\_quantity\_on\_hand) FROM tpcds.inventory;  
count  
-----  
11158087  
(1 row)
  - count(\*)**  
Description: Number of input rows  
Return type: bigint  
For example:  
SELECT COUNT(\*) FROM tpcds.inventory;  
count  
-----  
11745000  
(1 row)
  - array\_agg(expression)**  
Description: Input values, including nulls, concatenated into an array  
Return type: array of the argument type  
For example:  
SELECT ARRAY\_AGG(sr\_fee) FROM tpcds.store\_returns WHERE sr\_customer\_sk = 2;  
array\_agg  
-----  
{22.18,63.21}  
(1 row)
  - string\_agg(expression, delimiter)**  
Description: Input values concatenated into a string, separated by delimiter  
Return type: same as the argument type  
For example:  
SELECT string\_agg(sr\_item\_sk, ',') FROM tpcds.store\_returns where sr\_item\_sk < 3;  
string\_agg  
-----  
-----  
1,2,1,2,2,1,1,2,2,1,2,1,2,1,1,1,2,1,1,1,1,2,1,1,1,1,2,2,1,1,1,1,1,1,1,1,2,



```
2,1,1,1,1,1,1,2,2,1,1,2,1,1,1
(1 row)
```

- listagg(expression [, delimiter]) WITHIN GROUP(ORDER BY order-list)  
Description: Aggregation column data sorted according to the mode specified by **WITHIN GROUP**, and concatenated to a string using the specified delimiter
  - **expression**: Mandatory. It specifies an aggregation column name or a column-based, valid expression. It does not support the **DISTINCT** keyword and the **VARIADIC** parameter.
  - **delimiter**: Optional. It specifies a delimiter, which can be a string constant or a deterministic expression based on a group of columns. The default value is empty.
  - **order-list**: Mandatory. It specifies the sorting mode in a group.

Return type: text

 **NOTE**

**listagg** is a column-to-row aggregation function, compatible with Oracle Database 11g Release 2. You can specify the **OVER** clause as a window function. When **listagg** is used as a window function, the **OVER** clause does not support the window sorting or framework of **ORDER BY**, so as to avoid ambiguity in **listagg** and **ORDER BY** of the **WITHIN GROUP** clause.

For example:

The aggregation column is of the text character set type.

```
SELECT deptno, listagg(ename, ',') WITHIN GROUP(ORDER BY ename) AS employees FROM emp
GROUP BY deptno;
deptno |          employees
-----+-----
    10 | CLARK,KING,MILLER
    20 | ADAMS,FORD,JONES,SCOTT,SMITH
    30 | ALLEN,BLAKE,JAMES,MARTIN,TURNER,WARD
(3 rows)
```

The aggregation column is of the integer type.

```
SELECT deptno, listagg(mgrno, ',') WITHIN GROUP(ORDER BY mgrno NULLS FIRST) AS mgrnos FROM
emp GROUP BY deptno;
deptno |          mgrnos
-----+-----
    10 | 7782,7839
    20 | 7566,7566,7788,7839,7902
    30 | 7698,7698,7698,7698,7698,7839
(3 rows)
```

The aggregation column is of the floating point type.

```
SELECT job, listagg(bonus, '($); ') WITHIN GROUP(ORDER BY bonus DESC) || '($)' AS bonus FROM
emp GROUP BY job;
job |          bonus
-----+-----
CLERK | 10234.21($); 2000.80($); 1100.00($); 1000.22($)
PRESIDENT | 23011.88($)
ANALYST | 2002.12($); 1001.01($)
MANAGER | 10000.01($); 2399.50($); 999.10($)
SALESMAN | 1000.01($); 899.00($); 99.99($); 9.00($)
(5 rows)
```

The aggregation column is of the time type.

```
SELECT deptno, listagg(hiredate, ',') WITHIN GROUP(ORDER BY hiredate DESC) AS hiredates FROM
emp GROUP BY deptno;
deptno |          hiredates
-----+-----
```

```
-----
10 | 1982-01-23 00:00:00, 1981-11-17 00:00:00, 1981-06-09 00:00:00
20 | 2001-04-02 00:00:00, 1999-12-17 00:00:00, 1987-05-23 00:00:00, 1987-04-19 00:00:00,
1981-12-03 00:00:00
30 | 2015-02-20 00:00:00, 2010-02-22 00:00:00, 1997-09-28 00:00:00, 1981-12-03 00:00:00,
1981-09-08 00:00:00, 1981-05-01 00:00:00
(3 rows)
```

The aggregation column is of the time interval type.

```
SELECT deptno, listagg(vacationTime, '; ') WITHIN GROUP(ORDER BY vacationTime DESC) AS
vacationTime FROM emp GROUP BY deptno;
deptno |          vacationtime
-----+-----
10 | 1 year 30 days; 40 days; 10 days
20 | 70 days; 36 days; 9 days; 5 days
30 | 1 year 1 mon; 2 mons 10 days; 30 days; 12 days 12:00:00; 4 days 06:00:00; 24:00:00
(3 rows)
```

By default, the delimiter is empty.

```
SELECT deptno, listagg(job) WITHIN GROUP(ORDER BY job) AS jobs FROM emp GROUP BY deptno;
deptno |      jobs
-----+-----
10 | CLERKMANAGERPRESIDENT
20 | ANALYSTANALYSTCLERKCLERKMANAGER
30 | CLERKMANAGERSALESMANSALESMANSALESMANSALESMAN
(3 rows)
```

When **listagg** is used as a window function, the **OVER** clause does not support the window sorting of **ORDER BY**, and the **listagg** column is an ordered aggregation of the corresponding groups.

```
SELECT deptno, mgrno, bonus, listagg(ename, ';' ) WITHIN GROUP(ORDER BY hiredate)
OVER(PARTITION BY deptno) AS employees FROM emp;
deptno | mgrno | bonus |      employees
-----+-----+-----+-----
10 | 7839 | 10000.01 | CLARK; KING; MILLER
10 |      | 23011.88 | CLARK; KING; MILLER
10 | 7782 | 10234.21 | CLARK; KING; MILLER
20 | 7566 | 2002.12 | FORD; SCOTT; ADAMS; SMITH; JONES
20 | 7566 | 1001.01 | FORD; SCOTT; ADAMS; SMITH; JONES
20 | 7788 | 1100.00 | FORD; SCOTT; ADAMS; SMITH; JONES
20 | 7902 | 2000.80 | FORD; SCOTT; ADAMS; SMITH; JONES
20 | 7839 | 999.10 | FORD; SCOTT; ADAMS; SMITH; JONES
30 | 7839 | 2399.50 | BLAKE; TURNER; JAMES; MARTIN; WARD; ALLEN
30 | 7698 | 9.00 | BLAKE; TURNER; JAMES; MARTIN; WARD; ALLEN
30 | 7698 | 1000.22 | BLAKE; TURNER; JAMES; MARTIN; WARD; ALLEN
30 | 7698 | 99.99 | BLAKE; TURNER; JAMES; MARTIN; WARD; ALLEN
30 | 7698 | 1000.01 | BLAKE; TURNER; JAMES; MARTIN; WARD; ALLEN
30 | 7698 | 899.00 | BLAKE; TURNER; JAMES; MARTIN; WARD; ALLEN
(14 rows)
```

- covar\_pop(Y, X)

Description: Overall covariance

Return type: double precision

For example:

```
SELECT COVAR_POP(sr_fee, sr_net_loss) FROM tpcds.store_returns WHERE sr_customer_sk < 1000;
covar_pop
-----
829.749627587403
(1 row)
```

- covar\_samp(Y, X)

Description: Sample covariance

Return type: double precision

For example:

```
SELECT COVAR_SAMP(sr_fee, sr_net_loss) FROM tpccs.store_returns WHERE sr_customer_sk < 1000;
covar_samp
-----
830.052235037289
(1 row)
```

- `stddev_pop(expression)`

Description: Overall standard difference

Return type: **double precision** for floating-point arguments, otherwise **numeric**

For example:

```
SELECT STDDEV_POP(inv_quantity_on_hand) FROM tpccs.inventory WHERE inv_warehouse_sk = 1;
stddev_pop
-----
289.224294957556
(1 row)
```

- `stddev_samp(expression)`

Description: Sample standard deviation of the input values

Return type: **double precision** for floating-point arguments, otherwise **numeric**

For example:

```
SELECT STDDEV_SAMP(inv_quantity_on_hand) FROM tpccs.inventory WHERE inv_warehouse_sk = 1;
stddev_samp
-----
289.224359757315
(1 row)
```

- `var_pop(expression)`

Description: Population variance of the input values (square of the population standard deviation)

Return type: **double precision** for floating-point arguments, otherwise **numeric**

For example:

```
SELECT VAR_POP(inv_quantity_on_hand) FROM tpccs.inventory WHERE inv_warehouse_sk = 1;
var_pop
-----
83650.692793695475
(1 row)
```

- `var_samp(expression)`

Description: Sample variance of the input values (square of the sample standard deviation)

Return type: **double precision** for floating-point arguments, otherwise **numeric**

For example:

```
SELECT VAR_SAMP(inv_quantity_on_hand) FROM tpccs.inventory WHERE inv_warehouse_sk = 1;
var_samp
-----
83650.730277028768
(1 row)
```

- `bit_and(expression)`

Description: The bitwise AND of all non-null input values, or null if none

Return type: same as the argument type

For example:

```
SELECT BIT_AND(inv_quantity_on_hand) FROM tpccs.inventory WHERE inv_warehouse_sk = 1;
bit_and
-----
0
(1 row)
```

- **bit\_or(expression)**

Description: The bitwise OR of all non-null input values, or null if none

Return type: same as the argument type

For example:

```
SELECT BIT_OR(inv_quantity_on_hand) FROM tpccs.inventory WHERE inv_warehouse_sk = 1;
bit_or
-----
1023
(1 row)
```

- **bool\_and(expression)**

Description: Its value is **true** if all input values are **true**, otherwise **false**.

Return type: bool

For example:

```
SELECT bool_and(100 <2500);
bool_and
-----
t
(1 row)
```

- **bool\_or(expression)**

Description: Its value is **true** if at least one input value is **true**, otherwise **false**.

Return type: bool

For example:

```
SELECT bool_or(100 <2500);
bool_or
-----
t
(1 row)
```

- **corr(Y, X)**

Description: Correlation coefficient

Return type: double precision

For example:

```
SELECT CORR(sr_fee, sr_net_loss) FROM tpccs.store_returns WHERE sr_customer_sk < 1000;
corr
-----
.0381383624904186
(1 row)
```

- **every(expression)**

Description: Equivalent to **bool\_and**

Return type: bool

For example:

```
SELECT every(100 <2500);
every
-----
t
(1 row)
```

- **rank(expression)**

Description: The tuples in different groups are sorted non-consecutively by **expression**.

Return type: bigint

For example:

```
SELECT d_moy, d_fy_week_seq, rank() OVER(PARTITION BY d_moy ORDER BY d_fy_week_seq) FROM
tpcds.date_dim WHERE d_moy < 4 AND d_fy_week_seq < 7 ORDER BY 1,2;
```

```
d_moy | d_fy_week_seq | rank
```

| d_moy | d_fy_week_seq | rank |
|-------|---------------|------|
| 1     | 1             | 1    |
| 1     | 1             | 1    |
| 1     | 1             | 1    |
| 1     | 1             | 1    |
| 1     | 1             | 1    |
| 1     | 1             | 1    |
| 1     | 1             | 1    |
| 1     | 2             | 8    |
| 1     | 2             | 8    |
| 1     | 2             | 8    |
| 1     | 2             | 8    |
| 1     | 2             | 8    |
| 1     | 2             | 8    |
| 1     | 2             | 8    |
| 1     | 2             | 8    |
| 1     | 3             | 15   |
| 1     | 3             | 15   |
| 1     | 3             | 15   |
| 1     | 3             | 15   |
| 1     | 3             | 15   |
| 1     | 3             | 15   |
| 1     | 3             | 15   |
| 1     | 3             | 15   |
| 1     | 4             | 22   |
| 1     | 4             | 22   |
| 1     | 4             | 22   |
| 1     | 4             | 22   |
| 1     | 4             | 22   |
| 1     | 4             | 22   |
| 1     | 4             | 22   |
| 1     | 4             | 22   |
| 1     | 5             | 29   |
| 1     | 5             | 29   |
| 2     | 5             | 1    |
| 2     | 5             | 1    |
| 2     | 5             | 1    |
| 2     | 5             | 1    |
| 2     | 5             | 1    |
| 2     | 6             | 6    |
| 2     | 6             | 6    |
| 2     | 6             | 6    |
| 2     | 6             | 6    |
| 2     | 6             | 6    |
| 2     | 6             | 6    |
| 2     | 6             | 6    |
| 2     | 6             | 6    |
| 2     | 6             | 6    |

(42 rows)

- regr\_avgx(Y, X)

Description: Average of the independent variable (**sum(X)/N**)

Return type: double precision

For example:

```
SELECT REGR_AVGX(sr_fee, sr_net_loss) FROM tpcds.store_returns WHERE sr_customer_sk < 1000;
regr_avgx
```

```
-----
578.606576740795
(1 row)
```

- regr\_avgy(Y, X)

Description: Average of the dependent variable (**sum(Y)/N**)

Return type: double precision

For example:

```
SELECT REGR_AVGY(sr_fee, sr_net_loss) FROM tpods.store_returns WHERE sr_customer_sk < 1000;
      regr_avgy
-----
50.0136711629602
(1 row)
```

- `regr_count(Y, X)`

Description: Number of input rows in which both expressions are non-null

Return type: bigint

For example:

```
SELECT REGR_COUNT(sr_fee, sr_net_loss) FROM tpods.store_returns WHERE sr_customer_sk < 1000;
      regr_count
-----
2743
(1 row)
```

- `regr_intercept(Y, X)`

Description: y-intercept of the least-squares-fit linear equation determined by the (X, Y) pairs

Return type: double precision

For example:

```
SELECT REGR_INTERCEPT(sr_fee, sr_net_loss) FROM tpods.store_returns WHERE sr_customer_sk < 1000;
      regr_intercept
-----
49.2040847848607
(1 row)
```

- `regr_r2(Y, X)`

Description: Square of the correlation coefficient

Return type: double precision

For example:

```
SELECT REGR_R2(sr_fee, sr_net_loss) FROM tpods.store_returns WHERE sr_customer_sk < 1000;
      regr_r2
-----
.00145453469345058
(1 row)
```

- `regr_slope(Y, X)`

Description: Slope of the least-squares-fit linear equation determined by the (X, Y) pairs

Return type: double precision

For example:

```
SELECT REGR_SLOPE(sr_fee, sr_net_loss) FROM tpods.store_returns WHERE sr_customer_sk < 1000;
      regr_slope
-----
.00139920009665259
(1 row)
```

- `regr_sxx(Y, X)`

Description:  $\sum(X^2) - \sum(X)^2/N$  (sum of squares of the independent variables)

Return type: double precision

For example:

```
SELECT REGR_SXX(sr_fee, sr_net_loss) FROM tpods.store_returns WHERE sr_customer_sk < 1000;
regr_sxx
-----
1626645991.46135
(1 row)
```

- `regr_sxy(Y, X)`

Description:  $\text{sum}(X*Y) - \text{sum}(X) * \text{sum}(Y)/N$  ("sum of products" of independent times dependent variable)

Return type: double precision

For example:

```
SELECT REGR_SXY(sr_fee, sr_net_loss) FROM tpods.store_returns WHERE sr_customer_sk < 1000;
regr_sxy
-----
2276003.22847225
(1 row)
```

- `regr_syy(Y, X)`

Description:  $\text{sum}(Y^2) - \text{sum}(Y)^2/N$  ("sum of squares" of the dependent variable)

Return type: double precision

For example:

```
SELECT REGR_SYY(sr_fee, sr_net_loss) FROM tpods.store_returns WHERE sr_customer_sk < 1000;
regr_syy
-----
2189417.6547314
(1 row)
```

- `stddev(expression)`

Description: Alias of **stddev\_samp**

Return type: **double precision** for floating-point arguments, otherwise **numeric**

For example:

```
SELECT STDDEV(inv_quantity_on_hand) FROM tpods.inventory WHERE inv_warehouse_sk = 1;
stddev
-----
289.224359757315
(1 row)
```

- `variance(expression, expression)`

Description: Alias of **var\_samp**

Return type: **double precision** for floating-point arguments, otherwise **numeric**

For example:

```
SELECT VARIANCE(inv_quantity_on_hand) FROM tpods.inventory WHERE inv_warehouse_sk = 1;
variance
-----
83650.730277028768
(1 row)
```

- `checksum(expression)`

Description: Returns the CHECKSUM value of all input values. This function can be used to check whether the data in the tables before and after GaussDB(DWS) data restoration or migration is the same. Other databases cannot be checked by using this function. Before and after database backup, database restoration, or data migration, you need to manually run SQL commands to obtain the execution results. Compare the obtained execution

results to check whether the data in the tables before and after the backup or migration is the same.

#### NOTE

- For large tables, the CHECKSUM function may take a long time.
  - If the CHECKSUM values of two tables are different, it indicates that the contents of the two tables are different. Using the hash function in the CHECKSUM function may incur conflicts. There is low possibility that two tables with different contents may have the same CHECKSUM value. The same problem may occur when CHECKSUM is used for columns.
  - If the time type is timestamp, timestamptz, or smalldatetime, ensure that the time zone settings are the same when calculating the CHECKSUM value.
- If the CHECKSUM value of a column is calculated and the column type can be changed to TEXT by default, set *expression* to the column name.
  - If the CHECKSUM value of a column is calculated and the column type cannot be changed to TEXT by default, set *expression* to *Column name::TEXT*.
  - If the CHECKSUM value of all columns is calculated, set *expression* to *Table name::TEXT*.

The following types of data can be converted into TEXT types by default: char, name, int8, int2, int1, int4, raw, pg\_node\_tree, float4, float8, bpchar, varchar, nvarchar2, date, timestamp, timestamptz, numeric, and smalldatetime. Other types need to be forcibly converted to TEXT.

Return type: numeric

For example:

The following shows the CHECKSUM value of a column that can be converted to the TEXT type by default:

```
SELECT CHECKSUM(inv_quantity_on_hand) FROM tpcds.inventory;
checksum
-----
24417258945265247
(1 row)
```

The following shows the CHECKSUM value of a column that cannot be converted to the TEXT type by default: The CHECKSUM parameter is set to *Column name::TEXT*.

```
SELECT CHECKSUM(inv_quantity_on_hand::TEXT) FROM tpcds.inventory;
checksum
-----
24417258945265247
(1 row)
```

The following shows the CHECKSUM value of all columns in a table. Note that the CHECKSUM parameter is set to *Table name::TEXT*. The table name is not modified by its schema.

```
SELECT CHECKSUM(inventory::TEXT) FROM tpcds.inventory;
checksum
-----
25223696246875800
(1 row)
```



## 6.20 Window Functions

Regular aggregate functions return a single value calculated from values in a row, or group all rows into a single output row. Window functions perform a calculation across a set of rows and return a value for each row.

- A window function call represents the application of an aggregate-like function over some portion of the rows selected by a query. Therefore, aggregate functions ([Aggregate Functions](#)) can also be used as window functions. In addition, window functions are able to scan all the rows and divide the query rows into a partition by using the **PARTITION BY** clause.
- Column-store tables support only the window functions **rank (expression)** and **row\_number (expression)** and the aggregate functions **sum**, **count**, **avg**, **min**, and **max**. Row-store tables do not have such restrictions.
- Invoking a window function requires special syntax using the **OVER** clause to specify a window. The **OVER** clause is used for grouping data and sorting the elements in a group. Window functions are used for generating sequence numbers for the values in the group.
- **order by** in a window function must be followed by a column name. If it is followed by a number, the number is processed as a constant value and the target column is not ranked.

### Syntax of a Window Function

```
function_name ([expression [, expression ... ]]) OVER ( window_definition ) function_name ([expression [, expression ... ]]) OVER window_name function_name ( * ) OVER ( window_definition ) function_name ( * ) OVER window_name
```

**window\_definition** is defined as follows:

```
[ existing_window_name ] [ PARTITION BY expression [, ... ] ] [ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ] [, ... ] ] [ frame_clause ]
```

**frame\_clause** is defined as follows:

```
[ RANGE | ROWS ] frame_start [ RANGE | ROWS ] BETWEEN frame_start AND frame_end
```

You can use **RANGE** and **ROWS** to specify the window frame. **ROWS** specifies the window in physical units (rows). **RANGE** specifies the window as a logical offset.

In **RANGE** and **ROWS**, you can use **BETWEEN** *frame\_start* **AND** *frame\_end* to specify the window's first and last rows. If *frame\_end* is left blank, it defaults to **CURRENT ROW**.

The value options of **BETWEEN** *frame\_start* **AND** *frame\_end* are as follows:

- **CURRENT ROW**: The current row is used as the window frame's start or end point.
- *N* **PRECEDING**: The window frame starts from the *n*th row to the current row.
- **UNBOUNDED PRECEDING**: The window frame starts at the first row of the partition.
- *N* **FOLLOWING**: The window frame starts from the current row to the *n*th row.
- **UNBOUNDED FOLLOWING**: The window frame ends with the last row of the partition.

*frame\_start* cannot be **UNBOUNDED FOLLOWING**, *frame\_end* cannot be **UNBOUNDED PRECEDING**, and *frame\_end* cannot be earlier than *frame\_start*. For example, **RANGE BETWEEN CURRENT ROW AND value PRECEDING** is not allowed.

## Window Functions

- RANK()

Description: The **RANK** function is used for generating non-consecutive sequence numbers for the values in each group. The same values have the same sequence number.

Return type: bigint

For example:

```
SELECT d_moy, d_fy_week_seq, rank() OVER(PARTITION BY d_moy ORDER BY d_fy_week_seq) FROM
tpcds.date_dim WHERE d_moy < 4 AND d_fy_week_seq < 7 ORDER BY 1,2;
  d_moy | d_fy_week_seq | rank
```

| d_moy | d_fy_week_seq | rank |
|-------|---------------|------|
| 1     | 1             | 1    |
| 1     | 1             | 1    |
| 1     | 1             | 1    |
| 1     | 1             | 1    |
| 1     | 1             | 1    |
| 1     | 1             | 1    |
| 1     | 1             | 1    |
| 1     | 1             | 1    |
| 1     | 2             | 8    |
| 1     | 2             | 8    |
| 1     | 2             | 8    |
| 1     | 2             | 8    |
| 1     | 2             | 8    |
| 1     | 2             | 8    |
| 1     | 2             | 8    |
| 1     | 2             | 8    |
| 1     | 2             | 8    |
| 1     | 3             | 15   |
| 1     | 3             | 15   |
| 1     | 3             | 15   |
| 1     | 3             | 15   |
| 1     | 3             | 15   |
| 1     | 3             | 15   |
| 1     | 3             | 15   |
| 1     | 3             | 15   |
| 1     | 3             | 15   |
| 1     | 4             | 22   |
| 1     | 4             | 22   |
| 1     | 4             | 22   |
| 1     | 4             | 22   |
| 1     | 4             | 22   |
| 1     | 4             | 22   |
| 1     | 4             | 22   |
| 1     | 4             | 22   |
| 1     | 5             | 29   |
| 1     | 5             | 29   |
| 2     | 5             | 1    |
| 2     | 5             | 1    |
| 2     | 5             | 1    |
| 2     | 5             | 1    |
| 2     | 5             | 1    |
| 2     | 6             | 6    |
| 2     | 6             | 6    |
| 2     | 6             | 6    |
| 2     | 6             | 6    |
| 2     | 6             | 6    |
| 2     | 6             | 6    |
| 2     | 6             | 6    |
| 2     | 6             | 6    |
| 2     | 6             | 6    |

(42 rows)

- ROW\_NUMBER()

Description: The **ROW\_NUMBER** function is used for generating consecutive sequence numbers for the values in each group. The same values have different sequence numbers.

Return type: bigint

For example:

```
SELECT d_moy, d_fy_week_seq, Row_number() OVER(PARTITION BY d_moy ORDER BY
d_fy_week_seq) FROM tpcds.date_dim WHERE d_moy < 4 AND d_fy_week_seq < 7 ORDER BY 1,2;
d_moy | d_fy_week_seq | row_number
```

| d_moy | d_fy_week_seq | row_number |
|-------|---------------|------------|
| 1     | 1             | 1          |
| 1     | 1             | 2          |
| 1     | 1             | 3          |
| 1     | 1             | 4          |
| 1     | 1             | 5          |
| 1     | 1             | 6          |
| 1     | 1             | 7          |
| 1     | 2             | 8          |
| 1     | 2             | 9          |
| 1     | 2             | 10         |
| 1     | 2             | 11         |
| 1     | 2             | 12         |
| 1     | 2             | 13         |
| 1     | 2             | 14         |
| 1     | 3             | 15         |
| 1     | 3             | 16         |
| 1     | 3             | 17         |
| 1     | 3             | 18         |
| 1     | 3             | 19         |
| 1     | 3             | 20         |
| 1     | 3             | 21         |
| 1     | 4             | 22         |
| 1     | 4             | 23         |
| 1     | 4             | 24         |
| 1     | 4             | 25         |
| 1     | 4             | 26         |
| 1     | 4             | 27         |
| 1     | 4             | 28         |
| 1     | 5             | 29         |
| 1     | 5             | 30         |
| 2     | 5             | 1          |
| 2     | 5             | 2          |
| 2     | 5             | 3          |
| 2     | 5             | 4          |
| 2     | 5             | 5          |
| 2     | 6             | 6          |
| 2     | 6             | 7          |
| 2     | 6             | 8          |
| 2     | 6             | 9          |
| 2     | 6             | 10         |
| 2     | 6             | 11         |
| 2     | 6             | 12         |

(42 rows)

- **DENSE\_RANK()**

Description: The **DENSE\_RANK** function is used for generating consecutive sequence numbers for the values in each group. The same values have the same sequence number.

Return type: bigint

For example:

```
SELECT d_moy, d_fy_week_seq, dense_rank() OVER(PARTITION BY d_moy ORDER BY d_fy_week_seq)
FROM tpcds.date_dim WHERE d_moy < 4 AND d_fy_week_seq < 7 ORDER BY 1,2;
d_moy | d_fy_week_seq | dense_rank
```

| d_moy | d_fy_week_seq | dense_rank |
|-------|---------------|------------|
| 1     | 1             | 1          |

|   |   |   |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 1 | 1 |
| 1 | 1 | 1 |
| 1 | 1 | 1 |
| 1 | 1 | 1 |
| 1 | 1 | 1 |
| 1 | 2 | 2 |
| 1 | 2 | 2 |
| 1 | 2 | 2 |
| 1 | 2 | 2 |
| 1 | 2 | 2 |
| 1 | 2 | 2 |
| 1 | 2 | 2 |
| 1 | 2 | 2 |
| 1 | 3 | 3 |
| 1 | 3 | 3 |
| 1 | 3 | 3 |
| 1 | 3 | 3 |
| 1 | 3 | 3 |
| 1 | 3 | 3 |
| 1 | 3 | 3 |
| 1 | 3 | 3 |
| 1 | 4 | 4 |
| 1 | 4 | 4 |
| 1 | 4 | 4 |
| 1 | 4 | 4 |
| 1 | 4 | 4 |
| 1 | 4 | 4 |
| 1 | 4 | 4 |
| 1 | 4 | 4 |
| 1 | 5 | 5 |
| 1 | 5 | 5 |
| 2 | 5 | 1 |
| 2 | 5 | 1 |
| 2 | 5 | 1 |
| 2 | 5 | 1 |
| 2 | 5 | 1 |
| 2 | 6 | 2 |
| 2 | 6 | 2 |
| 2 | 6 | 2 |
| 2 | 6 | 2 |
| 2 | 6 | 2 |
| 2 | 6 | 2 |
| 2 | 6 | 2 |
| 2 | 6 | 2 |
| 2 | 6 | 2 |

(42 rows)

- PERCENT\_RANK()

Description: The **PERCENT\_RANK** function is used for generating corresponding sequence numbers for the values in each group. That is, the function calculates the value according to the formula  $\text{Sequence number} = (\text{Rank} - 1) / (\text{Total rows} - 1)$ . **Rank** is the corresponding sequence number generated based on the **RANK** function for the value and **Total rows** is the total number of elements in a group.

Return type: double precision

For example:

```
SELECT d_moy, d_fy_week_seq, percent_rank() OVER(PARTITION BY d_moy ORDER BY d_fy_week_seq)
FROM tpods.date_dim WHERE d_moy < 4 AND d_fy_week_seq < 7 ORDER BY 1,2;
d_moy | d_fy_week_seq | percent_rank
```

| d_moy | d_fy_week_seq | percent_rank     |
|-------|---------------|------------------|
| 1     | 1             | 0                |
| 1     | 1             | 0                |
| 1     | 1             | 0                |
| 1     | 1             | 0                |
| 1     | 1             | 0                |
| 1     | 1             | 0                |
| 1     | 1             | 0                |
| 1     | 1             | 0                |
| 1     | 1             | 0                |
| 1     | 2             | .241379310344828 |
| 1     | 2             | .241379310344828 |
| 1     | 2             | .241379310344828 |

```

1 |      2 | .241379310344828
1 |      2 | .241379310344828
1 |      2 | .241379310344828
1 |      2 | .241379310344828
1 |      3 | .482758620689655
1 |      3 | .482758620689655
1 |      3 | .482758620689655
1 |      3 | .482758620689655
1 |      3 | .482758620689655
1 |      3 | .482758620689655
1 |      3 | .482758620689655
1 |      3 | .482758620689655
1 |      3 | .482758620689655
1 |      4 | .724137931034483
1 |      4 | .724137931034483
1 |      4 | .724137931034483
1 |      4 | .724137931034483
1 |      4 | .724137931034483
1 |      4 | .724137931034483
1 |      4 | .724137931034483
1 |      4 | .724137931034483
1 |      5 | .96551724137931
1 |      5 | .96551724137931
2 |      5 | 0
2 |      5 | 0
2 |      5 | 0
2 |      5 | 0
2 |      5 | 0
2 |      6 | .454545454545455
2 |      6 | .454545454545455
2 |      6 | .454545454545455
2 |      6 | .454545454545455
2 |      6 | .454545454545455
2 |      6 | .454545454545455
2 |      6 | .454545454545455
2 |      6 | .454545454545455

```

(42 rows)

- **CUME\_DIST()**

Description: The **CUME\_DIST** function is used for generating accumulative distribution sequence numbers for the values in each group. That is, the function calculates the value according to the following formula: Sequence number = Number of rows preceding or peer with current row/Total rows.

Return type: double precision

For example:

```

SELECT d_moy, d_fy_week_seq, cume_dist() OVER(PARTITION BY d_moy ORDER BY d_fy_week_seq)
FROM tpods.date_dim e_dim WHERE d_moy < 4 AND d_fy_week_seq < 7 ORDER BY 1,2;

```

```

d_moy | d_fy_week_seq | cume_dist
-----+-----+-----
1 |      1 | .233333333333333
1 |      1 | .233333333333333
1 |      1 | .233333333333333
1 |      1 | .233333333333333
1 |      1 | .233333333333333
1 |      1 | .233333333333333
1 |      1 | .233333333333333
1 |      2 | .466666666666667
1 |      2 | .466666666666667
1 |      2 | .466666666666667
1 |      2 | .466666666666667
1 |      2 | .466666666666667
1 |      2 | .466666666666667
1 |      2 | .466666666666667
1 |      2 | .466666666666667
1 |      3 | .7
1 |      3 | .7
1 |      3 | .7
1 |      3 | .7
1 |      3 | .7
1 |      3 | .7
1 |      3 | .7
1 |      4 | .933333333333333

```

```

1 |      4 | .933333333333333
1 |      4 | .933333333333333
1 |      4 | .933333333333333
1 |      4 | .933333333333333
1 |      4 | .933333333333333
1 |      4 | .933333333333333
1 |      5 |      1
1 |      5 |      1
2 |      5 | .416666666666667
2 |      5 | .416666666666667
2 |      5 | .416666666666667
2 |      5 | .416666666666667
2 |      5 | .416666666666667
2 |      6 |      1
2 |      6 |      1
2 |      6 |      1
2 |      6 |      1
2 |      6 |      1
2 |      6 |      1
2 |      6 |      1
2 |      6 |      1

```

(42 rows)

- NTILE(num\_buckets integer)

Description: The **NTILE** function is used for equally allocating sequential data sets to the buckets whose quantity is specified by **num\_buckets** according to **num\_buckets integer** and allocating the bucket number to each row. Divide the partition as equally as possible.

Return type: integer

For example:

```

SELECT d_moy, d_fy_week_seq, ntile(3) OVER(PARTITION BY d_moy ORDER BY d_fy_week_seq) FROM
tpcds.date_dim WHERE d_moy < 4 AND d_fy_week_seq < 7 ORDER BY 1,2;
d_moy | d_fy_week_seq | ntile

```

```

-----+-----+-----
1 |      1 |      1
1 |      1 |      1
1 |      1 |      1
1 |      1 |      1
1 |      1 |      1
1 |      1 |      1
1 |      1 |      1
1 |      1 |      1
1 |      2 |      1
1 |      2 |      1
1 |      2 |      1
1 |      2 |      2
1 |      2 |      2
1 |      2 |      2
1 |      2 |      2
1 |      3 |      2
1 |      3 |      2
1 |      3 |      2
1 |      3 |      2
1 |      3 |      2
1 |      3 |      2
1 |      3 |      2
1 |      3 |      3
1 |      4 |      3
1 |      4 |      3
1 |      4 |      3
1 |      4 |      3
1 |      4 |      3
1 |      4 |      3
1 |      4 |      3
1 |      4 |      3
1 |      5 |      3
1 |      5 |      3
2 |      5 |      1
2 |      5 |      1
2 |      5 |      1
2 |      5 |      1

```

```

2 |      5 | 2
2 |      6 | 2
2 |      6 | 2
2 |      6 | 2
2 |      6 | 3
2 |      6 | 3
2 |      6 | 3
2 |      6 | 3
2 |      6 | 3
(42 rows)

```

- LAG(value any [, offset integer [, default any ]])

Description: The **LAG** function is used for generating lag values for the corresponding values in each group. That is, the value of the row obtained by moving forward the row corresponding to the current value by **offset** (integer) is the sequence number. If the row does not exist after the moving, the result value is the default value. If omitted, **offset** defaults to **1** and **default** to **null**.

Return type: same as the parameter type

For example:

```

SELECT d_moy, d_fy_week_seq, lag(d_moy,3,null) OVER(PARTITION BY d_moy ORDER BY
d_fy_week_seq) FROM tpcds.date_dim WHERE d_moy < 4 AND d_fy_week_seq < 7 ORDER BY 1,2;
d_moy | d_fy_week_seq | lag
-----+-----+-----

```

```

1 |      1 | 1
1 |      1 | 1
1 |      1 | 1
1 |      1 | 1
1 |      1 | 1
1 |      1 | 1
1 |      1 | 1
1 |      2 | 1
1 |      2 | 1
1 |      2 | 1
1 |      2 | 1
1 |      2 | 1
1 |      2 | 1
1 |      2 | 1
1 |      2 | 1
1 |      2 | 1
1 |      2 | 1
1 |      3 | 1
1 |      3 | 1
1 |      3 | 1
1 |      3 | 1
1 |      3 | 1
1 |      3 | 1
1 |      3 | 1
1 |      3 | 1
1 |      3 | 1
1 |      3 | 1
1 |      4 | 1
1 |      4 | 1
1 |      4 | 1
1 |      4 | 1
1 |      4 | 1
1 |      4 | 1
1 |      4 | 1
1 |      4 | 1
1 |      4 | 1
1 |      4 | 1
1 |      5 | 1
1 |      5 | 1
2 |      5 |
2 |      5 |
2 |      5 |
2 |      5 | 2
2 |      5 | 2
2 |      6 | 2
2 |      6 | 2
2 |      6 | 2
2 |      6 | 2
2 |      6 | 2
2 |      6 | 2
2 |      6 | 2
2 |      6 | 2
2 |      6 | 2
(42 rows)

```

- LEAD(value any [, offset integer [, default any ]])

Description: The **LEAD** function is used for generating leading values for the corresponding values in each group. That is, the value of the row obtained by moving backward the row corresponding to the current value by **offset** (integer) is the sequence number. If the number of rows after the moving exceeds the total number for the current group, the result value is the default value. If omitted, **offset** defaults to **1** and **default** to **null**.

Return type: same as the parameter type

For example:

```
SELECT d_moy, d_fy_week_seq, lead(d_fy_week_seq,2) OVER(PARTITION BY d_moy ORDER BY
d_fy_week_seq) FROM tpods.date_dim WHERE d_moy < 4 AND d_fy_week_seq < 7 ORDER BY
1,2;          d_moy | d_fy_week_seq | lead
```

|   |   |   |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 1 | 1 |
| 1 | 1 | 1 |
| 1 | 1 | 1 |
| 1 | 1 | 1 |
| 1 | 1 | 2 |
| 1 | 1 | 2 |
| 1 | 2 | 2 |
| 1 | 2 | 2 |
| 1 | 2 | 2 |
| 1 | 2 | 2 |
| 1 | 2 | 2 |
| 1 | 2 | 3 |
| 1 | 2 | 3 |
| 1 | 3 | 3 |
| 1 | 3 | 3 |
| 1 | 3 | 3 |
| 1 | 3 | 3 |
| 1 | 3 | 4 |
| 1 | 3 | 4 |
| 1 | 4 | 4 |
| 1 | 4 | 4 |
| 1 | 4 | 4 |
| 1 | 4 | 4 |
| 1 | 4 | 4 |
| 1 | 4 | 5 |
| 1 | 4 | 5 |
| 1 | 5 |   |
| 1 | 5 |   |
| 2 | 5 | 5 |
| 2 | 5 | 5 |
| 2 | 5 | 5 |
| 2 | 5 | 6 |
| 2 | 5 | 6 |
| 2 | 6 | 6 |
| 2 | 6 | 6 |
| 2 | 6 | 6 |
| 2 | 6 | 6 |
| 2 | 6 | 6 |
| 2 | 6 | 6 |
| 2 | 6 | 6 |
| 2 | 6 | 6 |
| 2 | 6 | 6 |
| 2 | 6 | 6 |

(42 rows)

- FIRST\_VALUE(value any)

Description: The **FIRST\_VALUE** function is used for returning the first value of each group.

Return type: same as the parameter type

For example:





```

1 |      2 |      1
1 |      2 |      1
1 |      2 |      1
1 |      3 |      1
1 |      3 |      1
1 |      3 |      1
1 |      3 |      1
1 |      3 |      1
1 |      3 |      1
1 |      3 |      1
1 |      3 |      1
1 |      4 |      1
1 |      4 |      1
1 |      4 |      1
1 |      4 |      1
1 |      4 |      1
1 |      4 |      1
1 |      4 |      1
1 |      4 |      1
1 |      4 |      1
1 |      5 |      1
1 |      5 |      1
2 |      5 |      2
2 |      5 |      2
2 |      5 |      2
2 |      5 |      2
2 |      5 |      2
(35 rows)

```

- **NTH\_VALUE**(value any, nth integer)

Description: The *n*th row for a group is the returned value. If the row does not exist, **NULL** is returned by default.

Return type: same as the parameter type

For example:

```

SELECT d_moy, d_fy_week_seq, nth_value(d_fy_week_seq,6) OVER(PARTITION BY d_moy ORDER BY
d_fy_week_seq) FROM tpcds.date_dim WHERE d_moy < 4 AND d_fy_week_seq < 6 ORDER BY 1,2;
d_moy | d_fy_week_seq | nth_value

```

```

-----+-----+-----
1 |      1 |      1
1 |      1 |      1
1 |      1 |      1
1 |      1 |      1
1 |      1 |      1
1 |      1 |      1
1 |      1 |      1
1 |      1 |      1
1 |      2 |      1
1 |      2 |      1
1 |      2 |      1
1 |      2 |      1
1 |      2 |      1
1 |      2 |      1
1 |      2 |      1
1 |      2 |      1
1 |      2 |      1
1 |      2 |      1
1 |      2 |      1
1 |      3 |      1
1 |      3 |      1
1 |      3 |      1
1 |      3 |      1
1 |      3 |      1
1 |      3 |      1
1 |      3 |      1
1 |      3 |      1
1 |      3 |      1
1 |      3 |      1
1 |      4 |      1
1 |      4 |      1
1 |      4 |      1
1 |      4 |      1
1 |      4 |      1
1 |      4 |      1
1 |      4 |      1
1 |      4 |      1
1 |      4 |      1
1 |      4 |      1
1 |      5 |      1
1 |      5 |      1
2 |      5 |
2 |      5 |

```

```

2 |      5 |
2 |      5 |
2 |      5 |
(35 rows)

```

## 6.21 Security Functions

### Security Functions

- `gs_encrypt_aes128(encryptstr,keystr)`

Description: Encrypts **encryptstr** strings using **keystr** as the key and returns encrypted strings. The length of **keystr** ranges from 1 to 16 bytes. Currently, the following types of data can be encrypted: numerals supported in the database; character type; RAW in binary type; and DATE, TIMESTAMP, and SMALLDATETIME in date/time type.

Return type: text

Length of the return value: At least 92 bytes and no more than  $(4*[Len/3]+68)$  bytes, where *Len* indicates the length of the data before encryption (unit: byte).

For example:

```
SELECT gs_encrypt_aes128('MPPDB','1234');
```

```

          gs_encrypt_aes128
-----
gwditQLQG8NhFw4OuoKhhQJoXojhFLYkjeG0aYdSCtLCnIUgkNwwYI04KbuhmcGZp8jWizBdR1vU9Cspjuzl
0lbz12A=
(1 row)

```

#### NOTE

A decryption password is required during the execution of this function. For security purposes, the **gsql** tool does not record the function in the execution history. That is, the execution history of this function cannot be found in **gsql** by paging up and down.

- `gs_decrypt_aes128(decryptstr,keystr)`

Description: Decrypts **decrypt** strings using **keystr** as the key and returns decrypted strings. The **keystr** used for decryption must be consistent with that used for encryption. **keystr** cannot be empty.

#### NOTE

This parameter needs to be used with the **gs\_encrypt\_aes128** encryption function.

Return type: text

For example:

```

SELECT
gs_decrypt_aes128('gwditQLQG8NhFw4OuoKhhQJoXojhFLYkjeG0aYdSCtLCnIUgkNwwYI04KbuhmcGZp8j
WizBdR1vU9Cspjuzl0lbz12A=','1234');
          gs_decrypt_aes128
-----
MPPDB
(1 row)

```

#### NOTE

A decryption password is required during the execution of this function. For security purposes, the **gsql** tool does not record the function in the execution history. That is, the execution history of this function cannot be found in **gsql** by paging up and down.

- **gs\_password\_deadline**  
Description: Indicates the number of remaining days before the password of the current user expires.

Return type: interval

For example:

```
SELECT gs_password_deadline();
gs_password_deadline
-----
83 days 17:44:32.196094
(1 row)
```

- **login\_audit\_messages**  
Description: Queries login information about a login user.

Return type: tuple

For example:

- Checks the date, time, and IP address successfully authenticated during the last login.

```
SELECT * FROM login_audit_messages(true);
username | database | logintime | type | result | client_conninfo
-----+-----+-----+-----+-----+-----
dbadmin | postgres | 2017-06-02 15:28:34+08 | login_success | ok | gsq|[local]
(1 row)
```

- Checks the date, time, and IP address that failed to be authenticated during the last login.

```
SELECT * FROM login_audit_messages(false) ORDER BY logintime desc limit 1;
username | database | logintime | type | result | client_conninfo
-----+-----+-----+-----+-----+-----
(0 rows)
```

- Checks the number of failed attempts, date, and time since the previous successful authentication.

```
SELECT * FROM login_audit_messages(false);
username | database | logintime | type | result | client_conninfo
-----+-----+-----+-----+-----+-----
(0 rows)
```

- **login\_audit\_messages\_pid**  
Description: Queries login information about a login user. Different from **login\_audit\_messages**, this function queries login information based on **backendid**. Information about subsequent logins of the same user does not alter the query result of previous logins and cannot be found using this function.

Return type: tuple

For example:

- Checks the date, time, and IP address successfully authenticated during the last login.

```
SELECT * FROM login_audit_messages(true);
username | database | logintime | type | result | client_conninfo | backendid
-----+-----+-----+-----+-----+-----+-----
dbadmin | postgres | 2017-06-02 15:28:34+08 | login_success | ok | gsq|[local] | 140311900702464
(1 row)
```

- Checks the date, time, and IP address that failed to be authenticated during the last login.

```
SELECT * FROM login_audit_messages(false) ORDER BY logintime desc limit 1;
username | database | logintime | type | result | client_conninfo | backendid
-----+-----+-----+-----+-----+-----+-----
(0 rows)
```

- Checks the number of failed attempts, date, and time since the previous successful authentication.

```
SELECT * FROM login_audit_messages(false);
username | database | logintime | type | result | client_conninfo | backendid
-----+-----+-----+-----+-----+-----+-----
(0 rows)
```

- **inet\_server\_addr**

Description: Displays the server IP address.

Return type: inet

For example:

```
SELECT inet_server_addr();
inet_server_addr
-----
10.10.0.13
(1 row)
```

 **NOTE**

- The client IP address 10.10.0.50 and server IP address 10.10.0.13 are used as an example.
  - If the database is connected to the local PC, the value is empty.
- **inet\_client\_addr**

Description: Displays the client IP address.

Return type: inet

For example:

```
SELECT inet_client_addr();
inet_client_addr
-----
10.10.0.50
(1 row)
```

 **NOTE**

- The client IP address 10.10.0.50 and server IP address 10.10.0.13 are used as an example.
  - If the database is connected to the local PC, the value is empty.
- **pg\_query\_audit**

Description: Displays audit logs of the CN.

Return type: record

The following table describes return fields.

| Name     | Type                     | Description                                 |
|----------|--------------------------|---------------------------------------------|
| time     | timestamp with time zone | Operation time                              |
| type     | text                     | Operation type                              |
| result   | text                     | Operation results                           |
| username | text                     | Name of the user who performs the operation |
| database | text                     | Database name                               |

| Name            | Type | Description                   |
|-----------------|------|-------------------------------|
| client_conninfo | text | Client connection information |
| object_name     | text | Object name                   |
| detail_info     | text | Operation details             |
| node_name       | text | Node name                     |
| thread_id       | text | Thread ID                     |
| local_port      | text | Local port                    |
| remote_port     | text | Remote port                   |

For details about how to use the function and details about function examples, see section "Querying Audit Results."

- pgxc\_query\_audit  
Description: Displays audit logs of all CNs.  
Return type: record  
The return fields of this function are the same as those of the **pg\_query\_audit** function.  
For details about how to use the function and details about function examples, see section "Querying Audit Results."
- pg\_delete\_audit  
Description: Deletes audit logs in a specified period. Return type: void

## 6.22 Set Returning Functions

### Series Generating Functions

- generate\_series(start, stop)  
Description: Generates a series of values, from **start** to **stop** with a step size of one.  
Parameter type: int, bigint, or numeric  
Return type: setof int, setof bigint, or setof numeric (same as the argument type)
- generate\_series(start, stop, step)  
Description: Generates a series of values, from **start** to **stop** with a step size of **step**.  
Parameter type: int, bigint, or numeric  
Return type: setof int, setof bigint, or setof numeric (same as the argument type)
- generate\_series(start, stop, step interval)  
Description: Generates a series of values, from **start** to **stop** with a step size of **step**.

Parameter type: timestamp or timestamp with time zone

Return type: setof timestamp or setof timestamp with time zone (same as argument type)

When **step** is positive, zero rows are returned if **start** is greater than **stop**. Conversely, when **step** is negative, zero rows are returned if **start** is less than **stop**. Zero rows are also returned for **NULL** inputs. It is an error for **step** to be zero.

For example:

```
SELECT * FROM generate_series(2,4);
generate_series
-----
         2
         3
         4
(3 rows)

SELECT * FROM generate_series(5,1,-2);
generate_series
-----
         5
         3
         1
(3 rows)

SELECT * FROM generate_series(4,3);
generate_series
-----
(0 rows)

-- this example relies on the date-plus-integer operator
SELECT current_date + s.a AS dates FROM generate_series(0,14,7) AS s(a);
dates
-----
2017-06-02
2017-06-09
2017-06-16
(3 rows)

SELECT * FROM generate_series('2008-03-01 00:00'::timestamp, '2008-03-04 12:00', '10 hours');
generate_series
-----
2008-03-01 00:00:00
2008-03-01 10:00:00
2008-03-01 20:00:00
2008-03-02 06:00:00
2008-03-02 16:00:00
2008-03-03 02:00:00
2008-03-03 12:00:00
2008-03-03 22:00:00
2008-03-04 08:00:00
(9 rows)
```

## Subscript Generating Functions

- `generate_subscripts(array anyarray, dim int)`  
 Description: Generates a series comprising the given array's subscripts.  
 Return type: setof int
- `generate_subscripts(array anyarray, dim int, reverse boolean)`  
 Description: Generates a series comprising the given array's subscripts. When **reverse** is true, the series is returned in reverse order.  
 Return type: setof int

**generate\_subscripts** is a function that generates the set of valid subscripts for the specified dimension of the given array. Zero rows are returned for arrays that do not have the requested dimension, or for NULL arrays (but valid subscripts are returned for NULL array elements). For example:

```
-- basic usage
SELECT generate_subscripts('{NULL,1,NULL,2}'::int[], 1) AS s;
s
---
1
2
3
4
(4 rows)
-- unnest a 2D array
CREATE OR REPLACE FUNCTION unnest2(anyarray)
RETURNS SETOF anyelement AS $$
SELECT $1[i][j]
  FROM generate_subscripts($1,1) g1(i),
       generate_subscripts($1,2) g2(j);
$$ LANGUAGE sql IMMUTABLE;

SELECT * FROM unnest2(ARRAY[[1,2],[3,4]]);
unnest2
-----
 1
 2
 3
 4
(4 rows)

-- Delete the function:
DROP FUNCTION unnest2;
```

## 6.23 Conditional Expression Functions

### Conditional Expression Functions

- `coalesce(expr1, expr2, ..., exprn)`

Description:

Returns the first of its arguments that is not null.

**COALESCE(expr1, expr2)** is equivalent to **CASE WHEN expr1 IS NOT NULL THEN expr1 ELSE expr2 END**.

For example:

```
SELECT coalesce(NULL,'hello');
coalesce
-----
hello
(1 row)
```

Note:

- Null is returned only if all parameters are null.
  - This value is replaced by the default value when data is displayed.
  - Like a CASE expression, COALESCE only evaluates the parameters that are needed to determine the result. That is, parameters to the right of the first non-null parameter are not evaluated.
- `decode(base_expr, compare1, value1, Compare2,value2, ... default)`



Description: Compares `base_expr` with each `compare(n)` and returns `value(n)` if they are matched. If `base_expr` does not match each **`compare(n)`**, the default value is returned.

For example:

```
SELECT decode('A','A',1,'B',2,0);
-----
1
(1 row)
```

- `nullif(expr1, expr2)`

Description:

Returns NULL if `expr1` and `expr2` are equal. Else, it returns `expr1`.

**`nullif(expr1, expr2)`** is equivalent to **`CASE WHEN expr1 = expr2 THEN NULL ELSE expr1 END`**.

For example:

```
SELECT nullif('hello','world');
-----
hello
(1 row)
```

Note:

Assume the two parameter data types are different:

- If implicit conversion exists between the two data types, implicitly convert the parameter of lower priority to this data type using the data type of higher priority. If the conversion succeeds, computation is performed. Otherwise, an error is returned. For example:

```
SELECT nullif('1234'::VARCHAR,123::INT4);
-----
1234
(1 row)
SELECT nullif('1234'::VARCHAR,'2012-12-24'::DATE);
ERROR:  invalid input syntax for type timestamp: "1234"
```

- If implicit conversion is not applied between two data types, an error is displayed. For example:

```
SELECT nullif(TRUE::BOOLEAN,'2012-12-24'::DATE);
ERROR:  operator does not exist: boolean = timestamp without time zone
LINE 1: SELECT nullif(TRUE::BOOLEAN,'2012-12-24'::DATE) FROM DUAL;
      ^
HINT:  No operator matches the given name and argument type(s). You might need to add explicit type casts.
```

- `nvl( expr1 , expr2 )`

Description:

- If **`expr1`** is NULL, **`expr2`** is returned.
- If **`expr1`** is not NULL, **`expr1`** is returned.

For example:

```
SELECT nvl('hello','world');
-----
hello
(1 row)
```

Parameters `expr1` and `expr2` can be of any data type. If `expr1` and `expr2` are of different data types, NVL checks whether `expr2` can be implicitly converted to `expr1`. If it can, the `expr1` data type is returned. If `expr2` cannot be implicitly

converted to expr1 but epr1 can be implicitly converted to expr2, the expr2 data type is returned. If no implicit type conversion exists between the two parameters and the parameters are different data types, an error is reported.

- `sys_context( 'namespace' , 'parameter')`

Description: Obtains and returns the parameter values of a specified **namespace**.

Return type: VARCHAR

For example:

```
SELECT sys_context('USERENV', 'CURRENT_SCHEMA');
sys_context
-----
public
(1 row)
```

The result varies according to the current actual schema.

Note: Currently, only the following formats are supported: `SYS_CONTEXT('USERENV', 'CURRENT_SCHEMA')` and `SYS_CONTEXT('USERENV', 'CURRENT_USER')`.

- `greatest(expr1 [, ...])`

Description: Selects the largest value from a list of any number of expressions.

Return type:

For example:

```
SELECT greatest(1*2,2-3,4-1);
greatest
-----
      3
(1 row)
SELECT greatest('HARRY', 'HARRIOT', 'HAROLD');
greatest
-----
HARRY
(1 row)
```

- `least(expr1 [, ...])`

Description: Selects the smallest value from a list of any number of expressions.

For example:

```
SELECT least(1*2,2-3,4-1);
least
-----
     -1
(1 row)
SELECT least('HARRY','HARRIOT','HAROLD');
least
-----
HAROLD
(1 row)
```

- `EMPTY_BLOB()`

Description: Initiates a BLOB variable in an INSERT or an UPDATE statement to a NULL value.

Return type: BLOB

For example:

```
-- Create a table:
CREATE TABLE blob_tb(b blob,id int) DISTRIBUTE BY REPLICATION;
-- Insert data:
```

```
INSERT INTO date_type_blob VALUES (empty_blob(),1);  
--Delete the table.  
DROP TAB blob_tb;
```

Note: The length is 0 obtained using **DBMS.GETLENGTH** in a parallel mode.

## 6.24 System Information Functions

### Session Information Functions

- **current\_catalog**  
Description: Name of the current database (called "catalog" in the SQL standard)  
Return type: name  
For example:  

```
SELECT current_catalog;  
current_database  
-----  
postgres  
(1 row)
```
- **current\_database()**  
Description: Name of the current database  
Return type: name  
For example:  

```
SELECT current_database();  
current_database  
-----  
postgres  
(1 row)
```
- **current\_query()**  
Description: Text of the currently executing query, as submitted by the client (might contain more than one statement)  
Return type: text  
For example:  

```
SELECT current_query();  
current_query  
-----  
SELECT current_query();  
(1 row)
```
- **current\_schema[()]**  
Description: Name of current schema  
Return type: name  
For example:  

```
SELECT current_schema();  
current_schema  
-----  
public  
(1 row)
```

Remarks: **current\_schema** returns the first valid schema name in the search path. (If the search path is empty or contains no valid schema name, **NULL** is returned.) This is the schema that will be used for any tables or other named objects that are created without specifying a target schema.

- `current_schemas(Boolean)`

Description: Names of schemas in search path

Return type: `name[]`

For example:

```
SELECT current_schemas(true);
 current_schemas
-----
 {pg_catalog,public}
(1 row)
```

Note:

**`current_schemas(Boolean)`** returns an array of the names of all schemas presently in the search path. The Boolean option determines whether implicitly included system schemas such as **`pg_catalog`** are included in the returned search path.

 **NOTE**

The search path can be altered at run time. The command is:  
`SET search_path TO schema [, schema, ...]`

- `current_user`

Description: User name of current execution context

Return type: `name`

For example:

```
SELECT current_user;
 current_user
-----
 dbadmin
(1 row)
```

Note: **`current_user`** is the user identifier that is applicable for permission checking. Normally it is equal to the session user, but it can be changed with **`SET ROLE`**. It also changes during the execution of functions with the attribute **`SECURITY DEFINER`**.

- `inet_client_addr()`

Description: Remote connection address. **`inet_client_addr`** returns the IP address of the current client.

 **NOTE**

It is available only in remote connection mode.

Return type: `inet`

For example:

```
SELECT inet_client_addr();
 inet_client_addr
-----
 10.10.0.50
(1 row)
```

- `inet_client_port()`

Description: Remote connection port. And **`inet_client_port`** returns the port number of the current client.

 **NOTE**

It is available only in remote connection mode.

Return type: int

For example:

```
SELECT inet_client_port();
inet_client_port
-----
          33143
(1 row)
```

- `inet_server_addr()`

Description: Local connection address. **inet\_server\_addr** returns the IP address on which the server accepted the current connection.

 **NOTE**

It is available only in remote connection mode.

Return type: inet

For example:

```
SELECT inet_server_addr();
inet_server_addr
-----
10.10.0.13
(1 row)
```

- `inet_server_port()`

Description: Local connection port. **inet\_server\_port** returns the port number. All these functions return NULL if the current connection is via a Unix-domain socket.

 **NOTE**

It is available only in remote connection mode.

Return type: int

For example:

```
SELECT inet_server_port();
inet_server_port
-----
          8000
(1 row)
```

- `pg_backend_pid()`

Description: Process ID of the server process attached to the current session

Return type: int

For example:

```
SELECT pg_backend_pid();
pg_backend_pid
-----
140229352617744
(1 row)
```

- `pg_conf_load_time()`

Description: Configures load time. **pg\_conf\_load\_time** returns the timestamp with time zone when the server configuration files were last loaded.

Return type: timestamp with time zone

For example:

```
SELECT pg_conf_load_time();
pg_conf_load_time
-----
```

```
2017-09-01 16:05:23.89868+08
(1 row)
```

- **pg\_my\_temp\_schema()**

Description: OID of the temporary schema of a session. The value is 0 if the OID does not exist.

Return type: OID

For example:

```
SELECT pg_my_temp_schema();
pg_my_temp_schema
-----
0
(1 row)
```

Note: **pg\_my\_temp\_schema** returns the OID of the current session's temporary schema, or zero if it has none (because it has not created any temporary tables). **pg\_is\_other\_temp\_schema** returns true if the given OID is the OID of another session's temporary schema.

- **pg\_is\_other\_temp\_schema(oid)**

Description: Whether the schema is the temporary schema of another session.

Return type: Boolean

For example:

```
SELECT pg_is_other_temp_schema(25356);
pg_is_other_temp_schema
-----
f
(1 row)
```

- **pg\_listening\_channels()**

Description: Channel names that the session is currently listening on

Return type: setof text

For example:

```
SELECT pg_listening_channels();
pg_listening_channels
-----
(0 rows)
```

Note: **pg\_listening\_channels** returns a set of names of channels that the current session is listening to.

- **pg\_postmaster\_start\_time()**

Description: Server start time **pg\_postmaster\_start\_time** returns the **timestamp with time zone** when the server started.

Return type: timestamp with time zone

For example:

```
SELECT pg_postmaster_start_time();
pg_postmaster_start_time
-----
2017-08-30 16:02:54.99854+08
(1 row)
```

- **pg\_trigger\_depth()**

Description: Current nesting level of triggers

Return type: int

For example:

```
SELECT pg_trigger_depth();
pg_trigger_depth
```

```
-----
          0
(1 row)
```

- `pgxc_version()`

Description: Postgres-XC version information

Return type: text

For example:

```
SELECT pgxc_version();
          pgxc_version
-----
Postgres-XC 1.1 on x86_64-unknown-linux-gnu, based on PostgreSQL 9.2.4, compiled by g++ (GCC)
5.4.0, 64-bit
(1 row)
```

- `session_user`

Description: Session user name

Return type: name

For example:

```
SELECT session_user;
 session_user
-----
dbadmin
(1 row)
```

Note: **session\_user** is usually the user who initiated the current database connection, but administrators can change this setting with **SET SESSION AUTHORIZATION**.

- `user`

Description: Is equivalent to **current\_user**.

Return type: name

For example:

```
SELECT user;
 current_user
-----
dbadmin
(1 row)
```

- `version()`

Description: version information. **version** returns a string describing a server's version.

Return type: text

For example:

```
SELECT version();
          version
-----
PostgreSQL 9.2.4 gsql ((GaussDB A 8.0.0 build af002019) compiled at 2020-01-10 05:43:20 commit
6995 last mr 11566 ) on x86_64-unknown-linux-gnu, compiled by g++ (GCC) 5.4.0, 64-bit
(1 row)
```

## Access Privilege Inquiry Functions

- `has_any_column_privilege(user, table, privilege)`

Description: Queries whether a specified user has permission for any column of table.

Return type: Boolean

- `has_any_column_privilege(table, privilege)`

Description: Queries whether the current user has permission for any column of table.

Return type: Boolean

**has\_any\_column\_privilege** checks whether a user can access any column of a table in a particular way. Its parameter possibilities are analogous to **has\_table\_privilege**, except that the desired access permission type must be some combination of **SELECT**, **INSERT**, **UPDATE**, or **REFERENCES**.

 **NOTE**

Note that having any of these permissions at the table level implicitly grants it for each column of the table, so **has\_any\_column\_privilege** will always return **true** if **has\_table\_privilege** does for the same parameters. But **has\_any\_column\_privilege** also succeeds if there is a column-level grant of the permission for at least one column.

- `has_column_privilege(user, table, column, privilege)`

Description: Queries whether a specified user has permission for column.

Return type: Boolean

- `has_column_privilege(table, column, privilege)`

Description: Queries whether the current user has permission for column.

Return type: Boolean

**has\_column\_privilege** checks whether a user can access a column in a particular way. Its argument possibilities are analogous to **has\_table\_privilege**, with the addition that the column can be specified either by name or attribute number. The desired access permission type must evaluate to some combination of **SELECT**, **INSERT**, **UPDATE**, or **REFERENCES**.

 **NOTE**

Note that having any of these permissions at the table level implicitly grants it for each column of the table.

- `has_database_privilege(user, database, privilege)`

Description: Queries whether a specified user has permission for database.

Return type: Boolean

- `has_database_privilege(database, privilege)`

Description: Queries whether the current user has permission for database.

Return type: Boolean

Note: **has\_database\_privilege** checks whether a user can access a database in a particular way. Its argument possibilities are analogous to **has\_table\_privilege**. The desired access permission type must evaluate to some combination of **CREATE**, **CONNECT**, **TEMPORARY**, or **TEMP** (which is equivalent to **TEMPORARY**).

- `has_foreign_data_wrapper_privilege(user, fdw, privilege)`

Description: Queries whether a specified user has permission for foreign-data wrapper.

Return type: Boolean

- `has_foreign_data_wrapper_privilege(fdw, privilege)`



Description: Queries whether the current user has permission for foreign-data wrapper.

Return type: Boolean

Note: **has\_foreign\_data\_wrapper\_privilege** checks whether a user can access a foreign-data wrapper in a particular way. Its argument possibilities are analogous to **has\_table\_privilege**. The desired access permission type must evaluate to **USAGE**.

- **has\_function\_privilege**(user, function, privilege)

Description: Queries whether a specified user has permission for function.

Return type: Boolean

- **has\_function\_privilege**(function, privilege)

Description: Queries whether the current user has permission for function.

Return type: Boolean

Note: **has\_function\_privilege** checks whether a user can access a function in a particular way. Its argument possibilities are analogous to **has\_table\_privilege**. When a function is specified by a text string rather than by OID, the allowed input is the same as that for the **regprocedure** data type (see [Object Identifier Types](#)). The desired access permission type must evaluate to **EXECUTE**.

- **has\_language\_privilege**(user, language, privilege)

Description: Queries whether a specified user has permission for language.

Return type: Boolean

- **has\_language\_privilege**(language, privilege)

Description: Queries whether the current user has permission for language.

Return type: Boolean

Note: **has\_language\_privilege** checks whether a user can access a procedural language in a particular way. Its argument possibilities are analogous to **has\_table\_privilege**. The desired access permission type must evaluate to **USAGE**.

- **has\_schema\_privilege**(user, schema, privilege)

Description: Queries whether a specified user has permission for schema.

Return type: Boolean

- **has\_schema\_privilege**(schema, privilege)

Description: Queries whether the current user has permission for schema.

Return type: Boolean

Note: **has\_schema\_privilege** checks whether a user can access a schema in a particular way. Its argument possibilities are analogous to **has\_table\_privilege**. The desired access permission type must evaluate to some combination of **CREATE** or **USAGE**.

- **has\_server\_privilege**(user, server, privilege)

Description: Queries whether a specified user has permission for foreign server.

Return type: Boolean

- **has\_server\_privilege**(server, privilege)

Description: Queries whether the current user has permission for foreign server.

Return type: Boolean

Note: **has\_server\_privilege** checks whether a user can access a foreign server in a particular way. Its argument possibilities are analogous to **has\_table\_privilege**. The desired access permission type must evaluate to **USAGE**.

- **has\_table\_privilege(user, table, privilege)**

Description: Queries whether a specified user has permission for table.

Return type: Boolean

- **has\_table\_privilege(table, privilege)**

Description: Queries whether the current user has permission for table.

Return type: Boolean

**has\_table\_privilege** checks whether a user can access a table in a particular way. The user can be specified by name, by OID (**pg\_authid.oid**), **public** to indicate the PUBLIC pseudo-role, or if the argument is omitted **current\_user** is assumed. The table can be specified by name or by OID. When specifying by name, the name can be schema-qualified if necessary. The desired access permission type is specified by a text string, which must be one of the values **SELECT**, **INSERT**, **UPDATE**, **DELETE**, **TRUNCATE**, **REFERENCES**, or **TRIGGER**. Optionally, **WITH GRANT OPTION** can be added to a permission type to test whether the permission is held with grant option. Also, multiple permission types can be listed separated by commas, in which case the result will be **true** if any of the listed permissions is held.

For example:

```
SELECT has_table_privilege('tpcds.web_site', 'select');
has_table_privilege
-----
t
(1 row)

SELECT has_table_privilege('dbadmin', 'tpcds.web_site', 'select,INSERT WITH GRANT OPTION ');
has_table_privilege
-----
t
(1 row)
```

- **pg\_has\_role(user, role, privilege)**

Description: Queries whether a specified user has permission for role.

Return type: Boolean

- **pg\_has\_role(role, privilege)**

Description: Specifies whether the current user has permission for role.

Return type: Boolean

Note: **pg\_has\_role** checks whether a user can access a role in a particular way. Its argument possibilities are analogous to **has\_table\_privilege**, except that **public** is not allowed as a user name. The desired access permission type must evaluate to some combination of **MEMBER** or **USAGE**. **MEMBER** denotes direct or indirect membership in the role (that is, the right to do **SET ROLE**), while **USAGE** denotes the permissions of the role are available without doing **SET ROLE**.

## Schema Visibility Inquiry Functions

Each function performs the visibility check for one type of database object. For functions and operators, an object in the search path is visible if there is no object of the same name and argument data type(s) earlier in the path. For operator classes, both name and associated index access method are considered.

All these functions require object OIDs to identify the object to be checked. If you want to test an object by name, it is convenient to use the OID alias types (**regclass**, **regtype**, **regprocedure**, **regoperator**, **regconfig**, or **regdictionary**).

For example, a table is said to be visible if its containing schema is in the search path and no table of the same name appears earlier in the search path. This is equivalent to the statement that the table can be referenced by name without explicit schema qualification. For example, to list the names of all visible tables:

```
SELECT relname FROM pg_class WHERE pg_table_is_visible(oid);
```

- `pg_collation_is_visible(collation_oid)`  
Description: Queries whether the collation is visible in search path.  
Return type: Boolean
- `pg_conversion_is_visible(conversion_oid)`  
Description: Queries whether the conversion is visible in search path.  
Return type: Boolean
- `pg_function_is_visible(function_oid)`  
Description: Queries whether the function is visible in search path.  
Return type: Boolean
- `pg_opclass_is_visible(opclass_oid)`  
Description: Queries whether the operator class is visible in search path.  
Return type: Boolean
- `pg_operator_is_visible(operator_oid)`  
Description: Queries whether the operator is visible in search path.  
Return type: Boolean
- `pg_opfamily_is_visible(opclass_oid)`  
Description: Queries whether the operator family is visible in search path.  
Return type: Boolean
- `pg_table_is_visible(table_oid)`  
Description: Queries whether the table is visible in search path.  
Return type: Boolean
- `pg_ts_config_is_visible(config_oid)`  
Description: Queries whether the text search configuration is visible in search path.  
Return type: Boolean
- `pg_ts_dict_is_visible(dict_oid)`  
Description: Queries whether the text search dictionary is visible in search path.  
Return type: Boolean

- `pg_ts_parser_is_visible(parser_oid)`  
Description: Queries whether the text search parser is visible in search path.  
Return type: Boolean
- `pg_ts_template_is_visible(template_oid)`  
Description: Queries whether the text search template is visible in search path.  
Return type: Boolean
- `pg_type_is_visible(type_oid)`  
Description: Queries whether the type (or domain) is visible in search path.  
Return type: Boolean

## System Catalog Information Functions

- `format_type(type_oid, typemod)`  
Description: Gets SQL name of a data type.  
Return type: text  
Note:  
**format\_type** returns the SQL name of a data type that is identified by its type OID and possibly a type modifier. Pass NULL for the type modifier if no specific modifier is known. Certain type modifiers are passed for data types with length limitations. The SQL name returned from **format\_type** contains the length of the data type, which can be calculated by taking `sizeof(int32)` from actual storage length [`actual storage len - sizeof(int32)`] in the unit of bytes. 32-bit space is required to store the customized length set by users. So the actual storage length contains 4 bytes more than the customized length. In the following example, the SQL name returned from **format\_type** is `character varying(6)`, indicating the length of `varchar` type is 6 bytes. So the actual storage length of `varchar` type is 10 bytes.  

```
SELECT format_type((SELECT oid FROM pg_type WHERE typname='varchar'), 10);
format_type
-----
character varying(6)
(1 row)
```
- `pg_check_authid(role_oid)`  
Description: Check whether a role name with given OID exists.  
Return type: Boolean
- `pg_describe_object(catalog_id, object_id, object_sub_id)`  
Description: Gets description of a database object.  
Return type: text  
Note: **pg\_describe\_object** returns a description of a database object specified by catalog OID, object OID and a (possibly zero) sub-object ID. This is useful to determine the identity of an object as stored in the **pg\_depend** catalog.
- `pg_get_constraintdef(constraint_oid)`  
Description: Gets definition of a constraint.  
Return type: text
- `pg_get_constraintdef(constraint_oid, pretty_bool)`  
Description: Gets definition of a constraint.  
Return type: text

Note: **pg\_get\_constraintdef** and **pg\_get\_indexdef** respectively reconstruct the creating command for a constraint and an index.

- **pg\_get\_expr**(pg\_node\_tree, relation\_oid)

Description: Decompiles internal form of an expression, assuming that any Vars in it refer to the relationship indicated by the second parameter.

Return type: text

- **pg\_get\_expr**(pg\_node\_tree, relation\_oid, pretty\_bool)

Description: Decompiles internal form of an expression, assuming that any Vars in it refer to the relationship indicated by the second parameter.

Return type: text

Note: **pg\_get\_expr** decompiles the internal form of an individual expression, such as the default value for a column. It can be useful when examining the contents of system catalogs. If the expression might contain Vars, specify the OID of the relationship they refer to as the second parameter; if no Vars are expected, zero is sufficient.

- **pg\_get\_functiondef**(func\_oid)

Description: Gets definition of a function.

Return type: text

- **pg\_get\_function\_arguments**(func\_oid)

Description: Gets argument list of function's definition (with default values).

Return type: text

Note: **pg\_get\_function\_arguments** returns the argument list of a function, in the form it would need to appear in within **CREATE FUNCTION**.

- **pg\_get\_function\_identity\_arguments**(func\_oid)

Description: Gets argument list to identify a function (without default values).

Return type: text

Note: **pg\_get\_function\_identity\_arguments** returns the argument list necessary to identify a function, in the form it would need to appear in within **ALTER FUNCTION**. This form omits default values.

- **pg\_get\_function\_result**(func\_oid)

Description: Gets **RETURNS** clause for function.

Return type: text

Note: **pg\_get\_function\_result** returns the appropriate **RETURNS** clause for the function.

- **pg\_get\_indexdef**(index\_oid)

Description: Gets **CREATE INDEX** command for index.

Return type: text

- **pg\_get\_indexdef**(index\_oid, column\_no, pretty\_bool)

Description: Gets **CREATE INDEX** command for index, or definition of just one index column when **column\_no** is not zero.

Return type: text

Note: **pg\_get\_functiondef** returns a complete **CREATE OR REPLACE FUNCTION** statement for a function.

- `pg_get_keywords()`  
Description: Gets list of SQL keywords and their categories.  
Return type: setof record  
Note: **pg\_get\_keywords** returns a set of records describing the SQL keywords recognized by the server. The **word** column contains the keyword. The **catcode** column contains a category code: **U** for unreserved, **C** for column name, **T** for type or function name, or **R** for reserved. The **catdesc** column contains a possibly-localized string describing the category.
- `pg_get_ruledef(rule_oid)`  
Description: Gets **CREATE RULE** command for a rule.  
Return type: text
- `pg_get_ruledef(rule_oid, pretty_bool)`  
Description: Gets **CREATE RULE** command for a rule.  
Return type: text
- `pg_get_userbyid(role_oid)`  
Description: Gets role name with given OID.  
Return type: name  
Note: **pg\_get\_userbyid** extracts a role's name given its OID.
- `pg_get_viewdef(view_name)`  
Description: Gets underlying **SELECT** command for view.  
Return type: text
- `pg_get_viewdef(view_name, pretty_bool)`  
Description: Gets underlying **SELECT** command for view, lines with columns are wrapped to 80 columns if **pretty\_bool** is true.  
Return type: text  
Note: **pg\_get\_viewdef** reconstructs the **SELECT** query that defines a view. Most of these functions come in two variants. When the function has the parameter **pretty\_bool** and the value is true, it can optionally "pretty-print" the result. The pretty-printed format is more readable. The other one is default format which is more likely to be interpreted the same way by future versions of PostgreSQL. Avoid using pretty-printed output for dump purposes. Passing **false** for the pretty-print parameter yields the same result as the variant that does not have the parameter at all.
- `pg_get_viewdef(view_oid)`  
Description: Gets underlying **SELECT** command for view.  
Return type: text
- `pg_get_viewdef(view_oid, pretty_bool)`  
Description: Gets underlying **SELECT** command for view, lines with columns are wrapped to 80 columns if **pretty\_bool** is true.  
Return type: text
- `pg_get_viewdef(view_oid, wrap_column_int)`  
Description: Gets underlying **SELECT** command for view, wrapping lines with columns as specified, printing is implied.  
Return type: text

- pg\_get\_tabledef(table\_oid)**  
Description: Obtains a table definition based on **table\_oid**.  
Return type: text
- pg\_get\_tabledef(table\_name)**  
Description: Obtains a table definition based on **table\_name**.  
Return type: text  
Remarks: **pg\_get\_tabledef** reconstructs the **CREATE** statement of the table definition, including the table definition, index information, and comments. Users need to create the dependent objects of the table, such as groups, schemas, tablespaces, and servers. The table definition does not include the statements for creating these dependent objects.
- pg\_options\_to\_table(reloptions)**  
Description: Gets the set of storage option name/value pairs.  
Return type: setof record  
Note: **pg\_options\_to\_table** returns the set of storage option name/value pairs (**option\_name/option\_value**) when passed **pg\_class.reloptions** or **pg\_attribute.attoptions**.
- pg\_typeof(any)**  
Description: Gets the data type of any value.  
Return type: regtype  
Note:  
**pg\_typeof** returns the OID of the data type of the value that is passed to it. This can be helpful for troubleshooting or dynamically constructing SQL queries. The function is declared as returning **regtype**, which is an OID alias type (see [Object Identifier Types](#)). This means that it is the same as an OID for comparison purposes but displays as a type name.  
For example:  

```
SELECT pg_typeof(33);
pg_typeof
-----
integer
(1 row)

SELECT typlen FROM pg_type WHERE oid = pg_typeof(33);
typlen
-----
4
(1 row)
```
- collation for (any)**  
Description: Gets the collation of the parameter.  
Return type: text  
Note:  
The expression **collation for** returns the collation of the value that is passed to it. For example:  

```
SELECT collation for (description) FROM pg_description LIMIT 1;
pg_collation_for
-----
"default"
(1 row)
```

The value might be quoted and schema-qualified. If no collation is derived for the argument expression, then a null value is returned. If the parameter is not of a collectable data type, then an error is thrown.

- `getdistributekey(table_name)`

Description: Gets a distribution column for a hash table.

Return type: text

For example:

```
SELECT getdistributekey('item');
getdistributekey
-----
i_item_sk
(1 row)
```

## Comment Information Functions

- `col_description(table_oid, column_number)`

Description: Gets comment for a table column.

Return type: text

Note: **col\_description** returns the comment for a table column, which is specified by the OID of its table and its column number.

- `obj_description(object_oid, catalog_name)`

Description: Gets comment for a shared database object.

Return type: text

Note: The two-parameter form of **obj\_description** returns the comment for a database object specified by its OID and the name of the containing system catalog. For example, **obj\_description(123456,'pg\_class')** would retrieve the comment for the table with OID 123456. The one-parameter form of **obj\_description** requires only the object OID.

**obj\_description** cannot be used for table columns since columns do not have OIDs of their own.

- `obj_description(object_oid)`

Description: Gets comment for a shared database object.

Return type: text

- `shobj_description(object_oid, catalog_name)`

Description: Gets comment for a shared database object.

Return type: text

Note: **shobj\_description** is used just like **obj\_description** except the former is used for retrieving comments on shared objects. Some system catalogs are global to all databases within each cluster, and the comments for objects in them are stored globally as well.

## Transaction IDs and Snapshots

The following functions provide server transaction information in an exportable form. The main use of these functions is to determine which transactions were committed between two snapshots.

- `pgxc_is_committed(transaction_id)`



Description: Determines whether the given XID is committed or ignored. NULL indicates the unknown status (such as running, preparing, and freezing).

Return type: Boolean

- txid\_current()

Description: Gets current transaction ID.

Return type: bigint

- txid\_current\_snapshot()

Description: Gets current snapshot.

Return type: txid\_snapshot

- txid\_snapshot\_xip(txid\_snapshot)

Description: Gets in-progress transaction IDs in snapshot.

Return type: setof bigint

- txid\_snapshot\_xmax(txid\_snapshot)

Description: Gets **xmax** of snapshot.

Return type: bigint

- txid\_snapshot\_xmin(txid\_snapshot)

Description: Gets **xmin** of snapshot.

Return type: bigint

- txid\_visible\_in\_snapshot(bigint, txid\_snapshot)

Description: Queries whether the transaction ID is visible in snapshot. (do not use with subtransaction ids)

Return type: Boolean

The internal transaction ID type (**xid**) is 32 bits wide and wraps around every 4 billion transactions. **txid\_snapshot**, the data type used by these functions, stores information about transaction ID visibility at a particular moment in time. [Table 6-8](#) describes its components.

**Table 6-8** Snapshot components

| Name     | Description                                                                                                                                                                                                                                                                                                                                                                                                                       |
|----------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| xmin     | Earliest transaction ID (txid) that is still active. All earlier transactions will either be committed and visible, or rolled back.                                                                                                                                                                                                                                                                                               |
| xmax     | First as-yet-unassigned txid. All txids greater than or equal to this are not yet started as of the time of the snapshot, so they are invisible.                                                                                                                                                                                                                                                                                  |
| xip_list | Active txids at the time of the snapshot. The list includes only those active txids between <b>xmin</b> and <b>xmax</b> ; there might be active txids higher than <b>xmax</b> . A txid that is <b>xmin</b> <= <b>txid</b> < <b>xmax</b> and not in this list was already completed at the time of the snapshot, and is either visible or dead according to its commit status. The list does not include txids of subtransactions. |

**txid\_snapshot**'s textual representation is **xmin:xmax:xip\_list**.

For example: **10:20:10,14,15** means **xmin=10, xmax=20, xip\_list=10, 14, 15**.

## Computing Node Group Function

`pv_compute_pool_workload()`

Description: Load status of a computing Node Group.

Return type: void

Example:

```
SELECT * from pv_compute_pool_workload();
nodename | rpinuse | maxrp | nodestate
-----+-----+-----+-----
datanode1 | 0 | 1000 | normal
datanode2 | 0 | 1000 | normal
(2 rows)
```

## 6.25 System Administration Functions

### 6.25.1 Configuration Settings Functions

Configuration setting functions are used for querying and modifying configuration parameters during running.

- `current_setting(setting_name)`

Description: Specifies the current setting.

Return type: text

Note: **current\_setting** obtains the current setting of **setting\_name** by query. It is equivalent to the **SHOW** statement. For example:

```
SELECT current_setting('datestyle');

current_setting
-----
ISO, MDY
(1 row)
```

- `set_config(setting_name, new_value, is_local)`

Description: Sets the parameter and returns a new value.

Return type: text

Note: **set\_config** sets the parameter **setting\_name** to **new\_value**. If **is\_local** is **true**, the new value will only apply to the current transaction. If you want the new value to apply for the current session, use **false** instead. The function corresponds to the **SET** statement. For example:

```
SELECT set_config('log_statement_stats', 'off', false);

set_config
-----
off
(1 row)
```

### 6.25.2 Universal File Access Functions

Universal file access functions provide local access interfaces for files on a database server. Only files in the database cluster directory and the **log\_directory**

directory can be accessed. Use a relative path for files in the cluster directory, and a path matching the **log\_directory** configuration setting for log files. Only database system administrators can use these functions.

- **pg\_ls\_dir**(dirname text)

Description: Lists files in a directory.

Return type: setof text

Note: **pg\_ls\_dir** returns all the names in the specified directory, except the special entries "." and "..".

For example:

```
SELECT pg_ls_dir('./');
 pg_ls_dir
```

```
-----
.postgresql.conf.swp
postgresql.conf
pg_tblspc
PG_VERSION
pg_ident.conf
core
server.crt
pg_serial
pg_twophase
postgresql.conf.lock
pg_stat_tmp
pg_notify
pg_subtrans
pg_ctl.lock
pg_xlog
pg_clog
base
pg_snapshots
postmaster.opts
postmaster.pid
server.key.rand
server.key.cipher
pg_multixact
pg_errorinfo
server.key
pg_hba.conf
pg_replslot
.pg_hba.conf.swp
cacert.pem
pg_hba.conf.lock
global
gaussdb.state
(32 rows)
```

- **pg\_read\_binary\_file**(filename text [, offset bigint, length bigint,missing\_ok boolean])

Description: Returns the content of a binary file.

Return type: bytea

Note: **pg\_read\_binary\_file** is similar to **pg\_read\_file**, except that the result is a **bytea** value; accordingly, no encoding checks are performed. In combination with the **convert\_from** function, this function can be used to read a file in a specified encoding:

```
SELECT convert_from(pg_read_binary_file('filename'), 'UTF8');
```

- **pg\_stat\_file**(filename text)

Description: Returns status information about a file.

Return type: record

Note: **pg\_stat\_file** returns a record containing the file size, last access timestamp, last modification timestamp, last file status change timestamp, and a **Boolean** value indicating if it is a directory. Typical use cases are as follows:

```
SELECT * FROM pg_stat_file('filename');
SELECT (pg_stat_file('filename')).modification;
```

For example:

```
SELECT * FROM pg_stat_file('postmaster.pid');

 size |      access      |      modification      |      change
-----+-----+-----+-----
| creation | isdir
-----+-----+-----+-----
+-----+-----+-----+-----
| 117 | 2017-06-05 11:06:34+08 | 2017-06-01 17:18:08+08 | 2017-06-01 17:18:08+08
|      | f
(1 row)
SELECT (pg_stat_file('postmaster.pid')).modification;
      modification
-----
2017-06-01 17:18:08+08
(1 row)
```

### 6.25.3 Server Signaling Functions

Server signaling functions send control signals to other server processes. Only system administrators can use these functions.

- **pg\_cancel\_backend**(pid int)

Description: Cancels the current query of a backend.

Return type: Boolean

Note: **pg\_cancel\_backend** sends a query cancellation (SIGINT) signal to the backend process identified by **pid**. The PID of an active backend process can be found in the **pid** column of the **pg\_stat\_activity** view, or can be found by listing the database process using **ps** on the server.

- **pg\_rotate\_logfile**()

Description: Rotates the log files of the server.

Return type: Boolean

Note: **pg\_rotate\_logfile** sends a signal to the log file manager, instructing the manager to immediately switch to a new output file. This function works only when **redirect\_stderr** is used for log output. Otherwise, no log file manager subprocess exists.

- **pg\_terminate\_backend**(pid int)

Description: Terminates a backend thread.

Return type: Boolean

Note: Each of these functions returns **true** if they are successful and **false** otherwise.

For example:

```
SELECT pid from pg_stat_activity;
      pid
-----
140657876268816
(1 rows)
SELECT pg_terminate_backend(140657876268816);
```

```
pg_terminate_backend
```

```
t  
(1 row)
```

- `pg_wlm_jump_queue(pid int)`  
Description: Moves a task to the top of the CN queue.  
Return type: Boolean  
Note: Each of these functions returns **true** if they are successful and **false** otherwise.
- `gs_wlm_switch_cgroup(pid int, cgroup text)`  
Description: Moves a job to other Cgroup to improve the job priority.  
Return type: Boolean  
Note: Each of these functions returns **true** if they are successful and **false** otherwise.

## 6.25.4 Backup and Restoration Control Functions

### Backup Control Functions

Backup control functions help online backup.

- `pg_create_restore_point(name text)`  
Description: Creates a named point for performing the restore operation (restricted to system administrators).  
Return type: text  
Note: **pg\_create\_restore\_point** creates a named transaction log record that can be used as a restoration target, and returns the corresponding transaction log location. The given name can then be used with **recovery\_target\_name** to specify the point up to which restoration will proceed. Avoid creating multiple restoration points with the same name, since restoration will stop at the first one whose name matches the restoration target.
- `pg_current_xlog_location()`  
Description: Obtains the write position of the current transaction log.  
Return type: text  
Note: **pg\_current\_xlog\_location** displays the write position of the current transaction log in the same format as those of the previous functions. Read-only operations do not require rights of the system administrator.
- `pg_current_xlog_insert_location()`  
Description: Obtains the insert position of the current transaction log.  
Return type: text  
Note: **pg\_current\_xlog\_insert\_location** displays the insert position of the current transaction log. The insertion point is the logical end of the transaction log at any instant, while the write location is the end of what has been written out from the server's internal buffers. The write position is the end that can be detected externally from the server. This operation can be performed to archive only some of completed transaction log files. The insert position is mainly used for commissioning the server. Read-only operations do not require rights of the system administrator.

- `pg_start_backup(label text [, fast boolean ])`

Description: Starts executing online backup (restricted to system administrators or replication roles).

Return type: text

Note: **pg\_start\_backup** receives a user-defined backup label (usually the name of the position where the backup dump file is stored). This function writes a backup label file to the data directory of the database cluster and then returns the starting position of backed up transaction logs in text mode.

```
SELECT pg_start_backup('label_goes_here');
pg_start_backup
-----
0/3000020
(1 row)
```
- `pg_stop_backup()`

Description: Completes online backup (restricted to system administrators or replication roles).

Return type: text

Note: **pg\_stop\_backup** deletes the label file created by **pg\_start\_backup** and creates a backup history file in the transaction log archive area. The history file includes the label given to **pg\_start\_backup**, the starting and ending transaction log locations for the backup, and the starting and ending times of the backup. The return value is the backup's ending transaction log location. After the ending position is calculated, the insert position of the current transaction log automatically goes ahead to the next transaction log file. This way, the ended transaction log file can be immediately archived so that backup is complete.
- `pg_switch_xlog()`

Description: Switches to a new transaction log file (restricted to system administrators).

Return type: text

Note: **pg\_switch\_xlog** moves to the next transaction log file so that the current log file can be archived (if continuous archive is used). The return value is the ending transaction log location + 1 within the just-completed transaction log file. If there has been no transaction log activity since the last transaction log switchover, **pg\_switch\_xlog** will do nothing but return the start location of the transaction log file currently in use.
- `pg_xlogfile_name(location text)`

Description: Converts the position string in a transaction log to a file name.

Return type: text

Note: **pg\_xlogfile\_name** extracts only the transaction log file name. If the given transaction log position is the transaction log file border, a transaction log file name will be returned for both the two functions. This is usually the desired behavior for managing transaction log archiving, since the preceding file is the last one that currently needs to be archived.
- `pg_xlogfile_name_offset(location text)`

Description: Converts the position string in a transaction log to a file name and returns the byte offset in the file.

Return type: text, integer

Note: **pg\_xlogfile\_name\_offset** can extract transaction log file names and byte offsets from the returned results of the preceding functions. For example:

```
SELECT * FROM pg_xlogfile_name_offset(pg_stop_backup());
NOTICE: pg_stop_backup cleanup done, waiting for required WAL segments to be archived
NOTICE: pg_stop_backup complete, all required WAL segments have been archived
  file_name      | file_offset
-----+-----
000000010000000000000003 |      272
(1 row)
```

- pg\_xlog\_location\_diff**(location text, location text)

Description: **pg\_xlog\_location\_diff** calculates the difference in bytes between two transaction log locations.

Return type: numeric
- pgxc\_get\_senders\_catchup\_time**()

Description: Displays the catchup information of the currently active primary/standby instance sending thread on all DNs.

Return type: record

The following information is returned:

**Table 6-9** pgxc\_get\_senders\_catchup\_time() columns

| Name          | Type                     | Description                         |
|---------------|--------------------------|-------------------------------------|
| node_name     | text                     | Node name                           |
| lwpid         | integer                  | Current sender lwpid                |
| local_role    | text                     | Local role                          |
| peer_role     | text                     | Peer role                           |
| state         | text                     | Current sender's replication status |
| sender        | text                     | Current sender type                 |
| catchup_start | timestamp with time zone | Startup time of a catchup task      |
| catchup_end   | timestamp with time zone | End time of a catchup task          |

## Restoration Control Functions

Restoration control functions provide information about the status of standby nodes. These functions may be executed both during restoration and in normal running.

- pg\_is\_in\_recovery**()

Description: Returns **true** if restoration is still in progress.

Return type: bool
- pg\_last\_xlog\_receive\_location**()

Description: Gets the last transaction log location received and synchronized to disk by streaming replication. While streaming replication is in progress, this will increase monotonically. If restoration has completed, then this value will remain static at the value of the last WAL record received and synchronized to disk during restoration. If streaming replication is disabled or if not yet started, the function return will return **NULL**.

Return type: text

- `pg_last_xlog_replay_location()`

Description: Gets last transaction log location replayed during restoration. If restoration is still in progress, this will increase monotonically. If restoration has completed, then this value will remain static at the value of the last WAL record received during that restoration. When the server has been started normally without restoration, the function returns **NULL**.

Return type: text

- `pg_last_xact_replay_timestamp()`

Description: Gets the timestamp of last transaction replayed during restoration. This is the time to commit a transaction or abort a WAL record on the primary node. If no transactions have been replayed during restoration, this function will return **NULL**. Otherwise, if restoration is still in progress, this will increase monotonically. If restoration has completed, then this value will remain static at the value of the last WAL record received during that restoration. If the server normally starts without manual intervention, this function will return **NULL**.

Return type: timestamp with time zone

Restoration control functions control restoration processes. These functions may be executed only during restoration.

- `pg_is_xlog_replay_paused()`

Description: Returns **true** if restoration is paused.

Return type: bool

- `pg_xlog_replay_pause()`

Description: Pauses restoration immediately.

Return type: void

- `pg_xlog_replay_resume()`

Description: Restarts restoration if it was paused.

Return type: void

While restoration is paused, no further database changes are applied. In hot standby mode, all new queries will see the same consistent snapshot of the database, and no further query conflicts will be generated until restoration is resumed.

If streaming replication is disabled, the paused state may continue indefinitely without problem. While streaming replication is in progress, WAL records will continue to be received, which will eventually fill available disk space. This progress depends on the duration of the pause, the rate of WAL generation, and available disk space.



## 6.25.5 Snapshot Synchronization Functions

Snapshot synchronization functions save the current snapshot and return its identifier.

`pg_export_snapshot()`

Description: Saves the current snapshot and returns its identifier.

Return type: text

Note: **pg\_export\_snapshot** saves the current snapshot and returns a text string identifying the snapshot. This string must be passed to clients that want to import the snapshot. A snapshot can be imported when the **set transaction snapshot snapshot\_id**; command is executed. Doing so is possible only when the transaction is set to the **REPEATABLE READ** isolation level. The output of the function cannot be used as the input of **set transaction snapshot**.

## 6.25.6 Database Object Functions

### Database Object Size Functions

Database object size functions calculate the actual disk space used by database objects.

- `pg_column_size(any)`

Description: Specifies the number of bytes used to store a particular value (possibly compressed).

Return type: int

Note: **pg\_column\_size** displays the space for storing an independent data value.

```
SELECT pg_column_size(1);
 pg_column_size
-----
          4
(1 row)
```

- `pg_database_size(oid)`

Description: Specifies the disk space used by the database with the specified OID.

Return type: bigint

- `pg_database_size(name)`

Description: Specifies the disk space used by the database with the specified name.

Return type: bigint

Note: **pg\_database\_size** receives the OID or name of a database and returns the disk space used by the corresponding object.

For example:

```
SELECT pg_database_size('postgres');
 pg_database_size
-----
          51590112
(1 row)
```

- pg\_relation\_size(oid)**  
Description: Specifies the disk space used by the table with a specified OID or index.  
Return type: bigint
- get\_db\_source\_datasize()**  
Description: Estimates the total size of non-compressed data in the current database.  
Return type: bigint  
Note: (1) **ANALYZE** must be performed before this function is called. (2) Calculate the total size of non-compressed data by estimating the compression rate of column-store tables.  
For example:

```
analyze;
ANALYZE
select get_db_source_datasize();
get_db_source_datasize
-----
          35384925667
(1 row)
```
- pg\_relation\_size(text)**  
Description: Specifies the disk space used by the table with a specified name or index. The table name can be schema-qualified.  
Return type: bigint
- pg\_relation\_size(relation regclass, fork text)**  
Description: Specifies the disk space used by the specified bifurcating tree ('main', 'fsm', or 'vm') of a certain table or index.  
Return type: bigint
- pg\_relation\_size(relation regclass)**  
Description: Is an abbreviation of **pg\_relation\_size(..., 'main')**.  
Return type: bigint  
Note: **pg\_relation\_size** receives the OID or name of a table, index, or compressed table, and returns the size.
- pg\_partition\_size(oid,oid)**  
Description: Specifies the disk space used by the partition with a specified OID. The first **oid** is the OID of the table and the second **oid** is the OID of the partition.  
Return type: bigint
- pg\_partition\_size(text, text)**  
Description: Specifies the disk space used by the partition with a specified name. The first **text** is the table name and the second **text** is the partition name.  
Return type: bigint
- pg\_partition\_indexes\_size(oid,oid)**  
Description: Specifies the disk space used by the index of the partition with a specified OID. The first **oid** is the OID of the table and the second **oid** is the OID of the partition.  
Return type: bigint

- `pg_partition_indexes_size(text,text)`  
Description: Specifies the disk space used by the index of the partition with a specified name. The first **text** is the table name and the second **text** is the partition name.  
Return type: `bigint`
- `pg_indexes_size(regclass)`  
Description: Specifies the total disk space used by the index appended to the specified table.  
Return type: `bigint`
- `pg_size_pretty(bigint)`  
Description: Converts the calculated byte size into a size readable to human beings.  
Return type: `text`
- `pg_size_pretty(numeric)`  
Description: Converts the calculated byte size indicated by a numeral into a size readable to human beings.  
Return type: `text`  
Note: **pg\_size\_pretty** formats the results of other functions into a human-readable format. KB/MB/GB/TB can be used.
- `pg_table_size(regclass)`  
Description: Specifies the disk space used by the specified table, excluding indexes (but including TOAST, free space mapping, and visibility mapping).  
Return type: `bigint`
- `pg_total_relation_size(oid)`  
Description: Specifies the disk space used by the table with a specified OID, including the index and the compressed data.  
Return type: `bigint`
- `pg_total_relation_size(regclass)`  
Description: Specifies the total disk space used by the specified table, including all indexes and TOAST data.  
Return type: `bigint`
- `pg_total_relation_size(text)`  
Description: Specifies the disk space used by the table with a specified name, including the index and the compressed data. The table name can be schema-qualified.  
Return type: `bigint`  
Note: **pg\_total\_relation\_size** receives the OID or name of a table or a compressed table, and returns the sizes of the data, related indexes, and the compressed table in bytes.

## Database Object Position Functions

- `pg_relation_filenode(regclass)`  
Description: Specifies the ID of a filenode with the specified relationship.  
Return type: `oid`

Description: **pg\_relation\_filenode** receives the OID or name of a table, index, sequence, or compressed table, and returns the filenode number allocated to it. The filenode is the basic component of the file name used by the relationship. For most tables, the result is the same as that of **pg\_class.relfilenode**. For the specified system directory, **relfilenode** is **0** and this function must be used to obtain the correct value. If a relationship that is not stored is transmitted, such as a view, this function returns **NULL**.

- **pg\_relation\_filepath**(relation regclass)

Description: Specifies the name of a file path with the specified relationship.

Return type: text

Description: **pg\_relation\_filepath** is similar to **pg\_relation\_filenode**, except that **pg\_relation\_filepath** returns the whole file path name for the relationship (relative to the data directory **PGDATA** of the database cluster).

## 6.25.7 Advisory Lock Functions

Advisory lock functions manage advisory locks. These functions are only for internal use currently.

- **pg\_advisory\_lock**(key bigint)

Description: Obtains an exclusive session-level advisory lock.

Return type: void

Note: **pg\_advisory\_lock** locks resources defined by an application. The resources can be identified using a 64-bit or two nonoverlapped 32-bit key values. If another session locks the resources, the function blocks the resources until they can be used. The lock is exclusive. Multiple locking requests are pushed into the stack. Therefore, if the same resource is locked three times, it must be unlocked three times so that it is released to another session.

- **pg\_advisory\_lock**(key1 int, key2 int)

Description: Obtains an exclusive session-level advisory lock.

Return type: void

- **pg\_advisory\_lock\_shared**(key bigint)

Description: Obtains a shared session-level advisory lock.

Return type: void

- **pg\_advisory\_lock\_shared**(key1 int, key2 int)

Description: Obtains a shared session-level advisory lock.

Return type: void

Note: **pg\_advisory\_lock\_shared** works in the same way as **pg\_advisory\_lock**, except the lock can be shared with other sessions requesting shared locks. Only would-be exclusive lockers are locked out.

- **pg\_advisory\_unlock**(key bigint)

Description: Releases an exclusive session-level advisory lock.

Return type: Boolean

- **pg\_advisory\_unlock**(key1 int, key2 int)

Description: Releases an exclusive session-level advisory lock.

Return type: Boolean

Note: **pg\_advisory\_unlock** releases the obtained exclusive advisory lock. If the release is successful, the function returns **true**. If the lock was not held, it will return **false**. In addition, a SQL warning will be reported by the server.

- **pg\_advisory\_unlock\_shared**(key bigint)

Description: Releases a shared session-level advisory lock.

Return type: Boolean

- **pg\_advisory\_unlock\_shared**(key1 int, key2 int)

Description: Releases a shared session-level advisory lock.

Return type: Boolean

Note: **pg\_advisory\_unlock\_shared** works in the same way as **pg\_advisory\_unlock**, except it releases a shared session-level advisory lock.

- **pg\_advisory\_unlock\_all**()

Description: Releases all advisory locks owned by the current session.

Return type: void

Note: **pg\_advisory\_unlock\_all** releases all advisory locks owned by the current session. The function is implicitly invoked when the session ends even if the client is abnormally disconnected.

- **pg\_advisory\_xact\_lock**(key bigint)

Description: Obtains an exclusive transaction-level advisory lock.

Return type: void

- **pg\_advisory\_xact\_lock**(key1 int, key2 int)

Description: Obtains an exclusive transaction-level advisory lock.

Return type: void

Note: **pg\_advisory\_xact\_lock** works in the same way as **pg\_advisory\_lock**, except the lock is automatically released at the end of the current transaction and cannot be released explicitly.

- **pg\_advisory\_xact\_lock\_shared**(key bigint)

Description: Obtains a shared transaction-level advisory lock.

Return type: void

- **pg\_advisory\_xact\_lock\_shared**(key1 int, key2 int)

Description: Obtains a shared transaction-level advisory lock.

Return type: void

Note: **pg\_advisory\_xact\_lock\_shared** works in the same way as **pg\_advisory\_lock\_shared**, except the lock is automatically released at the end of the current transaction and cannot be released explicitly.

- **pg\_try\_advisory\_lock**(key bigint)

Description: Obtains an exclusive session-level advisory lock if available.

Return type: Boolean

Note: **pg\_try\_advisory\_lock** is similar to **pg\_advisory\_lock**, except **pg\_try\_advisory\_lock** does not block the resource until the resource is released. **pg\_try\_advisory\_lock** either immediately obtains the lock and returns **true** or returns **false**, which indicates the lock cannot be performed currently.

- `pg_try_advisory_lock(key1 int, key2 int)`  
Description: Obtains an exclusive session-level advisory lock if available.  
Return type: Boolean
- `pg_try_advisory_lock_shared(key bigint)`  
Description: Obtains a shared session-level advisory lock if available.  
Return type: Boolean
- `pg_try_advisory_lock_shared(key1 int, key2 int)`  
Description: Obtains a shared session-level advisory lock if available.  
Return type: Boolean  
Note: **`pg_try_advisory_lock_shared`** is similar to **`pg_try_advisory_lock`**, except **`pg_try_advisory_lock_shared`** attempts to obtain a shared lock instead of an exclusive lock.
- `pg_try_advisory_xact_lock(key bigint)`  
Description: Obtains an exclusive transaction-level advisory lock if available.  
Return type: Boolean
- `pg_try_advisory_xact_lock(key1 int, key2 int)`  
Description: Obtains an exclusive transaction-level advisory lock if available.  
Return type: Boolean  
Note: **`pg_try_advisory_xact_lock`** works in the same way as **`pg_try_advisory_lock`**, except the lock, if acquired, is automatically released at the end of the current transaction and cannot be released explicitly.
- `pg_try_advisory_xact_lock_shared(key bigint)`  
Description: Obtains a shared transaction-level advisory lock if available.  
Return type: Boolean
- `pg_try_advisory_xact_lock_shared(key1 int, key2 int)`  
Description: Obtains a shared transaction-level advisory lock if available.  
Return type: Boolean  
Note: **`pg_try_advisory_xact_lock_shared`** works in the same way as **`pg_try_advisory_lock_shared`**, except the lock, if acquired, is automatically released at the end of the current transaction and cannot be released explicitly.

## 6.25.8 Replication Functions:

A replication function synchronizes logs and data between instances. It is a statistics or operation method provided by the system to implement HA.

### NOTE

Replication functions except statistics queries are internal functions. You are not advised to use them directly.

- `pg_create_logical_replication_slot('slot_name', 'plugin_name')`  
Description: Creates a logical replication slot.  
Parameter:
  - `slot_name`

Indicates the name of the streaming replication slot.

Value range: a string, supporting only letters, digits, and the following special characters: \_?-..

- plugin\_name

Indicates the name of the plugin.

Value range: a string, supporting only **mppdb\_decoding**

Return type: name, text

Note: The first return value is the slot name, and the second is the start LSN position for decoding in the logical replication slot.

- pg\_create\_physical\_replication\_slot ('slot\_name', isDummyStandby)

Description: Creates a physical replication slot.

Parameter:

- slot\_name

Indicates the name of the streaming replication slot.

Value range: a string, supporting only letters, digits, and the following special characters: \_?.-

- isDummyStandby

Indicates whether the replication slot is the secondary one.

Value range: a boolean value, **true** or **false**

Return type: name, text

Note: The first return value is the slot name, and the second is the start LSN position for decoding in the physical replication slot.

- pg\_get\_replication\_slots()

Description: Displays information about all replication slots on the current DN.

Return type: record

The following information is returned:

**Table 6-10** pg\_get\_replication\_slots() fields

| Field     | Type    | Description                                                |
|-----------|---------|------------------------------------------------------------|
| slot_name | text    | Replication slot name                                      |
| plugin    | name    | Name of the output plug-in of the logical replication slot |
| slot_type | text    | Replication slot type                                      |
| datoid    | oid     | Replication slot's database OID                            |
| active    | boolean | Whether the replication slot is active                     |
| xmin      | xid     | Transaction ID of the replication slot                     |

| Field         | Type    | Description                                                                                                                                                                                             |
|---------------|---------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| catalog_xmin  | text    | ID of the earliest-decoded transaction corresponding to the logical replication slot.<br>Xlog file information on the replication slot.<br>Indicates whether the replication slot is the secondary one. |
| restart_lsn   | text    |                                                                                                                                                                                                         |
| dummy_standby | boolean |                                                                                                                                                                                                         |

- `pg_drop_replication_slot('slot_name')`

Description: Deletes a streaming replication slot.

Parameter:

- `slot_name`

Indicates the name of the streaming replication slot.

Value range: a string, supporting only letters, digits, and the following special characters: `_?-`.

Return type: void

- `pg_logical_slot_peek_changes('slot_name', 'LSN', upto_nchanges, 'options_name', 'options_value')`

Description: Performs decoding but does not go to the next streaming replication slot. (The decoding result will be returned again on future calls.)

Parameter:

- `slot_name`

Indicates the name of the streaming replication slot.

Value range: a string, supporting only letters, digits, and the following special characters: `_?-`.

- `LSN`

Indicates a target LSN. Decoding is performed only when an LSN is less than or equal to this value.

Value range: a string, in the format of `xlogid/xrecoff`, for example, `'1/2AAFC60'` (If this parameter is set to **NULL**, the target LSN indicating the end position of decoding is not specified.)

- `upto_nchanges`

Indicates the number of decoded records (including the **begin** and **commit** timestamps). Assume that there are three transactions, which involve 3, 5, and 7 records, respectively. If **upto\_nchanges** is **4**, 8 records of the first two transactions will be decoded. Specifically, decoding is stopped when the number of decoded records exceeds 4 after decoding in the first two transactions is finished.

Value range: a non-negative integer

#### NOTE

If any of the **LSN** and **upto\_nchanges** values are reached, decoding ends.

- `options` (optional)



- **include-xids**  
Indicates whether the decoded **data** column contains XID information.  
Valid value: **0** and **1**. The default value is **1**.
  - **0**: The decoded **data** column does not contain XID information.
  - **1**: The decoded **data** column contains XID information.
- **skip-empty-xacts**  
Indicates whether to ignore empty transaction information during decoding.  
Valid value: **0** and **1**. The default value is **0**.
  - **0**: The empty transaction information is not ignored during decoding.
  - **1**: The empty transaction information is ignored during decoding.
- **include-timestamp**  
Indicates whether decoding information contains the **commit** timestamp.  
Valid value: **0** and **1**. The default value is **0**.
  - **0**: The decoding information does not contain the **commit** timestamp.
  - **1**: The decoding information contains the **commit** timestamp.

Return type: text, uint, text

Note: The function returns the decoding result. Each decoding result contains three columns, corresponding to the above return types and indicating the LSN position, XID, and decoded content, respectively.

- `pg_logical_slot_get_changes('slot_name', 'LSN', upto_nchanges, 'options_name', 'options_value')`

Description: Performs decoding and goes to the next streaming replication slot.

Parameter: This function has the same parameters as **pg\_logical\_slot\_peek\_changes**. For details, see [pg\\_logical\\_slot\\_peek\\_ch...](#)

- `pg_replication_slot_advance ('slot_name', 'LSN')`

Description: Directly goes to the streaming replication slot for a specified LSN, without outputting any decoding result.

Parameter:

- `slot_name`

Indicates the name of the streaming replication slot.

Value range: a string, supporting only letters, digits, and the following special characters: `_?-`.

- `LSN`

Indicates a target LSN. Next decoding will be performed only in transactions whose commission position is greater than this value. If an input LSN is smaller than the position recorded in the current streaming

replication slot, the function directly returns. If the input LSN is greater than the LSN of the current physical log, the latter LSN will be directly used for decoding.

Value range: a string, in the format of xlogid/xrecoff

Return type: name, text

Note: A return result contains the slot name and LSN that is actually used for decoding.

- `pg_stat_get_data_senders()`

Description: Displays statistics about replication sending threads on all data page on the current DN.

Return type: record

The following information is returned:

**Table 6-11** `pg_stat_get_data_senders()` fields

| Field         | Type                     | Description                         |
|---------------|--------------------------|-------------------------------------|
| pid           | bigint                   | Thread PID                          |
| sender_pid    | integer                  | Current sender PID                  |
| local_role    | text                     | Local role                          |
| peer_role     | text                     | Peer role                           |
| state         | text                     | Current sender's replication status |
| catchup_start | timestamp with time zone | Startup time of a catchup task      |

- `pg_stat_get_wal_senders()`

Description: Displays statistics about replication sending threads on all WALs on the current DN.

Return type: record

The following information is returned:

**Table 6-12** `pg_stat_get_wal_senders()` fields

| Field      | Type    | Description                         |
|------------|---------|-------------------------------------|
| pid        | bigint  | Thread PID                          |
| sender_pid | integer | Current sender PID                  |
| local_role | text    | Local role                          |
| peer_role  | text    | Peer role                           |
| peer_state | text    | Peer status                         |
| state      | text    | Current sender's replication status |

| Field                      | Type                     | Description                                                                                             |
|----------------------------|--------------------------|---------------------------------------------------------------------------------------------------------|
| catchup_start              | timestamp with time zone | Startup time of a catchup task                                                                          |
| catchup_end                | timestamp with time zone | End time of a catchup task                                                                              |
| sender_sent_location       | text                     | Location where the sender sends LSNs                                                                    |
| sender_write_location      | text                     | Location where the sender writes LSNs                                                                   |
| sender_flush_location      | text                     | Location where the sender flushes LSNs                                                                  |
| sender_replay_location     | text                     | Location where the sender replays LSNs                                                                  |
| receiver_received_location | text                     | Location where the receiver receives LSNs                                                               |
| receiver_write_location    | text                     | Location where the receiver writes LSNs                                                                 |
| receiver_flush_location    | text                     | Location where the receiver flushes LSNs                                                                |
| receiver_replay_location   | text                     | Location where the receiver replays LSNs                                                                |
| sync_percent               | text                     | Specifies the synchronization percentage.                                                               |
| sync_state                 | text                     | Synchronization state (asynchronous duplication, synchronous duplication, or potential synchronization) |
| sync_priority              | integer                  | Priority of synchronous duplication ( <b>0</b> indicates asynchronous)                                  |
| sync_most_available        | text                     | Whether to block the active node when the synchronization on the standby node fails                     |
| channel                    | text                     | WALSender channel information                                                                           |

- `pg_stat_get_wal_receiver()`

Description: Displays statistics about replication receiving threads on all WALs on the current DN.

Return type: record

The following information is returned:

**Table 6-13** pg\_stat\_get\_wal\_receiver()

| Field                      | Type    | Description                               |
|----------------------------|---------|-------------------------------------------|
| receiver_pid               | integer | Current receiver PID                      |
| local_role                 | text    | Local role                                |
| peer_role                  | text    | Peer role                                 |
| peer_state                 | text    | Peer status                               |
| state                      | text    | Current receiver's replication status     |
| sender_sent_location       | text    | Location where the sender sends LSNs      |
| sender_write_location      | text    | Location where the sender writes LSNs     |
| sender_flush_location      | text    | Location where the sender flushes LSNs    |
| sender_replay_location     | text    | Location where the sender replays LSNs    |
| receiver_received_location | text    | Location where the receiver receives LSNs |
| receiver_write_location    | text    | Location where the receiver writes LSNs   |
| receiver_flush_location    | text    | Location where the receiver flushes LSNs  |
| receiver_replay_location   | text    | Location where the receiver replays LSNs  |
| sync_percent               | text    | Specifies the synchronization percentage. |
| channel                    | text    | WALReceiver channel information           |

- pg\_stat\_get\_stream\_replications()  
Description: Displays information about all replication statistics on the current DN.  
Return type: record  
The following information is returned:

**Table 6-14** pg\_stat\_get\_stream\_replications()

| Field      | Type | Description |
|------------|------|-------------|
| local_role | text | Local role  |

| Field              | Type    | Description           |
|--------------------|---------|-----------------------|
| static_connections | integer | Connection statistics |
| db_state           | text    | Database status       |
| detail_information | text    | Detail information    |

## 6.25.9 Other Functions

- pgxc\_pool\_check()**  
Description: Checks whether the connection data buffered in the pool is consistent with **pgxc\_node**.  
Return type: Boolean
- pgxc\_pool\_reload()**  
Description: Updates the connection information buffered in the pool.  
Return type: Boolean
- pgxc\_lock\_for\_backup()**  
Description: Locks the cluster before backup. Backup is performed to restore data on new nodes.  
Return type: Boolean

### NOTE

- pgxc\_lock\_for\_backup** locks a cluster before **gs\_dump** or **gs\_dumpall** is used to back up the cluster. After a cluster is locked, operations changing the system structure are not allowed. This function does not affect DML statements.
- pg\_pool\_validate(clear boolean, co\_node\_name cstring)**  
Description: Clears invalid backend threads on a CN. (These backend threads hold invalid pooler connections to standby DN.)  
Return type: record
- pg\_nodes\_memory()**  
Description: Queries for the memory usage of all nodes.  
Return type: record
- table\_skewness(text)**  
Description: Queries for the percentage of table data among all nodes.  
Parameter: Indicates that the type of the name of the to-be-queried table is text.  
Return type: record
- table\_distribution(schemaname text, tablename text)**  
Description: Queries for the storage space occupied by a specified table on each node.  
Parameter: Indicates that the types of the schema name and table name for the table to be queried are both text.  
Return type: record

 **NOTE**

- To query for the storage distribution of a specified table by using this function, you must have the **SELECT** permission for the table.
- The performance of **table\_distribution** is better than that of **table\_skewness**. Especially in a large cluster with a large amount of data, **table\_distribution** is recommended.
- When you use **table\_distribution** and want to view the space usage, you can use **dnsize** or **(sum(dnsize) over ())** to view the percentage.

- **table\_distribution()**

Description: Queries for the storage distribution of all tables in the current database.

Return type: record

 **NOTE**

- This function involves the query for information about all tables in the database. To execute this function, you must have the administrator rights.
  - Based on the **table\_distribution()** function, GaussDB(DWS) provides the **PGXC\_GET\_TABLE\_SKEWNESS** view as an alternative way to query for data skew. You are advised to use this view when the number of tables in the database is less than 10000.
- **plan\_seed**  
Description: Obtains the seed value of the previous query statement (internal use).  
Return type: int
  - **pg\_stat\_get\_env**  
Description: Obtains the environment variable information about the current node.  
Return type: record
  - **pg\_stat\_get\_thread**  
Description: Provides information about the status of all threads under the current node.  
Return type: record
  - **pgxc\_get\_os\_threads**  
Description: Provides information about the status of threads under all normal nodes in a cluster.  
Return type: record
  - **pg\_stat\_get\_sql\_count**  
Description: Provides statistics on the number of **SELECT/UPDATE/INSERT/DELETE/MERGE INTO** statements executed by all users on the current node, response time, and the number of DDL, DML, and DCL statements.  
Return type: record
  - **pgxc\_get\_sql\_count**  
Description: Provides statistics on the number of **SELECT/UPDATE/INSERT/DELETE/MERGE INTO** statements executed by all users on all nodes of the current cluster, response time, and the number of DDL, DML, and DCL statements.

- Return type: record
- `pgxc_get_workload_sql_count`  
Description: Provides statistics on the number of **SELECT/UPDATE/INSERT/DELETE** statements executed in all workload Cgroup on all CNs of the current cluster and the number of DDL, DML, and DCL statements.  
Return type: record
  - `pgxc_get_workload_sql_elapse_time`  
Description: Provides statistics on response time of **SELECT/UPDATE/INSERT/DELETE** statements executed in all workload Cgroup on all CNs of the current cluster.  
Return type: record
  - `get_instr_unique_sql`  
Description: Provides information about Unique SQL statistics collected on the current node. If the node is a CN, the system returns the complete information about the Unique SQL statistics collected on the CN. That is, the system collects and summarizes the information about the Unique SQL statistics on other CNs and DNs. If the node is a DN, the Unique SQL statistics on the DN is returned. For details, see the `GS_INSTR_UNIQUE_SQL` view.  
Return type: record
  - `reset_instr_unique_sql(cstring, cstring, INT8)`  
Description: Clears collected Unique SQL statistics. The input parameters are described as follows:
    - **GLOBAL/LOCAL**: Data is cleared from all nodes or the current node.
    - **ALL/BY\_USERID/BY\_CNID/BY\_GUC**: **ALL** indicates that all data is cleared. **BY\_USERID/BY\_CNID** indicates that data is cleared by **USERID** or **CNID**. **BY\_GUC** indicates that the clearance operation is caused by the decrease of the value of the GUC parameter `instr_unique_sql_count`.
    - The third parameter corresponds to the second parameter. The parameter is invalid for **ALL/BY\_GUC**.Return type: bool
  - `pgxc_get_instr_unique_sql`  
Description: Provides complete information about Unique SQL statistics collected on all CNs in a cluster.  
Return type: record
  - `pgxc_get_node_env`  
Description: Provides the environment variable information about all nodes in a cluster.  
Return type: record
  - `gs_switch_relfilenode`  
Description: Exchanges meta information of two tables or partitions. (This is only used for the redistribution tool. An error message is displayed when the function is directly used by users).  
Return type: int
  - `copy_error_log_create()`  
Description: Creates the error table (**public.pgxc\_copy\_error\_log**) required for creating the **COPY FROM** error tolerance mechanism.

Return type: Boolean

 NOTE

- This function attempts to create the **public.pgxc\_copy\_error\_log** table. For details about the table, see [Table 6-15](#).
- Create the B-tree index on the **relname** column and execute **REVOKE ALL on public.pgxc\_copy\_error\_log FROM public** to manage permissions for the error table (the permissions are the same as those of the **COPY** statement).
- **public.pgxc\_copy\_error\_log** is a row-store table. Therefore, this function can be executed and **COPY FROM** error tolerance is available only when row-store tables can be created in the cluster. Row-store tables cannot be created in the cluster if **enable\_hadoop\_env** is set to **on** ().
- Same as the error table and the **COPY** statement, the function requires **sysadmin** or higher permissions.
- If the **public.pgxc\_copy\_error\_log** table or the **copy\_error\_log\_relname\_idx** index already exists before the function creates it, the function will report an error and roll back.

**Table 6-15** Error table public.pgxc\_copy\_error\_log

| Column    | Type                     | Description                                                                                                                                         |
|-----------|--------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------|
| relname   | varchar                  | Table name in the form of <i>Schema name.Table name</i>                                                                                             |
| begintime | timestamp with time zone | Time when a data format error was reported                                                                                                          |
| filename  | character varying        | Name of the source data file where a data format error occurs                                                                                       |
| rownum    | bigint                   | Number of the row where a data format error occurs in a source data file                                                                            |
| rawrecord | text                     | Raw record of a data format error in the source data file To prevent a field from being too long, the length of the field cannot exceed 1024 bytes. |
| detail    | text                     | Error details                                                                                                                                       |

- `pv_compute_pool_workload()`  
Description: Provides the current load information about computing Node Groups on cloud.  
Return type: record

## 6.26 Data Redaction Functions



## 6.26.1 Creating a Redact Policy

Create a redact policy by using a function. GaussDB(DWS) provides the following function interfaces:

```
DBMS_REDACT.ADD_POLICY (  
object_schema      IN name,  
object_name        IN name,  
policy_name        IN name,  
column_name        IN name default NULL,  
function_type      IN int2 default 1,  
function_parameters IN Text default NULL,  
expression         IN Text,  
enable             IN BOOLEAN default TRUE,  
regexp_pattern     IN Text default NULL,  
regexp_replace_string IN Text default NULL,  
regexp_position    IN int4 default 1,  
regexp_occurrence  IN int4 default 0,  
regexp_match_parameter IN Text default NULL,  
policy_description IN Text default NULL,  
column_description IN Text default NULL  
)
```

In the preceding information:

- **object\_schema**: specifies the schema of the object to be redacted.
- **object\_name**: specifies the name of the table to be redacted.
- **policy\_name**: specifies the name of the redact policy.
- **column\_name**: specifies the name of the column to be redacted. If this parameter is left empty, only a policy is created, and **redact\_column** is not created. In this case, no redact column takes effect.
- **function\_type**: specifies type of the redaction function. The value can be **NONE** or **FULL**. The default value is **FULL**, which takes effect on redact columns.
  - **NONE**: No redact policy is configured for the column object.
  - **FULL**: The column object is redacted to a fixed value.
- **function\_parameters**: specifies the redact function parameters. This parameter is valid only when **function\_type** is set to **PARTIAL**.
- **expression**: The redaction takes effect only when **expression** is set to **true**. If the redaction takes effect for all users, set **expression** to **1=1**. Otherwise, use the **sys\_context** function expression. **expression** can be used as the judgment basis for **SYS\_CONTEXT**.
  - The expression of the **CURRENT\_USER** parameter is supported. The policy takes effect for user filtering.  
function\_parameters => 'SYS\_CONTEXT("USERENV", "CURRNET\_USER") != "HR"'
  - Multiple expressions can be connected using **AND**, **OR**, and **IN**.  
function\_parameters => 'SYS\_CONTEXT("USERENV", "CURRNET\_USER") = "HR" and SYS\_CONTEXT("USERENV", "USER\_GROUP") = "GAUSS"'
- **enable**: indicates whether the policy takes effect after the action is created. The default value is **TRUE**.
- **regexp\_pattern**: specifies the regular pattern. This parameter is valid only when **function\_type** is set to **REGEXP**.
- **regexp\_replace\_string**: specifies the regular replacement string.
- **regexp\_position**: specifies the start position for matching the regular expression source string.

- **regexp\_occurrence**: replaces the *n*th matched character string. The value **0** indicates that all matched character strings are replaced.
- **regexp\_match\_parameter**: specifies the regular expression match parameter.
- **policy\_description**: specifies the policy description.
- **column\_description**: If **column\_name** is specified, the description of the column can be added.

## 6.26.2 Modifying a Redact Policy

Modify a created redact policy, for example, add columns to be redacted to a table policy, delete the columns to be redacted, modify the expression, and modify the column redaction type. The following function interfaces are provided:

```
DBMS_REDACT.ALTER_POLICY(  
object_schema      IN name,  
object_name        IN name,  
policy_name        IN name,  
action              IN int2 default 1,  
column_name        IN name default NULL,  
function_type       IN int2 default 1,  
function_parameters IN Text default NULL,  
expression          IN Text default NULL,  
regexp_pattern      IN Text default NULL,  
regexp_replace_string IN Text default NULL,  
regexp_position     IN int4 default 1,  
regexp_occurrence  IN int4 default 0,  
regexp_match_parameter IN Text default NULL,  
policy_description  IN Text default NULL,  
column_description  IN Text default NULL  
)
```

Except the **action** parameter, other interfaces are the same as those in **ADD\_POLICY**.

- **action**: specifies the action of **ALTER\_POLICY**. The value of **ALTER\_POLICY** ranges from **1** to **6**. The meanings are as follows:
  - a. **ADD\_COLUMN**: adds a column to a policy. **function\_type** must be specified.
  - b. **DROP\_COLUMN**: deletes a column from a policy.
  - c. **MODIFY\_EXPRESSION**: modifies the policy judgment expression.
  - d. **MODIFY\_COLUMN**: Modifies **function\_type** or **function\_parameters** of the redacted column.
  - e. **SET\_POLICY\_DESCRIPTION**: sets the policy description.
  - f. **SET\_COLUMN\_DESCRIPTION**: sets the description of the redacted column.

## 6.26.3 Deleting a Redact Policy

Delete a created redact policy. The function interface is as follows:

```
DBMS_REDACT.DROP_POLICY(  
object_schema      IN Text,  
object_name        IN Text,  
policy_name        IN Text  
)
```

## 6.26.4 Enabling the Redact Function for a Table Object

After a redact policy is created in a data table, the policy takes effect only when the policy of the data table is enabled (that is, **enable** is set to **true**). By default, the policy is enabled.

The function interface for enabling a redact function of the data table is as follows:

```
DBMS_REDACT.ENABLE_POLICY(  
object_schema      IN Text,  
object_name        IN Text,  
policy_name        IN Text  
)
```

## 6.26.5 Disabling the Redact Function for a Table Object

After a redact policy is created in a data table, the policy of the data table is enabled by default (that is, **enable** is set to **true**). If you do not want to use the redact feature, you can disable the redact switch of the corresponding data table.

The function interface for disabling a redact function of the data table is as follows:

```
DBMS_REDACT.DISABLE_POLICY(  
object_schema      IN Text,  
object_name        IN Text,  
policy_name        IN Text  
)
```

## 6.27 Statistics Information Functions

Statistics information functions are divided into the following two categories: functions that access databases, using the OID of each table or index in a database to mark the database for which statistics are generated; functions that access servers, identified by the server process ID, whose value ranges from 1 to the number of currently active servers.

- `pg_stat_get_db_numbackends(oid)`  
Description: Number of active server processes for a database  
Return type: integer
- `pg_stat_get_db_xact_commit(oid)`  
Description: Number of transactions committed in a database  
Return type: bigint
- `pg_stat_get_db_xact_rollback(oid)`  
Description: Number of transactions rolled back in a database  
Return type: bigint
- `pg_stat_get_db_blocks_fetched(oid)`  
Description: Number of disk blocks fetch requests for a database  
Return type: bigint
- `pg_stat_get_db_blocks_hit(oid)`  
Description: Number of disk block fetch requests found in cache for a database

- Return type: bigint
- `pg_stat_get_db_tuples_returned(oid)`  
Description: Number of tuples returned for a database  
Return type: bigint
  - `pg_stat_get_db_tuples_fetched(oid)`  
Description: Number of tuples fetched for a database  
Return type: bigint
  - `pg_stat_get_db_tuples_inserted(oid)`  
Description: Number of tuples inserted in a database  
Return type: bigint
  - `pg_stat_get_db_tuples_updated(oid)`  
Description: Number of tuples updated in a database  
Return type: bigint
  - `pg_stat_get_db_tuples_deleted(oid)`  
Description: Number of tuples deleted in a database  
Return type: bigint
  - `pg_stat_get_db_conflict_lock(oid)`  
Description: Number of lock conflicts in a database  
Return type: bigint
  - `pg_stat_get_db_deadlocks(oid)`  
Description: Number of deadlocks in a database  
Return type: bigint
  - `pg_stat_get_numscans(oid)`  
Description: Number of sequential row scans done if parameters are in a table or number of index scans done if parameters are in an index  
Return type: bigint
  - `pg_stat_get_tuples_returned(oid)`  
Description: Number of sequential row scans done if parameters are in a table or number of index entries returned if parameters are in an index  
Return type: bigint
  - `pg_stat_get_tuples_fetched(oid)`  
Description: Number of table rows fetched by bitmap scans if parameters are in a table,  
or table rows fetched by simple index scans using the index if parameters are in an index  
Return type: bigint
  - `pg_stat_get_tuples_inserted(oid)`  
Description: Number of rows inserted into table  
Return type: bigint
  - `pg_stat_get_tuples_updated(oid)`  
Description: Number of rows updated in table

- Return type: bigint
- `pg_stat_get_tuples_deleted(oid)`  
Description: Number of rows deleted from table  
Return type: bigint
  - `pg_stat_get_tuples_changed(oid)`  
Description: Total number of inserted, updated, and deleted rows after the table was last analyzed or autoanalyzed  
Return type: bigint
  - `pg_stat_get_tuples_hot_updated(oid)`  
Description: Number of rows HOT-updated in table  
Return type: bigint
  - `pg_stat_get_live_tuples(oid)`  
Description: Number of live rows in table  
Return type: bigint
  - `pg_stat_get_dead_tuples(oid)`  
Description: Number of dead rows in table  
Return type: bigint
  - `pg_stat_get_blocks_fetched(oid)`  
Description: Number of disk block fetch requests for table or index  
Return type: bigint
  - `pg_stat_get_blocks_hit(oid)`  
Description: Number of disk block requests found in cache for table or index  
Return type: bigint
  - `pg_stat_get_partition_tuples_inserted(oid)`  
Description: Number of rows in the corresponding table partition  
Return type: bigint
  - `pg_stat_get_partition_tuples_updated(oid)`  
Description: Number of rows that have been updated in the corresponding table partition  
Return type: bigint
  - `pg_stat_get_partition_tuples_deleted(oid)`  
Description: Number of rows deleted from the corresponding table partition  
Return type: bigint
  - `pg_stat_get_partition_tuples_changed(oid)`  
Description: Total number of inserted, updated, and deleted rows after the table partition was last analyzed or autoanalyzed  
Return type: bigint
  - `pg_stat_get_partition_live_tuples(oid)`  
Description: Number of live rows in a table partition  
Return type: bigint
  - `pg_stat_get_partition_dead_tuples(oid)`

Description: Number of dead rows in a table partition

Return type: bigint

- `pg_stat_get_xact_tuples_inserted(oid)`

Description: Number of tuple inserted into the active subtransactions related to the table.

Return type: bigint

- `pg_stat_get_xact_tuples_deleted(oid)`

Description: Number of deleted tuples in the active subtransactions related to a table

Return type: bigint

- `pg_stat_get_xact_tuples_hot_updated(oid)`

Description: Number of hot updated tuples in the active subtransactions related to a table

Return type: bigint

- `pg_stat_get_xact_tuples_updated(oid)`

Description: Number of updated tuples in the active subtransactions related to a table

Return type: bigint

- `pg_stat_get_xact_partition_tuples_inserted(oid)`

Description: Number of inserted tuples in the active subtransactions related to a table partition

Return type: bigint

- `pg_stat_get_xact_partition_tuples_deleted(oid)`

Description: Number of deleted tuples in the active subtransactions related to a table partition

Return type: bigint

- `pg_stat_get_xact_partition_tuples_hot_updated(oid)`

Description: Number of hot updated tuples in the active subtransactions related to a table partition

Return type: bigint

- `pg_stat_get_xact_partition_tuples_updated(oid)`

Description: Number of updated tuples in the active subtransactions related to a table partition

Return type: bigint

- `pg_stat_get_last_vacuum_time(oid)`

Description: Last time when the autovacuum thread is manually started to clear a table

Return type: timestamptz

- `pg_stat_get_last_autovacuum_time(oid)`

Description: Time of the last vacuum initiated by the autovacuum daemon on this table

Return type: timestamptz

- `pg_stat_get_vacuum_count(oid)`

Description: Number of times a table is manually cleared

Return type: bigint

- `pg_stat_get_autovacuum_count(oid)`

Description: Number of times the autovacuum daemon is started to clear a table

Return type: bigint

- `pg_stat_get_last_analyze_time(oid)`

Description: Last time when a table starts to be analyzed manually or by the autovacuum thread

Return type: timestamptz

- `pg_stat_get_last_autoanalyze_time(oid)`

Description: Time of the last analysis initiated by the autovacuum daemon on this table

Return type: timestamptz

- `pg_stat_get_analyze_count(oid)`

Description: Number of times a table is manually analyzed

Return type: bigint

- `pg_stat_get_autoanalyze_count(oid)`

Description: Number of times the autovacuum daemon analyzes a table

Return type: bigint

- `pg_total_autovac_tuples(bool,bool)`

Description: Returns tuple records related to the total autovac, such as **nodename**, **nspname**, **relname**, and tuple IUDs. The input parameters specify whether to query **relation** and **local** information, respectively.

Return type: setofrecord

- `pg_autovac_status(oid)`

Description: Returns autovac information, such as **nodename**, **nspname**, **relname**, **analyze**, **vacuum**, thresholds of **analyze** and **vacuum**, and the number of analyzed or vacuumed tuples.

Return type: setofrecord

- `pg_autovac_timeout(oid)`

Description: Returns the number of consecutive timeouts during the autovac operation on a table. If the table information is invalid or the node information is abnormal, **NULL** will be returned.

Return type: bigint

- `pg_autovac_coordinator(oid)`

Description: Returns the name of the CN performing the autovac operation on a table. If the table information is invalid or the node information is abnormal, **NULL** will be returned.

Return type: text

- `pgxc_get_wlm_session_info_bytime(text, timestamp without time zone, timestamp without time zone, int)`

Description: The query performance of the PGXC\_WLM\_SESSION\_INFO view is poor if the view contains a large number of records. In this case, you are

advised to use this function to filter the query. The input parameters are *time column* (**start\_time** or **finish\_time**), *start time*, *end time*, and *maximum number of records returned for each CN*. The return result is a subset of records in the `GS_WLM_SESSION_HISTORY` view.

Return type: setofrecord

- `pgxc_get_wlm_current_instance_info(text, int default null)`  
Description: Queries the current resource usage of each node in the cluster on the CN and reads the data that is not stored in the `GS_WLM_INSTANCE_HISTORY` system catalog in the memory. The input parameters are the node name (**ALL**, **C**, **D**, or *instance name*) and the maximum number of records returned by each node. The returned value is **GS\_WLM\_INSTANCE\_HISTORY**.  
Return type: setofrecord
- `pgxc_get_wlm_history_instance_info(text, TIMESTAMP, TIMESTAMP, int default null)`  
Description: Queries the historical resource usage of each cluster node on the CN node and reads data from the **GS\_WLM\_INSTANCE\_HISTORY** system catalog. The input parameters are as follows: node name (**ALL**, **C**, **D**, or *instance name*), start time, end time, and maximum number of records returned for each instance. The returned value is **GS\_WLM\_INSTANCE\_HISTORY**.  
Return type: setofrecord
- `pg_stat_get_last_data_changed_time(oid)`  
Description: Returns the time when **INSERT**, **UPDATE**, **DELETE**, or **EXCHANGE/TRUNCATE/DROP PARTITION** was performed last time on a table. The data in the **last\_data\_changed** column of the `PG_STAT_ALL_TABLES` view is calculated by using this function. The performance of obtaining the last modification time by using the view is poor when the table has a large amount of data. In this case, you are advised to use the function.  
Return type: timestamptz
- `pg_stat_set_last_data_changed_time(oid)`  
Description: Manually changes the time when **INSERT**, **UPDATE**, **DELETE**, or **EXCHANGE/TRUNCATE/DROP PARTITION** was performed last time.  
Return type: void
- `pg_backend_pid()`  
Description: Thread ID of the server thread attached to the current session  
Return type: integer
- `pv_session_time()`  
Description: Collects statistics on the running time of each session thread on the current node and the time consumed in each execution phase.  
Return type: record
- `pv_instance_time()`  
Description: Collects statistics on the running time of the current node and the time consumed in each execution phase.  
Return type: record



- pg\_stat\_get\_activity(integer)**  
 Description: Returns a record about the backend with the specified PID. A record for each active backend in the system is returned if **NULL** is specified. The return result is a subset of records (excluding the **connection\_info** column) in the PG\_STAT\_ACTIVITY view.  
 Return type: setofrecord
- pg\_stat\_get\_activity\_with\_conninfo(integer)**  
 Description: Returns a record about the backend with the specified PID. A record for each active backend in the system is returned if **NULL** is specified. The return result is a subset of records in the PG\_STAT\_ACTIVITY view.  
 Return type: setofrecord
- pg\_user\_iostat(text)**  
 Description: Displays the I/O load management information about the job currently executed by the user.  
 Return type: record

The following table describes return fields.

| Name          | Type | Description                                                                                                                                        |
|---------------|------|----------------------------------------------------------------------------------------------------------------------------------------------------|
| userid        | oid  | User ID                                                                                                                                            |
| min_curr_iops | int4 | Minimum I/O of the current user across DNs. The IOPS is counted by ones for column storage and by thousands for row storage.                       |
| max_curr_iops | int4 | Maximum I/O of the current user across DNs. The IOPS is counted by ones for column storage and by thousands for row storage.                       |
| min_peak_iops | int4 | Minimum peak I/O of the current user across DNs. The IOPS is counted by ones for column storage and by thousands for row storage.                  |
| max_peak_iops | int4 | Maximum peak I/O of the current user across DNs. The IOPS is counted by ones for column storage and by thousands for row storage.                  |
| io_limits     | int4 | <b>io_limits</b> set for the resource pool specified by the user. The IOPS is counted by ones for column storage and by thousands for row storage. |
| io_priority   | text | <b>io_priority</b> set for the user. The IOPS is counted by ones for column storage and by thousands for row storage.                              |

- pg\_stat\_get\_function\_calls(oid)**  
 Description: Number of times the function has been called  
 Return type: bigint
- pg\_stat\_get\_function\_time(oid)**

Description: Total wall clock time spent in the function, in microseconds. Includes the time spent in functions called by this one.

Return type: bigint

- `pg_stat_get_function_self_time(oid)`

Description: Time spent in only this function. Time spent in called functions is excluded.

Return type: bigint

- `pg_stat_get_backend_idset()`

Description: Set of currently active server process numbers (from 1 to the number of active server processes)

Return type: setofinteger

- `pg_stat_get_backend_pid(integer)`

Description: Thread ID of the given server thread

Return type: bigint

- `pg_stat_get_backend_dbid(integer)`

Description: ID of the database connected to the given server process

Return type: OID

- `pg_stat_get_backend_userid(integer)`

Description: User ID of the given server process

Return type: OID

- `pg_stat_get_backend_activity(integer)`

Description: Active command of the given server process, but only if the current user is a system administrator or the same user as that of the session being queried and **track\_activities** is on

Return type: text

- `pg_stat_get_backend_waiting(integer)`

Description: True if the given server process is waiting for a lock, but only if the current user is a system administrator or the same user as that of the session being queried and **track\_activities** is on

Return type: Boolean

- `pg_stat_get_backend_activity_start(integer)`

Description: The time at which the given server process's currently executing query was started, but only if the current user is a system administrator or the same user as that of the session being queried and **track\_activities** is on

Return type: timestamp with time zone

- `pg_stat_get_backend_xact_start(integer)`

Description: The time at which the given server process's currently executing transaction was started, but only if the current user is a system administrator or the same user as that of the session being queried and **track\_activities** is on

Return type: timestamp with time zone

- `pg_stat_get_backend_start(integer)`

Description: The time at which the given server process was started, or **NULL** if the current user is neither a system administrator nor the same user as that of the session being queried

Return type: timestamp with time zone

- `pg_stat_get_backend_client_addr(integer)`

Description: IP address of the client connected to the given server process.

If the connection is over a Unix domain socket, or if the current user is neither a system administrator nor the same user as that of the session being queried, **NULL** will be returned.

Return type: inet

- `pg_stat_get_backend_client_port(integer)`

Description: TCP port number of the client connected to the given server process

If the connection is over a Unix domain socket, **-1** will be returned. If the current user is neither a system administrator nor the same user as that of the session being queried, **NULL** will be returned.

Return type: integer

- `pg_stat_get_bgwriter_timed_checkpoints()`

Description: The number of times the background writer has started timed checkpoints (because the **checkpoint\_timeout** time has expired)

Return type: bigint

- `pg_stat_get_bgwriter_requested_checkpoints()`

Description: The number of times the background writer has started checkpoints based on requests from the backend because **checkpoint\_segments** has been exceeded or the **CHECKPOINT** command has been executed

Return type: bigint

- `pg_stat_get_bgwriter_buf_written_checkpoints()`

Description: The number of buffers written by the background writer during checkpoints

Return type: bigint

- `pg_stat_get_bgwriter_buf_written_clean()`

Description: The number of buffers written by the background writer for routine cleaning of dirty pages

Return type: bigint

- `pg_stat_get_bgwriter_maxwritten_clean()`

Description: The number of times the background writer has stopped its cleaning scan because it has written more buffers than specified in the **bgwriter\_lru\_maxpages** parameter

Return type: bigint

- `pg_stat_get_buf_written_backend()`

Description: The number of buffers written by the backend because they needed to allocate a new buffer

Return type: bigint

- `pg_stat_get_buf_alloc()`  
Description: The total number of buffer allocations  
Return type: bigint
- `pg_stat_clear_snapshot()`  
Description: Discards the current statistics snapshot.  
Return type: void
- `pg_stat_reset()`  
Description: Resets all statistics counters for the current database to zero (requires system administrator permissions).  
Return type: void
- `pg_stat_reset_shared(text)`  
Description: Resets all statistics counters for the current database in each node in a shared cluster to zero (requires system administrator permissions).  
Return type: void
- `pg_stat_reset_single_table_counters(oid)`  
Description: Resets statistics for a single table or index in the current database to zero (requires system administrator permissions).  
Return type: void
- `pg_stat_reset_single_function_counters(oid)`  
Description: Resets statistics for a single function in the current database to zero (requires system administrator permissions).  
Return type: void
- `pg_stat_session_cu(int, int, int)`  
Description: Obtains the compression unit (CU) hit statistics of sessions running on the current node.  
Return type: record
- `gs_get_stat_session_cu(text, int, int, int)`  
Description: Obtains the CU hit statistics of all sessions running in a cluster.  
Return type: record
- `gs_get_stat_db_cu(text, text, int, int, int)`  
Description: Obtains the CU hit statistics of a database in a cluster.  
Return type: record
- `pg_stat_get_cu_mem_hit(oid)`  
Description: Obtains the number of CU memory hits of a column storage table in the current database of the current node.  
Return type: bigint
- `pg_stat_get_cu_hdd_sync(oid)`  
Description: Obtains the times CU is synchronously read from a disk by a column storage table in the current database of the current node.  
Return type: bigint
- `pg_stat_get_cu_hdd_asyn(oid)`  
Description: Obtains the times CU is asynchronously read from a disk by a column storage table in the current database of the current node.

Return type: bigint

- `pg_stat_get_db_cu_mem_hit(oid)`

Description: Obtains the CU memory hit in a database of the current node.

Return type: bigint

- `pg_stat_get_db_cu_hdd_sync(oid)`

Description: Obtains the times CU is synchronously read from a disk by a database of the current node.

Return type: bigint

- `pg_stat_get_db_cu_hdd_asyn(oid)`

Description: Obtains the times CU is asynchronously read from a disk by a database of the current node.

Return type: bigint

- `pgxc_fenced_udf_process()`

Description: Shows the number of UDF Master and Work processes.

Return type: record

- `pgxc_terminate_all_fenced_udf_process()`

Description: Kills all UDF Work processes.

Return type: bool

- `GS_ALL_NODEGROUP_CONTROL_GROUP_INFO(text)`

Description: Provides Cgroup information for all logical clusters. Before invoking this function, you need to specify the name of a logical cluster to be queried. For example, to query the Cgroup information for the **installation** logical cluster, run the following command:

```
SELECT * FROM GS_ALL_NODEGROUP_CONTROL_GROUP_INFO('installation')
```

Return type: record

The following table describes return fields.

| Name     | Type   | Description                                                          |
|----------|--------|----------------------------------------------------------------------|
| name     | text   | Name of a Cgroup                                                     |
| type     | text   | Type of the Cgroup                                                   |
| gid      | bigint | Cgroup ID                                                            |
| classgid | bigint | ID of the <b>Class</b> Cgroup where a <b>Workload</b> Cgroup belongs |
| class    | text   | <b>Class</b> Cgroup                                                  |
| workload | text   | <b>Workload</b> Cgroup                                               |
| shares   | bigint | CPU quota allocated to a Cgroup                                      |
| limits   | bigint | Limit of CPUs allocated to a Cgroup                                  |
| wdlevel  | bigint | <b>Workload</b> Cgroup level                                         |
| cpucores | text   | Usage of CPU cores in a Cgroup                                       |

- `gs_get_nodegroup_tablecount(name)`  
Description: Total number of user tables in all the databases in a logical cluster  
Return type: integer
- `pgxc_max_datanode_size(name)`  
Description: Maximum disk space occupied by database files in all the DN of a logical cluster. The unit is byte.  
Return type: bigint
- `gs_check_logic_cluster_consistency()`  
Description: Checks whether the system information of all logical clusters in the system is consistent. If no record is returned, the information is consistent. Otherwise, the Node Group information on CNs and DN in the logical cluster is inconsistent. This function cannot be invoked during redistribution in a scale-in or scale-out.  
Return type: record
- `gs_check_tables_distribution()`  
Description: Checks whether the user table distribution in the system is consistent. If no record is returned, table distribution is consistent. This function cannot be invoked during redistribution in a scale-in or scale-out.  
Return type: record
- `pg_stat_bad_block(text, int, int, int, int, int, timestamp with time zone, timestamp with time zone)`  
Description: Obtains damage information about pages or CUs after the current node is started.  
Return type: record
- `pgxc_stat_bad_block(text, int, int, int, int, int, timestamp with time zone, timestamp with time zone)`  
Description: Obtains damage information about pages or CUs after all the nodes in the cluster are started.  
Return type: record
- `pg_stat_bad_block_clear()`  
Description: Deletes the page and CU damage information that is read and recorded on the node. (System administrator rights are required.)  
Return type: void
- `pgxc_stat_bad_block_clear`  
Description: Deletes the page and CU damage information that is read and recorded on all the nodes in the cluster. (System administrator rights are required.)  
Return type: void
- `gs_respool_exception_info(pool text)`  
Description: Queries for the query rule of a specified resource pool.  
Return type: record
- `gs_control_group_info(pool text)`  
Description: Queries for information about Cgroups associated with a resource pool.

Return type: record

The following information is displayed:

| Attribute | Value               | Description                                             |
|-----------|---------------------|---------------------------------------------------------|
| name      | class_a:workload_a1 | Class name and workload name                            |
| class     | class_a             | Class Cgroup name                                       |
| workload  | workload_a1         | Workload Cgroup name                                    |
| type      | DEFWD               | Cgroup type (Top, CLASS, BAKWD, DEFWD, and TSWD)        |
| gid       | 87                  | Cgroup ID                                               |
| shares    | 30                  | Percentage of CPU resources to those on the parent node |
| limits    | 0                   | Percentage of CPU cores to those on the parent node     |
| rate      | 0                   | Allocation raio in Timeshare                            |
| cpucores  | 0-3                 | Number of CPU cores                                     |

- `gs_wlm_user_resource_info(name text)`

Description: Queries for a user's resource quota and resource usage.

Return type: record

For example:

The function `pg_backend_pid` shows the ID of the current server thread.

```
SELECT pg_backend_pid();
pg_backend_pid
-----
139706243217168
(1 row)
```

The function `pg_stat_get_backend_pid` shows the ID of a given server thread.

```
SELECT pg_stat_get_backend_pid(1);
pg_stat_get_backend_pid
-----
139706243217168
(1 row)
```

## 6.28 Trigger Functions

- `pg_get_triggerdef(oid)`

Description: Obtains the definition information of a trigger.

Parameter: OID of the trigger to be queried

Return type: text

- `pg_get_triggerdef(oid, boolean)`

Description: Obtains the definition information of a trigger.

Parameter: OID of the trigger to be queried and whether it is displayed in pretty mode

Return type: text



# 7 Expressions

---

## 7.1 Simple Expressions

### Logical Expressions

[Logical Operators](#) lists the operators and calculation rules of logical expressions.

### Comparative Expressions

[Comparison Operators](#) lists the common comparative operators.

In addition to comparative operators, you can also use the following sentence structure:

- BETWEEN operator  
**a BETWEEN x AND y** is equivalent to **a >= x AND a <= y**.  
**a NOT BETWEEN x AND y** is equivalent to **a < x OR a > y**.
- To check whether a value is null, use:  
expression IS NULL  
expression IS NOT NULL  
or an equivalent (non-standard) sentence structure:  
expression ISNULL  
expression NOTNULL

---

#### NOTICE

Do not write **expression=NULL** or **expression<>(!=)NULL**, because **NULL** represents an unknown value, and these expressions cannot determine whether two unknown values are equal.

---

### Examples

```
SELECT 2 BETWEEN 1 AND 3 AS RESULT;  
result
```

```
-----  
t  
(1 row)  
  
SELECT 2 >= 1 AND 2 <= 3 AS RESULT;  
result  
-----  
t  
(1 row)  
  
SELECT 2 NOT BETWEEN 1 AND 3 AS RESULT;  
result  
-----  
f  
(1 row)  
  
SELECT 2 < 1 OR 2 > 3 AS RESULT;  
result  
-----  
f  
(1 row)  
  
SELECT 2+2 IS NULL AS RESULT;  
result  
-----  
f  
(1 row)  
  
SELECT 2+2 IS NOT NULL AS RESULT;  
result  
-----  
t  
(1 row)  
  
SELECT 2+2 ISNULL AS RESULT;  
result  
-----  
f  
(1 row)  
  
SELECT 2+2 NOTNULL AS RESULT;  
result  
-----  
t  
(1 row)  
  
SELECT 2+2 IS DISTINCT FROM NULL AS RESULT;  
result  
-----  
t  
(1 row)  
  
SELECT 2+2 IS NOT DISTINCT FROM NULL AS RESULT;  
result  
-----  
f  
(1 row)
```

## 7.2 Conditional Expressions

Data that meets the requirements specified by conditional expressions are filtered during SQL statement execution.

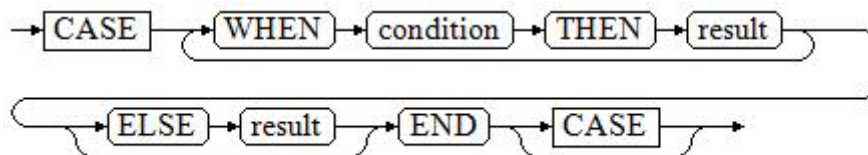
Conditional expressions include the following types:

- CASE

**CASE** expressions are similar to the **CASE** statements in other coding languages.

**Figure 7-1** shows the syntax of a **CASE** expression.

**Figure 7-1** case::=



A **CASE** clause can be used in a valid expression. **condition** is an expression that returns a value of Boolean type.

- If the result is **true**, the result of the **CASE** expression is the required result.
- If the result is false, the following **WHEN** or **ELSE** clauses are processed in the same way.
- If every **WHEN condition** is false, the result of the expression is the result of the **ELSE** clause. If the **ELSE** clause is omitted and has no match condition, the result is NULL.

For example:

```

CREATE TABLE tpcds.case_when_t1(CW_COL1 INT) DISTRIBUTE BY HASH (CW_COL1);
INSERT INTO tpcds.case_when_t1 VALUES (1), (2), (3);

SELECT * FROM tpcds.case_when_t1;
a
---
1
2
3
(3 rows)

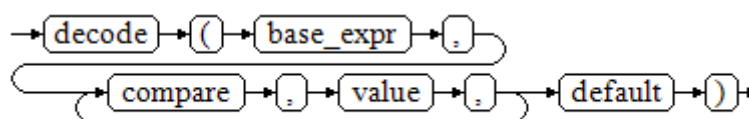
SELECT CW_COL1, CASE WHEN CW_COL1=1 THEN 'one' WHEN CW_COL1=2 THEN 'two' ELSE 'other'
END FROM tpcds.case_when_t1;
a | case
---+-----
3 | other
1 | one
2 | two
(3 rows)

DROP TABLE tpcds.case_when_t1;
  
```

- **DECODE**

**Figure 7-2** shows the syntax of a **DECODE** expression.

**Figure 7-2** decode::=



Compare each following **compare(n)** with **base\_expr**, **value(n)** is returned if a **compare(n)** matches the **base\_expr** expression. If **base\_expr** does not match each **compare(n)**, the default value is returned.

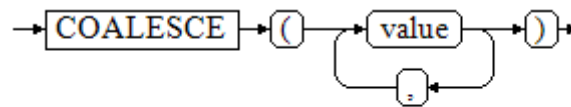
**Conditional Expression Functions** describes the examples.

```
SELECT DECODE('A','A',1,'B',2,0);
-----
1
(1 row)
```

- COALESCE

**Figure 7-3** shows the syntax of a **COALESCE** expression.

**Figure 7-3** coalesce::=



**COALESCE** returns its first non-NULL value. If all the arguments are NULL, return **NULL**. This value is replaced by the default value when data is displayed. Like a **CASE** expression, **COALESCE** only evaluates the parameters that are needed to determine the result. That is, parameters to the right of the first non-null parameter are not evaluated.

**Examples**

```
CREATE TABLE tpcds.c_tabl(description varchar(10), short_description varchar(10), last_value
varchar(10))
DISTRIBUTE BY HASH (last_value);

INSERT INTO tpcds.c_tabl VALUES('abc', 'efg', '123');
INSERT INTO tpcds.c_tabl VALUES(NULL, 'efg', '123');
INSERT INTO tpcds.c_tabl VALUES(NULL, NULL, '123');

SELECT description, short_description, last_value, COALESCE(description, short_description, last_value)
FROM tpcds.c_tabl ORDER BY 1, 2, 3, 4;
-----
description | short_description | last_value | coalesce
-----
abc         | efg              | 123       | abc
           | efg              | 123       | efg
           |                  | 123       | 123
(3 rows)

DROP TABLE tpcds.c_tabl;
```

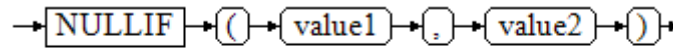
If **description** is not **NULL**, the value of **description** is returned. Otherwise, parameter **short\_description** is calculated. If **short\_description** is not **NULL**, the value of **short\_description** is returned. Otherwise, parameter **last\_value** is calculated. If **last\_value** is not **NULL**, the value of **last\_value** is returned. Otherwise, **none** is returned.

```
SELECT COALESCE(NULL,'Hello World');
-----
coalesce
-----
Hello World
(1 row)
```

- NULLIF

**Figure 7-4** shows the syntax of a **NULLIF** expression.

**Figure 7-4** nullif::=



Only if **value1** is equal to **value2** can **NULLIF** return the **NULL** value. Otherwise, **value1** is returned.

Examples

```
CREATE TABLE tpcds.null_if_t1 (
  NI_VALUE1 VARCHAR(10),
  NI_VALUE2 VARCHAR(10)
) DISTRIBUTED BY HASH (NI_VALUE1);

INSERT INTO tpcds.null_if_t1 VALUES('abc', 'abc');
INSERT INTO tpcds.null_if_t1 VALUES('abc', 'efg');

SELECT NI_VALUE1, NI_VALUE2, NULLIF(NI_VALUE1, NI_VALUE2) FROM tpcds.null_if_t1 ORDER BY 1,
2, 3;

ni_value1 | ni_value2 | nullif
-----+-----
abc      | abc      |
abc      | efg      | abc
(2 rows)
DROP TABLE tpcds.null_if_t1;
```

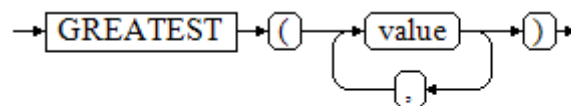
If **value1** is equal to **value2**, **NULL** is returned. Otherwise, **value1** is returned.

```
SELECT NULLIF('Hello','Hello World');
nullif
-----
Hello
(1 row)
```

- GREATEST (maximum value) and LEAST (minimum value)

**Figure 7-5** shows the syntax of a **GREATEST** expression.

**Figure 7-5** greatest::=

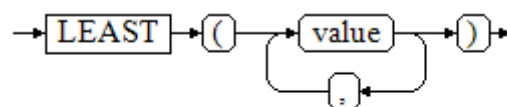


You can select the maximum value from any numerical expression list.

```
SELECT greatest(9000,155555,2.01);
greatest
-----
155555
(1 row)
```

**Figure 7-6** shows the syntax of a **LEAST** expression.

**Figure 7-6** least::=



You can select the minimum value from any numerical expression list.

Each of the preceding numeric expressions can be converted into a common data type, which will be the data type of the result.

The NULL values in the list will be ignored. The result is **NULL** only if the results of all expressions are **NULL**.

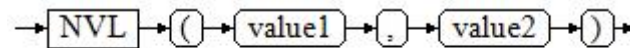
```
SELECT least(9000,2);
least
-----
2
(1 row)
```

[Conditional Expression Functions](#) describes the examples.

- NVL

[Figure 7-7](#) shows the syntax of an **NVL** expression.

**Figure 7-7** nvl::=



If the value of **value1** is **NULL**, **value2** is returned. Otherwise, **value1** is returned.

For example:

```
SELECT nvl(null,1);
NVL
-----
1
(1 row)
SELECT nvl ('Hello World' ,1);
nvl
-----
Hello World
(1 row)
```

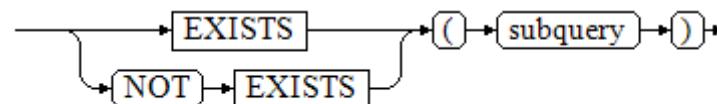
## 7.3 Subquery Expressions

Subquery expressions include the following types:

- EXISTS/NOT EXISTS

[Figure 7-8](#) shows the syntax of an **EXISTS/NOT EXISTS** expression.

**Figure 7-8** EXISTS/NOT EXISTS::=



The parameter of an **EXISTS** expression is an arbitrary **SELECT** statement, or subquery. The subquery is evaluated to determine whether it returns any rows. If it returns at least one row, the result of **EXISTS** is "true". If the subquery returns no rows, the result of **EXISTS** is "false".

The subquery will generally only be executed long enough to determine whether at least one row is returned, not all the way to completion.

For example:

```
SELECT sr_reason_sk,sr_customer_sk FROM tpcds.store_returns WHERE EXISTS (SELECT d_dom FROM
tpcds.date_dim WHERE d_dom = store_returns.sr_reason_sk and sr_customer_sk <10);
sr_reason_sk | sr_customer_sk
```

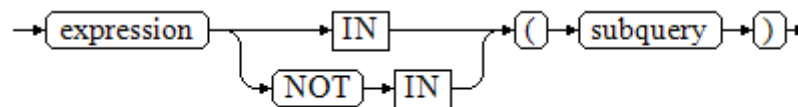
|    |   |
|----|---|
| 13 | 2 |
| 22 | 5 |
| 17 | 7 |
| 25 | 7 |
| 3  | 7 |
| 31 | 5 |
| 7  | 7 |
| 14 | 6 |
| 20 | 4 |
| 5  | 6 |
| 10 | 3 |
| 1  | 5 |
| 15 | 2 |
| 4  | 1 |
| 26 | 3 |

(15 rows)

- IN/NOT IN

Figure 7-9 shows the syntax of an **IN/NOT IN** expression.

Figure 7-9 IN/NOT IN::=



The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result. The result of **IN** is "true" if any equal subquery row is found. The result is "false" if no equal row is found (including the case where the subquery returns no rows).

This is in accordance with SQL's normal rules for Boolean combinations of null values. If the columns corresponding to two rows equal and are not empty, the two rows are equal to each other. If any columns corresponding to the two rows do not equal and are not empty, the two rows are not equal to each other. Otherwise, the result is **NULL**. If there are no equal right-hand values and at least one right-hand row yields null, the result of **IN** will be null, not false.

For example:

```
SELECT sr_reason_sk,sr_customer_sk FROM tpcds.store_returns WHERE sr_customer_sk IN (SELECT
d_dom FROM tpcds.date_dim WHERE d_dom < 10);
sr_reason_sk | sr_customer_sk
```

|    |   |
|----|---|
| 10 | 3 |
| 26 | 3 |
| 22 | 5 |
| 31 | 5 |
| 1  | 5 |
| 32 | 5 |
| 32 | 5 |
| 4  | 1 |

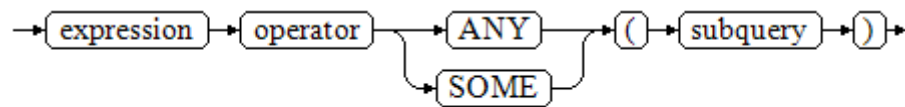
|    |  |   |
|----|--|---|
| 15 |  | 2 |
| 13 |  | 2 |
| 33 |  | 4 |
| 20 |  | 4 |
| 33 |  | 8 |
| 5  |  | 6 |
| 14 |  | 6 |
| 17 |  | 7 |
| 3  |  | 7 |
| 25 |  | 7 |
| 7  |  | 7 |

(19 rows)

- ANY/SOME

**Figure 7-10** shows the syntax of an **ANY/SOME** expression.

**Figure 7-10** any/some::=



The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result using the given operator, which must yield a Boolean result. The result of **ANY** is "true" if any true result is obtained. The result is "false" if no true result is found (including the case where the subquery returns no rows). **SOME** is a synonym of **ANY**. **IN** can be equivalently replaced with **ANY**.

For example:

```

SELECT sr_reason_sk,sr_customer_sk FROM tpcds.store_returns WHERE sr_customer_sk < ANY
(SELECT d_dom FROM tpcds.date_dim WHERE d_dom < 10);
sr_reason_sk | sr_customer_sk
-----+-----

```

|    |  |   |
|----|--|---|
| 26 |  | 3 |
| 17 |  | 7 |
| 32 |  | 5 |
| 32 |  | 5 |
| 13 |  | 2 |
| 31 |  | 5 |
| 25 |  | 7 |
| 5  |  | 6 |
| 7  |  | 7 |
| 10 |  | 3 |
| 1  |  | 5 |
| 14 |  | 6 |
| 4  |  | 1 |
| 3  |  | 7 |
| 22 |  | 5 |
| 33 |  | 4 |
| 20 |  | 4 |
| 33 |  | 8 |
| 15 |  | 2 |

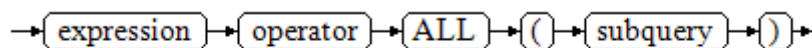
(19 rows)

- ALL

**Figure 7-11** shows the syntax of an **ALL** expression.



Figure 7-11 all::=



The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result using the given operator, which must yield a Boolean result. The result of **ALL** is "true" if all rows yield true (including the case where the subquery returns no rows). The result is "false" if any false result is found.

For example:

```

SELECT sr_reason_sk,sr_customer_sk FROM tpcds.store_returns WHERE sr_customer_sk < all(SELECT
d_dom FROM tpcds.date_dim WHERE d_dom < 10);
sr_reason_sk | sr_customer_sk
-----+-----
(0 rows)
  
```

## 7.4 Array Expressions

### IN

*expression* **IN** (*value* [, ...])

The parentheses on the right contain an expression list. The expression result on the left is compared with the content in the expression list. If the content in the list meets the expression result on the left, the result of **IN** is **true**. If no result meets the requirements, the result of **IN** is **false**.

Example:

```

SELECT 8000+500 IN (10000, 9000) AS RESULT;
result
-----
f
(1 row)
  
```

#### NOTE

If the expression result is null or the expression list does not meet the expression conditions and at least one empty value is returned for the expression list on the right, the result of **IN** is **null** rather than **false**. This method is consistent with the Boolean rules used when SQL statements return empty values.

### NOT IN

*expression* **NOT IN** (*value* [, ...])

The parentheses on the right contain an expression list. The expression result on the left is compared with the content in the expression list. If the content in the list does not meet the expression result on the left, the result of **NOT IN** is **true**. If any content meets the expression result, the result of **NOT IN** is **false**.

Example:

```

SELECT 8000+500 NOT IN (10000, 9000) AS RESULT;
result
  
```

```
-----
t
(1 row)
```

 **NOTE**

If the query statement result is null or the expression list does not meet the expression conditions and at least one empty value is returned for the expression list on the right, the result of **NOT IN** is **null** rather than **false**. This method is consistent with the Boolean rules used when SQL statements return empty values.

In all situations, **X NOT IN Y** equals to **NOT(X IN Y)**.

## ANY/SOME (array)

*expression operator ANY (array expression)*

*expression operator SOME (array expression)*

```
SELECT 8000+500 < SOME (array[10000,9000]) AS RESULT;
result
-----
t
(1 row)
SELECT 8000+500 < ANY (array[10000,9000]) AS RESULT;
result
-----
t
(1 row)
```

The parentheses on the right contain an array expression, which must generate an array value. The result of the expression on the left uses operators to compute and compare the results in each row of the array expression. The comparison result must be a Boolean value.

- If at least one comparison result is true, the result of **ANY** is **true**.
- If no comparison result is true, the result of **ANY** is false.

 **NOTE**

If no comparison result is true and the array expression generates at least one null value, the value of **ANY** is **NULL**, rather than false. This method is consistent with the Boolean rules used when SQL statements return empty values.

**SOME** is a synonym of **ANY**.

## ALL (array)

*expression operator ALL (array expression)*

The parentheses on the right contain an array expression, which must generate an array value. The result of the expression on the left uses operators to compute and compare the results in each row of the array expression. The comparison result must be a Boolean value.

- The result of **ALL** is "true" if all comparisons yield **true** (including the case where the array has zero elements).
- The result is **false** if any false result is found.

If the array expression yields a null array, the result of **ALL** will be null. If the left-hand expression yields null, the result of **ALL** is ordinarily null (though a non-strict

comparison operator could possibly yield a different result). Also, if the right-hand array contains any null elements and no false comparison result is obtained, the result of **ALL** will be null, not true (again, assuming a strict comparison operator). This method is consistent with the Boolean rules used when SQL statements return empty values.

```
SELECT 8000+500 < ALL (array[10000,9000]) AS RESULT;  
result  
-----  
t  
(1 row)
```

## 7.5 Row Expressions

Syntax:

*row\_constructor operator row\_constructor*

Both sides of the row expression are row constructors. The values of both rows must have the same number of fields and they are compared with each other. The row comparison allows operators including =, <>, <, <=, and >= or a similar operator.

The use of operators =<> is slightly different from other operators. If all fields of two rows are not empty and equal, the two rows are equal. If any field in two rows is not empty and not equal, the two rows are not equal. Otherwise, the comparison result is null.

For operators <, <=, >, and >=, the fields in rows are compared from left to right until a pair of fields that are not equal or are empty are detected. If the pair of fields contains at least one null value, the comparison result is null. Otherwise, the comparison result of this pair of fields is the final result.

For example:

```
SELECT ROW(1,2,NULL) < ROW(1,3,0) AS RESULT;  
result  
-----  
t  
(1 row)
```

# 8 Type Conversion

## 8.1 Overview

### Context

SQL is a typed language. That is, every data item has an associated data type which determines its behavior and allowed usage. GaussDB(DWS) has an extensible type system that is more general and flexible than other SQL implementations. Hence, most type conversion behavior in GaussDB(DWS) is governed by general rules. This allows the use of mixed-type expressions.

The GaussDB(DWS) scanner/parser divides lexical elements into five fundamental categories: integers, floating-point numbers, strings, identifiers, and keywords. Constants of most non-numeric types are first classified as strings. The SQL language definition allows specifying type names with constant strings. For example, the query:

```
SELECT text 'Origin' AS "label", point '(0,0)' AS "value";
label | value
-----+-----
Origin | (0,0)
(1 row)
```

has two literal constants, of type **text** and **point**. If a type is not specified for a string literal, then the placeholder type **unknown** is assigned initially.

There are four fundamental SQL constructs requiring distinct type conversion rules in the GaussDB(DWS) parser:

- **Function calls**  
Much of the SQL type system is built around a rich set of functions. Functions can have one or more arguments. Since SQL permits function overloading, the function name alone does not uniquely identify the function to be called. The parser must select the right function based on the data types of the supplied arguments.
- **Operators**  
SQL allows expressions with prefix and postfix unary (one-argument) operators, as well as binary (two-argument) operators. Like functions,

operators can be overloaded, so the same problem of selecting the right operator exists.

- Value Storage

SQL **INSERT** and **UPDATE** statements place the results of expressions into a table. The expressions in the statement must be matched up with, and perhaps converted to, the types of the target columns.

- **UNION, CASE,** and related constructs

Since all query results from a unionized **SELECT** statement must appear in a single set of columns, the types of the results of each **SELECT** clause must be matched up and converted to a uniform set. Similarly, the result expressions of a **CASE** construct must be converted to a common type so that the **CASE** expression as a whole has a known output type. The same holds for **ARRAY** constructs, and for the **GREATEST** and **LEAST** functions.

The system catalog `pg_cast` stores information about which conversions, or casts, exist between which data types, and how to perform those conversions. For details, see `PG_CAST`.

The return type and conversion behavior of an expression are determined during semantic analysis. Data types are divided into several basic type categories, including **Boolean, numeric, string, bitstring, datetime, timespan, geometric,** and **network**. Within each category there can be one or more preferred types, which are preferred when there is a choice of possible types. With careful selection of preferred types and available implicit casts, it is possible to ensure that ambiguous expressions (those with multiple candidate parsing solutions) can be resolved in a useful way.

All type conversion rules are designed based on the following principles:

- Implicit conversions should never have surprising or unpredictable outcomes.
- There should be no extra overhead in the parser or executor if a query does not need implicit type conversion. That is, if a query is well-formed and the types already match, then the query should execute without spending extra time in the parser and without introducing unnecessary implicit conversion calls in the query.
- Additionally, if a query usually requires an implicit conversion for a function, and if then the user defines a new function with the correct argument types, the parser should use this new function.

## Converting Empty Strings to Numeric Values in TD-Compatible Mode

- Different from the Oracle database, which processes an empty string as `NULL`, Teradata database converts an empty string to `0` by default. Therefore, when an empty string is queried, value `0` is found. Similarly, in TD-compatible mode, the empty string is converted to `0` of the corresponding numeric type by default. In addition, `'-'`, `'+'`, and `' '` are converted to `0` by default in TD-compatible mode, but an error is reported for a decimal point string. For example:

```
SELECT * FROM t1 WHERE a = '';
a
---
0
(1 row)
```

## 8.2 Operators

### Operator Type Resolution

1. Select the operators to be considered from the **pg\_operator** system catalog. Considered operators are those with the matching name and argument count. If the search path finds multiple available operators, only the most suitable one is considered.
2. Look for the best match.
  - a. Discard candidate operators for which the input types do not match and cannot be converted (using an implicit conversion) to match. **unknown** literals are assumed to be convertible to anything for this purpose. If only one candidate remains, use it; else continue to the next step.
  - b. Run through all candidates and keep those with the most exact matches on input types. Domains are considered the same as their base type for this purpose. Keep all candidates if there are no exact matches. If only one candidate remains, use it; else continue to the next step.
  - c. Run through all candidates and keep those that accept preferred types (of the input data type's type category) at the most positions where type conversion will be required. Keep all candidates if none accepts preferred types. If only one candidate remains, use it; else continue to the next step.
  - d. If any input arguments are of **unknown** types, check the type categories accepted at those argument positions by the remaining candidates. At each position, select the string category if any candidate accepts that category. (This bias towards string is appropriate since an unknown-type literal looks like a string.) Otherwise, if all the remaining candidates accept the same type category, select that category; otherwise fail because the correct choice cannot be deduced without more clues. Now discard candidates that do not accept the selected type category. Furthermore, if any candidate accepts a preferred type in that category, discard candidates that accept non-preferred types for that argument. Keep all candidates if none survive these tests. If only one candidate remains, use it; else continue to the next step.
  - e. If there are both **unknown** and known-type arguments, and all the known-type arguments have the same type, assume that the **unknown** arguments are also of that type, and check which candidates can accept that type at the unknown-argument positions. If exactly one candidate passes this test, use it. Otherwise, an error is reported.

### Examples

Example 1: Use factorial operator type resolution. There is only one factorial operator (postfix !) defined in the system catalog, and it takes an argument of type **bigint**. The scanner assigns an initial type of **bigint** to the argument in this query expression:

```
SELECT 40 ! AS "40 factorial";  
      40 factorial
```

```
-----  
815915283247897734345611269596115894272000000000  
(1 row)
```

So the parser does a type conversion on the operand and the query is equivalent to:

```
SELECT CAST(40 AS bigint) ! AS "40 factorial";
```

Example 2: String concatenation operator type resolution. A string-like syntax is used for working with string types and for working with complex extension types. Strings with unspecified type are matched with likely operator candidates. An example with one unspecified argument:

```
SELECT text 'abc' || 'def' AS "text and unknown";  
text and unknown  
-----  
abcdef  
(1 row)
```

In this example, the parser looks for an operator whose parameters are of the text type. Such an operator is found.

Here is a concatenation of two values of unspecified types:

```
SELECT 'abc' || 'def' AS "unspecified";  
unspecified  
-----  
abcdef  
(1 row)
```

#### NOTE

In this case there is no initial hint for which type to use, since no types are specified in the query. So, the parser looks for all candidate operators and finds that there are candidates accepting both string-category and bit-string-category inputs. Since string category is preferred when available, that category is selected, and then the preferred type for strings, **text**, is used as the specific type to resolve the unknown-type literals.

Example 3: Absolute-value and negation operator type resolution. The GaussDB(DWS) operator catalog has several entries for the prefix operator @. All the entries implement absolute-value operations for various numeric data types. One of these entries is for type **float8**, which is the preferred type in the numeric category. Therefore, GaussDB(DWS) will use that entry when faced with an **unknown** input:

```
SELECT @ '-4.5' AS "abs";  
abs  
-----  
4.5  
(1 row)
```

Here the system has implicitly resolved the unknown-type literal as type **float8** before applying the chosen operator.

Example 4: Use the array inclusion operator type resolution as an example. Here is another example of resolving an operator with one known and one unknown input:

```
SELECT array[1,2] <@ '{1,2,3}' as "is subset";  
is subset  
-----  
t  
(1 row)
```

 NOTE

The GaussDB(DWS) operator catalog has several entries for the infix operator <@, but the only two that could possibly accept an integer array on the left side are array inclusion (anyarray <@ anyarray) and range inclusion (anyelement <@ anyrange). Since none of these polymorphic pseudo-types (see [Pseudo-Types](#)) is considered preferred, the parser cannot resolve the ambiguity on that basis. However, the last resolution rule tells it to assume that the unknown-type literal is of the same type as the other input, that is, integer array. Now only one of the two operators can match, so array inclusion is selected. (Had range inclusion been selected, we would have gotten an error, because the string does not have the right format to be a range literal.)

## 8.3 Functions

### Function Type Resolution

1. Select the functions to be considered from the **pg\_proc** system catalog. If a non-schema-qualified function name was used, the functions in the current search path are considered. If a qualified function name was given, only functions in the specified schema are considered.  
If the search path finds multiple functions of different argument types, a proper function in the path is considered.
2. Check for a function accepting exactly the input argument types. If the function exists, use it. Cases involving **unknown** will never find a match at this step.
3. If no exact match is found, see if the function call appears to be a special type conversion request.
4. Look for the best match.
  - a. Discard candidate functions for which the input types do not match and cannot be converted (using an implicit conversion) to match. **unknown** literals are assumed to be convertible to anything for this purpose. If only one candidate remains, use it; else continue to the next step.
  - b. Run through all candidates and keep those with the most exact matches on input types. Domains are considered the same as their base type for this purpose. Keep all candidates if none has exact matches. If only one candidate remains, use it; else continue to the next step.
  - c. Run through all candidates and keep those that accept preferred types at the most positions where type conversion will be required. Keep all candidates if none accepts preferred types. If only one candidate remains, use it; else continue to the next step.
  - d. If any input arguments are of **unknown** types, check the type categories accepted at those argument positions by the remaining candidates. At each position, select the string category if any candidate accepts that category. (This bias towards string is appropriate since an unknown-type literal looks like a string.) Otherwise, if all the remaining candidates accept the same type category, select that category; otherwise fail because the correct choice cannot be deduced without more clues. Now discard candidates that do not accept the selected type category. Furthermore, if any candidate accepts a preferred type in that category, discard candidates that accept non-preferred types for that argument.



Keep all candidates if none survives these tests. If only one candidate remains, use it; else continue to the next step.

- e. If there are both **unknown** and known-type arguments, and all the known-type arguments have the same type, assume that the **unknown** arguments are also of that type, and check which candidates can accept that type at the **unknown**-argument positions. If exactly one candidate passes this test, use it. Otherwise, fail.

## Examples

Example 1: Use the rounding function argument type resolution as the first example. There is only one **round** function that takes two arguments; it takes a first argument of type **numeric** and a second argument of type **integer**. So the following query automatically converts the first argument of type **integer** to **numeric**:

```
SELECT round(4, 4);
round
-----
4.0000
(1 row)
```

That query is converted by the parser to:

```
SELECT round(CAST (4 AS numeric), 4);
```

Since numeric constants with decimal points are initially assigned the type **numeric**, the following query will require no type conversion and therefore might be slightly more efficient:

```
SELECT round(4.0, 4);
```

Example 2: Use the substring function type resolution as the second example. There are several **substr** functions, one of which takes types **text** and **integer**. If called with a string constant of unspecified type, the system chooses the candidate function that accepts an argument of the preferred category **string** (namely of type **text**).

```
SELECT substr('1234', 3);
substr
-----
34
(1 row)
```

If the string is declared to be of type **varchar**, as might be the case if it comes from a table, then the parser will try to convert it to become **text**:

```
SELECT substr(varchar '1234', 3);
substr
-----
34
(1 row)
```

This is transformed by the parser to effectively become:

```
SELECT substr(CAST (varchar '1234' AS text), 3);
```

### NOTE

The parser learns from the **pg\_cast** catalog that **text** and **varchar** are binary-compatible, meaning that one can be passed to a function that accepts the other without doing any physical conversion. Therefore, no type conversion is inserted in this case.

And, if the function is called with an argument of type **integer**, the parser will try to convert that to **text**:

```
SELECT substr(1234, 3);
substr
-----
34
(1 row)
```

This is transformed by the parser to effectively become:

```
SELECT substr(CAST (1234 AS text), 3);
substr
-----
34
(1 row)
```

## 8.4 Value Storage

### Value Storage Type Resolution

1. Search for an exact match with the target column.
2. Try to convert the expression to the target type. This will succeed if there is a registered cast between the two types. If the expression is an unknown-type literal, the content of the literal string will be fed to the input conversion routine for the target type.
3. Check to see if there is a sizing cast for the target type. A sizing cast is a cast from that type to itself. If one is found in the **pg\_cast** catalog, apply it to the expression before storing into the destination column. The implementation function for such a cast always takes an extra parameter of type **integer**. The parameter receives the destination column's **atttypmod** value (typically its declared length, although the interpretation of **atttypmod** varies for different data types), and may take a third Boolean parameter that says whether the cast is explicit or implicit. The cast function is responsible for applying any length-dependent semantics such as size checking or truncation.

### Examples

Use the **character** storage type conversion as an example. For a target column declared as **character(20)** the following statement shows that the stored value is sized correctly:

```
CREATE TABLE tpods.value_storage_t1 (
  VS_COL1 CHARACTER(20)
) DISTRIBUTE BY HASH (VS_COL1);

SELECT VS_COL1, octet_length(VS_COL1) FROM tpods.value_storage_t1;
  vs_col1      | octet_length
-----+-----
abcdef        |          20
(1 row)
)

DROP TABLE tpods.value_storage_t1;
```

 NOTE

What has really happened here is that the two unknown literals are resolved to **text** by default, allowing the || operator to be resolved as **text** concatenation. Then the **text** result of the operator is converted to **bpchar** ("blank-padded char", the internal name of the **character** data type) to match the target column type. Since the conversion from **text** to **bpchar** is binary-coercible, this conversion does not insert any real function call. Finally, the sizing function **bpchar(bpchar, integer, Boolean)** is found in the system catalog and used for the operator's result and the stored column length. This type-specific function performs the required length check and addition of padding spaces.

## 8.5 UNION, CASE, and Related Constructs

SQL **UNION** constructs must match up possibly dissimilar types to become a single result set. The resolution algorithm is applied separately to each output column of a union query. The **INTERSECT** and **EXCEPT** constructs resolve dissimilar types in the same way as **UNION**. The **CASE, ARRAY, VALUES, GREATEST** and **LEAST** constructs use the identical algorithm to match up their component expressions and select a result data type.

### Type Resolution for UNION, CASE, and Related Constructs

- If all inputs are of the same type, and it is not **unknown**, resolve as that type.
- If all inputs are of type **unknown**, resolve as type **text** (the preferred type of the string category). Otherwise, **unknown** inputs are ignored.
- If the non-unknown inputs are not all of the same type category, fail. (Type **unknown** is not included.)
- If the non-unknown inputs are all of the same type category, choose the first non-unknown input type which is a preferred type in that category, if there is one. (Exception: The **UNION** operation regards the type of the first branch as the selected type.)

 NOTE

**typcategory** in the **pg\_type** system catalog indicates the data type category.  
**typispreferred** indicates whether a type is preferred in **typcategory**.

- All the input is converted to the selected type. (The original length of a string is retained). Fail if there is not an implicit conversion from a given input to the selected type.
- If the input contains the **json, txid\_snapshot, sys\_refcursor, or geometry** type, **UNION** cannot be performed.

### Type Resolution for CASE and COALESCE in TD Compatibility Type

- If all inputs are of the same type, and it is not **unknown**, resolve as that type.
- If all inputs are of type **unknown**, resolve as type **text**.
- If inputs are of string type (including **unknown** which is resolved as type **text**) and digit type, resolve as the string type. If the inputs are not of the two types, fail.
- If the non-unknown inputs are all of the same type category, choose the input type which is a preferred type in that category, if there is one.
- Convert all inputs to the selected type. Fail if there is not an implicit conversion from a given input to the selected type.

## Examples

Example 1: Use type resolution with underspecified types in a union as the first example. Here, the unknown-type literal **'b'** will be resolved to type **text**.

```
SELECT text 'a' AS "text" UNION SELECT 'b';
text
-----
a
b
(2 rows)
```

Example 2: Use type resolution in a simple union as the second example. The literal **1.2** is of type **numeric**, and the **integer** value **1** can be cast implicitly to **numeric**, so that type is used.

```
SELECT 1.2 AS "numeric" UNION SELECT 1;
numeric
-----
1
1.2
(2 rows)
```

Example 3: Use type resolution in a transposed union as the third example. Here, since type **real** cannot be implicitly cast to **integer**, but **integer** can be implicitly cast to **real**, the union result type is resolved as **real**.

```
SELECT 1 AS "real" UNION SELECT CAST('2.2' AS REAL);
real
-----
1
2.2
(2 rows)
```

Example 4: In the **TD** type, if input parameters for **COALESCE** are of **int** and **varchar** types, resolve as type **varchar**. Fail in the **ORA** type.

```
-- In Oracle mode, create the oracle_1 database compatible with Oracle:
CREATE DATABASE oracle_1 dbcompatibility = 'ORA';

-- Switch to the oracle_1 database:
\c oracle_1

-- Create the t1 table:
oracle_1=# CREATE TABLE t1(a int, b varchar(10));

-- Show the execution plan of a statement for querying the types int and varchar of input parameters for COALESCE:
oracle_1=# EXPLAIN SELECT coalesce(a, b) FROM t1;
ERROR: COALESCE types integer and character varying cannot be matched
CONTEXT: referenced column: coalesce

-- Delete the tables:
oracle_1=# DROP TABLE t1;

-- Switch to the postgres database:
oracle_1=# \c postgres

-- In TD mode, create the td_1 database compatible with Teradata:
CREATE DATABASE td_1 dbcompatibility = 'TD';

-- Switch to the td_1 database:
\c td_1

-- Create the t2 table:
td_1=# CREATE TABLE t2(a int, b varchar(10));

-- Show the execution plan of a statement for querying the types int and varchar of input parameters for
```

**COALESCE:**

```
td_1=# EXPLAIN VERBOSE select coalesce(a, b) from t2;  
QUERY PLAN
```

```
-----  
Data Node Scan (cost=0.00..0.00 rows=0 width=0)  
Output: (COALESCE((t1.a)::character varying, t1.b))  
Node/s: All datanodes  
Remote query: SELECT COALESCE(a::character varying, b) AS "coalesce" FROM public.t1  
(4 rows)
```

-- Delete the tables:

```
td_1=# DROP TABLE t2;
```

-- Switch to the **postgres** database:

```
td_1=# \c postgres
```

-- Delete databases in **Oracle** and **TD** mode:

```
DROP DATABASE oracle_1;
```

```
DROP DATABASE td_1;
```

# 9 Full Text Search

---

## 9.1 Introduction

### 9.1.1 Full-Text Retrieval

Textual search operators have been used in databases for years. GaussDB(DWS) has `~`, `~*`, `LIKE`, and `ILIKE` operators for textual data types, but they lack many essential properties required by modern information systems. They can be supplemented by indexes and dictionaries.

Text search lacks the following essential properties required by information systems:

- There is no linguistic support, even for English.  
Regular expressions are not sufficient because they cannot easily handle derived words. For example, you might miss documents that contain **satisfies**, although you probably would like to find them when searching for **satisfy**. It is possible to use **OR** to search for multiple derived forms, but this is tedious and error-prone, because some words can have several thousand derivatives.
- They provide no ordering (ranking) of search results, which makes them ineffective when thousands of matching documents are found.
- They tend to be slow because there is no index support, so they must process all documents for every search.

Full text indexing allows documents to be preprocessed and an index is saved for later rapid searching. Preprocessing includes:

- Parsing documents into tokens  
It is useful to identify various classes of tokens, for example, numbers, words, complex words, and email addresses, so that they can be processed differently. In principle, token classes depend on the specific application, but for most purposes it is adequate to use a predefined set of classes.
- Converting tokens into lexemes  
A lexeme is a string, just like a token, but it has been normalized so that different forms of the same word are made alike. For example, normalization almost always includes folding upper-case letters to lower-case, and often

involves removal of suffixes (such as **s** or **es** in English) This allows searches to find variant forms of the same word, without tediously entering all the possible variants. Also, this step typically eliminates stop words, which are words that are so common that they are useless for searching. (In short, tokens are raw fragments of the document text, while lexemes are words that are believed useful for indexing and searching.) GaussDB(DWS) uses dictionaries to perform this step and provides various standard dictionaries.

- Storing preprocessed documents optimized for searching

For example, each document can be represented as a sorted array of normalized lexemes. Along with the lexemes, it is often desirable to store positional information for proximity ranking. Therefore, a document that contains a more "dense" region of query words is assigned with a higher rank than the one with scattered query words.

Dictionaries allow fine-grained control over how tokens are normalized. With appropriate dictionaries, you can define stop words that should not be indexed.

A data type **tsvector** is provided for storing preprocessed documents, along with a type **tsquery** for storing query conditions. For details, see [Text Search Types](#). For details about the functions and operators available for these data types, see [Text Search Functions and Operators](#). The match operator **@@**, which is the most important among those functions and operators, is introduced in [Basic Text Matching](#).

## 9.1.2 What Is a Document?

A document is the unit of searching in a full text search system; for example, a magazine article or email message. The text search engine must be able to parse documents and store associations of lexemes (keywords) with their parent document. Later, these associations are used to search for documents that contain query words.

For searches within GaussDB(DWS), a document is normally a textual column within a row of a database table, or possibly a combination (concatenation) of such columns, perhaps stored in several tables or obtained dynamically. In other words, a document can be constructed from different parts for indexing and it might not be stored anywhere as a whole. For example:

```
SELECT d_dow || '-' || d_dom || '-' || d_fy_week_seq AS identify_serials FROM tpods.date_dim WHERE  
d_fy_week_seq = 1;  
identify_serials
```

```
-----  
5-6-1  
0-8-1  
2-3-1  
3-4-1  
4-5-1  
1-2-1  
6-7-1  
(7 rows)
```

### NOTICE

Actually, in these example queries, **coalesce** should be used to prevent a single **NULL** attribute from causing a **NULL** result for the whole document.

Another possibility is to store the documents as simple text files in the file system. In this case, the database can be used to store the full text index and to execute searches, and some unique identifier can be used to retrieve the document from the file system. However, retrieving files from outside the database requires system administrator permissions or special function support, so this is less convenient than keeping all the data inside the database. Also, keeping everything inside the database allows easy access to document metadata to assist in indexing and display.

For text search purposes, each document must be reduced to the preprocessed **tsvector** format. Searching and relevance-based ranking are performed entirely on the **tsvector** representation of a document. The original text is retrieved only when the document has been selected for display to a user. We therefore often speak of the **tsvector** as being the document, but it is only a compact representation of the full document.

### 9.1.3 Basic Text Matching

Full text search in GaussDB(DWS) is based on the match operator **@@**, which returns **true** if a **tsvector** (document) matches a **tsquery** (query). It does not matter which data type is written first:

```
SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector @@ 'cat & rat'::tsquery AS RESULT;
result
-----
t
(1 row)
SELECT 'fat & cow'::tsquery @@ 'a fat cat sat on a mat and ate a fat rat'::tsvector AS RESULT;
result
-----
f
(1 row)
```

As the above example suggests, a **tsquery** is not raw text, any more than a **tsvector** is. A **tsquery** contains search terms, which must be already-normalized lexemes, and may combine multiple terms using **AND**, **OR**, and **NOT** operators. For details, see [Text Search Types](#). There are functions **to\_tsquery** and **plainto\_tsquery** that are helpful in converting user-written text into a proper **tsquery**, for example by normalizing words appearing in the text. Similarly, **to\_tsvector** is used to parse and normalize a document string. So in practice a text search match would look more like this:

```
SELECT to_tsvector('fat cats ate fat rats') @@ to_tsquery('fat & rat') AS RESULT;
result
-----
t
(1 row)
```

Observe that this match would not succeed if written as follows:

```
SELECT 'fat cats ate fat rats'::tsvector @@ to_tsquery('fat & rat') AS RESULT;
result
-----
f
(1 row)
```

In the preceding match, no normalization of the word **rats** will occur. Therefore, **rats** does not match **rat**.

The **@@** operator also supports text input, allowing explicit conversion of a text string to **tsvector** or **tsquery** to be skipped in simple cases. The variants available are:



```
tsvector @@ tsquery  
tsquery @@ tsvector  
text @@ tsquery  
text @@ text
```

We already saw the first two of these. The form **text @@ tsquery** is equivalent to **to\_tsvector(text) @@ tsquery**. The form **text @@ text** is equivalent to **to\_tsvector(text) @@ plainto\_tsquery(text)**.

## 9.1.4 Configurations

Full text search functionality includes the ability to do many more things: skip indexing certain words (stop words), process synonyms, and use sophisticated parsing, for example, parse based on more than just white space. This functionality is controlled by text search configurations. GaussDB(DWS) comes with predefined configurations for many languages, and you can easily create your own configurations. (The `\df` command of `gsql` shows all available configurations.)

During installation an appropriate configuration is selected and **default\_text\_search\_config** is set accordingly in **postgresql.conf**. If you are using the same text search configuration for the entire cluster you can use the value in **postgresql.conf**. To use different configurations throughout the cluster but the same configuration within any one database, use `ALTER DATABASE ... SET`. Otherwise, you can set **default\_text\_search\_config** in each session.

Each text search function that depends on a configuration has an optional argument, so that the configuration to use can be specified explicitly. **default\_text\_search\_config** is used only when this argument is omitted.

To make it easier to build custom text search configurations, a configuration is built up from simpler database objects. GaussDB(DWS)'s text search facility provides the following types of configuration-related database objects:

- Text search parsers break documents into tokens and classify each token (for example, as words or numbers).
- Text search dictionaries convert tokens to normalized form and reject stop words.
- Text search templates provide the functions underlying dictionaries. (A dictionary simply specifies a template and a set of parameters for the template.)
- Text search configurations select a parser and a set of dictionaries to use to normalize the tokens produced by the parser.

## 9.2 Table and index

### 9.2.1 Searching a Table

It is possible to do a full text search without an index.

- A simple query to print each row that contains the word **science** in its **body** column is as follows:

```
DROP SCHEMA IF EXISTS tsearch CASCADE;
```

```
CREATE SCHEMA tsearch;

CREATE TABLE tsearch.pgweb(id int, body text, title text, last_mod_date date);

INSERT INTO tsearch.pgweb VALUES(1, 'Philology is the study of words, especially the history and
development of the words in a particular language or group of languages.', 'Philology', '2010-1-1');

INSERT INTO tsearch.pgweb VALUES(2, 'Mathematics is the science that deals with the logic of shape,
quantity and arrangement.', 'Mathematics', '2010-1-1');

INSERT INTO tsearch.pgweb VALUES(3, 'Computer science is the study of processes that interact with
data and that can be represented as data in the form of programs.', 'Computer science', '2010-1-1');

INSERT INTO tsearch.pgweb VALUES(4, 'Chemistry is the scientific discipline involved with elements
and compounds composed of atoms, molecules and ions.', 'Chemistry', '2010-1-1');

INSERT INTO tsearch.pgweb VALUES(5, 'Geography is a field of science devoted to the study of the
lands, features, inhabitants, and phenomena of the Earth and planets.', 'Geography', '2010-1-1');

INSERT INTO tsearch.pgweb VALUES(6, 'History is a subject studied in schools, colleges, and
universities that deals with events that have happened in the past.', 'History', '2010-1-1');

INSERT INTO tsearch.pgweb VALUES(7, 'Medical science is the science of dealing with the
maintenance of health and the prevention and treatment of disease.', 'Medical science', '2010-1-1');

INSERT INTO tsearch.pgweb VALUES(8, 'Physics is one of the most fundamental scientific disciplines,
and its main goal is to understand how the universe behaves.', 'Physics', '2010-1-1');

SELECT id, body, title FROM tsearch.pgweb WHERE to_tsvector('english', body) @@
to_tsquery('english', 'science');
id | body | title
----+-----+-----
2 | Mathematics is the science that deals with the logic of shape, quantity and
arrangement. | Mathematics
3 | Computer science is the study of processes that interact with data and that can be represented as
data in the form of programs. | Computer science
5 | Geography is a field of science devoted to the study of the lands, features, inhabitants, and
phenomena of the Earth and planets. | Geography
7 | Medical science is the science of dealing with the maintenance of health and the prevention and
treatment of disease. | Medical science
(4 rows)
```

This will also find related words, such as **science**, since all these are reduced to the same normalized lexeme.

The query above specifies that the **english** configuration is to be used to parse and normalize the strings. Alternatively we could omit the configuration parameters, and use the configuration set by **default\_text\_search\_config**.

```
SHOW default_text_search_config;
default_text_search_config
-----
pg_catalog.english
(1 row)

SELECT id, body, title FROM tsearch.pgweb WHERE to_tsvector(body) @@ to_tsquery('science');
id | body | title
----+-----+-----
2 | Mathematics is the science that deals with the logic of shape, quantity and
arrangement. | Mathematics
3 | Computer science is the study of processes that interact with data and that can be represented as
data in the form of programs. | Computer science
5 | Geography is a field of science devoted to the study of the lands, features, inhabitants, and
```

```
phenomena of the Earth and planets. | Geography
7 | Medical science is the science of dealing with the maintenance of health and the prevention and
treatment of disease. | Medical science
```

(4 rows)

- A more complex example to select the ten most recent documents that contain **treatment** and **science** in the **title** or **body** column is as follows:  
SELECT title FROM tsearch.pgweb WHERE to\_tsvector(title || ' ' || body) @@ to\_tsquery('treatment & science') ORDER BY last\_mod\_date DESC LIMIT 10;

```
title
-----
```

```
Medical science
```

(1 rows)

For clarity we omitted the **coalesce** function calls which would be needed to find rows that contain **NULL** in one of the two columns.

The preceding examples show queries without using indexes. Most applications will find this approach too slow. Therefore, practical use of text searching usually requires creating an index, except perhaps for occasional ad-hoc searches.

## 9.2.2 Creating an Index

You can create a **GIN** index to speed up text searches:

```
CREATE INDEX pgweb_idx_1 ON tsearch.pgweb USING gin(to_tsvector('english', body));
```

The **to\_tsvector** function comes in two versions: the 1-argument version and the 2-argument version. When the 1-argument version is used, the system uses the configuration specified by **default\_text\_search\_config** by default.

Notice that the 2-argument version of **to\_tsvector** is used for index creation. Only text search functions that specify a configuration name can be used in expression indexes. This is because the index contents must be unaffected by **default\_text\_search\_config**, whose value can be changed at any time. If they were affected, the index contents might be inconsistent, because different entries could contain **tsvectors** that were created with different text search configurations, and there would be no way to guess which was which. It would be impossible to dump and restore such an index correctly.

Because the two-argument version of **to\_tsvector** was used in the index above, only a query reference that uses the 2-argument version of **to\_tsvector** with the same configuration name will use that index. That is, **WHERE to\_tsvector('english', body) @@ 'a & b'** can use the index, but **WHERE to\_tsvector(body) @@ 'a & b'** cannot. This ensures that an index will be used only with the same configuration used to create the index entries.

It is possible to set up more complex expression indexes wherein the configuration name is specified by another column. For example:

```
CREATE INDEX pgweb_idx_2 ON tsearch.pgweb USING gin(to_tsvector('zhparser', body));
```

where **body** is a column in the **pgweb** table. This allows mixed configurations in the same index while recording which configuration was used for each index entry. This would be useful, for example, if the document collection contained documents in different languages. Again, queries that are meant to use the index must be phrased to match, for example, **WHERE to\_tsvector(config\_name, body) @@ 'a & b'** must match **to\_tsvector** in the index.

Indexes can even concatenate columns:

```
CREATE INDEX pgweb_idx_3 ON tsearch.pgweb USING gin(to_tsvector('english', title || ' ' || body));
```

Another approach is to create a separate **tsvector** column to hold the output of **to\_tsvector**. This example is a concatenation of **title** and **body**, using **coalesce** to ensure that one column will still be indexed when the other is **NULL**:

```
ALTER TABLE tsearch.pgweb ADD COLUMN textsearchable_index_col tsvector;  
UPDATE tsearch.pgweb SET textsearchable_index_col = to_tsvector('english', coalesce(title,'') || ' ' ||  
coalesce(body,''));
```

Then, create a GIN index to speed up the search:

```
CREATE INDEX textsearch_idx_4 ON tsearch.pgweb USING gin(textsearchable_index_col);
```

Now you are ready to perform a fast full text search:

```
SELECT title  
FROM tsearch.pgweb  
WHERE textsearchable_index_col @@ to_tsquery('science & Computer')  
ORDER BY last_mod_date DESC  
LIMIT 10;
```

```
title  
-----  
Computer science  
  
(1 rows)
```

One advantage of the separate-column approach over an expression index is that it is unnecessary to explicitly specify the text search configuration in queries in order to use the index. As shown in the preceding example, the query can depend on **default\_text\_search\_config**. Another advantage is that searches will be faster, since it will not be necessary to redo the **to\_tsvector** calls to verify index matches. The expression-index approach is simpler to set up, however, and it requires less disk space since the **tsvector** representation is not stored explicitly.

## 9.2.3 Constraints on Index Use

The following is an example of index use:

```
create table table1 (c_int int,c_bigint bigint,c_varchar varchar,c_text text) with(orientation=row);  
  
create text search configuration ts_conf_1(parser=POUND);  
create text search configuration ts_conf_2(parser=POUND) with(split_flag='%');  
  
set default_text_search_config='ts_conf_1';  
create index idx1 on table1 using gin(to_tsvector(c_text));  
  
set default_text_search_config='ts_conf_2';  
create index idx2 on tscp_u_m_005_tbl using gin(to_tsvector(c_text));  
  
select c_varchar,to_tsvector(c_varchar) from table1 where to_tsvector(c_text) @@  
plainto_tsquery('%#@...&***') and to_tsvector(c_text) @@ plainto_tsquery('Company') and c_varchar is  
not null order by 1 desc limit 3;
```

In this example, **table1** has two GIN indexes created on the same column **c\_text**, **idx1** and **idx2**, but these two indexes are created under different settings of **default\_text\_search\_config**. Differences between this example and the scenario where one table has common indexes created on the same column are as follows:

- GIN indexes use different parsers (that is, different delimiters). In this case, the index data of **idx1** is different from that of **idx2**.

- In the specified scenario, the index data of multiple common indexes created on the same column is the same.

As a result, using **idx1** and **idx2** for the same query returns different results.

## Constraints

Still use the above example. When:

- Multiple GIN indexes are created on the same column of the same table.
- The GIN indexes use different parsers (that is, different delimiters).
- The column is used in a query, and an index scan is used in the execution plan.

To avoid different query results caused by different GIN indexes, ensure that only one GIN index is available on a column of the physical table.

## 9.3 Controlling Text Search

### 9.3.1 Parsing Documents

GaussDB(DWS) provides function **to\_tsvector** for converting a document to the **tsvector** data type.

```
to_tsvector([ config regconfig, ] document text) returns tsvector
```

**to\_tsvector** parses a textual document into tokens, reduces the tokens to lexemes, and returns a **tsvector**, which lists the lexemes together with their positions in the document. The document is processed according to the specified or default text search configuration. Here is a simple example:

```
SELECT to_tsvector('english', 'a fat cat sat on a mat - it ate a fat rats');
       to_tsvector
-----
'ate':9 'cat':3 'fat':2,11 'mat':7 'rat':12 'sat':4
```

In the preceding example we see that the resulting **tsvector** does not contain the words **a**, **on**, or **it**, the word **rats** became **rat**, and the punctuation sign (-) was ignored.

The **to\_tsvector** function internally calls a parser which breaks the document text into tokens and assigns a type to each token. For each token, a list of dictionaries is consulted. where the list can vary depending on the token type. The first dictionary that recognizes the token emits one or more normalized lexemes to represent the token. For example:

- **rats** became **rat** because one of the dictionaries recognized that the word **rats** is a plural form of **rat**.
- Some words are recognized as stop words (see [Stop Words](#)), which causes them to be ignored since they occur too frequently to be useful in searching. In our example these are **a**, **on**, and **it**.
- If no dictionary in the list recognizes the token then it is also ignored. In this example that happened to the punctuation sign (-) because there are in fact no dictionaries assigned for its token type (**Space symbols**), meaning space tokens will never be indexed.

The choices of parser, dictionaries and which types of tokens to index are determined by the selected text search configuration. It is possible to have many different configurations in the same database, and predefined configurations are available for various languages. In our example we used the default configuration **english** for the English language.

The function **setweight** can be used to label the entries of a **tsvector** with a given weight, where a weight is one of the letters **A**, **B**, **C**, or **D**. This is typically used to mark entries coming from different parts of a document, such as title versus body. Later, this information can be used for ranking of search results.

Because **to\_tsvector(NULL)** will return **NULL**, you are advised to use **coalesce** whenever a column might be **NULL**. Here is the recommended method for creating a **tsvector** from a structured document:

```
CREATE TABLE tsearch.tt (id int, title text, keyword text, abstract text, body text, ti tsvector);

INSERT INTO tsearch.tt(id, title, keyword, abstract, body) VALUES (1, 'book', 'literature', 'Ancient poetry', 'Tang poem Song jambic verse');

UPDATE tsearch.tt SET ti =
    setweight(to_tsvector(coalesce(title,'')), 'A') ||
    setweight(to_tsvector(coalesce(keyword,'')), 'B') ||
    setweight(to_tsvector(coalesce(abstract,'')), 'C') ||
    setweight(to_tsvector(coalesce(body,'')), 'D');

DROP TABLE tsearch.tt;
```

Here we have used **setweight** to label the source of each lexeme in the finished **tsvector**, and then merged the labeled **tsvector** values using the **tsvector** concatenation operator **||**. For details about these operations, see [Manipulating tsvector](#).

## 9.3.2 Parsing Queries

GaussDB(DWS) provides functions **to\_tsquery** and **plainto\_tsquery** for converting a query to the **tsquery** data type. **to\_tsquery** offers access to more features than **plainto\_tsquery**, but is less forgiving about its input.

```
to_tsquery([ config regconfig, ] querytext text) returns tsquery
```

**to\_tsquery** creates a **tsquery** value from **querytext**, which must consist of single tokens separated by the Boolean operators **&** (AND), **|** (OR), and **!** (NOT). These operators can be grouped using parentheses. In other words, the input to **to\_tsquery** must already follow the general rules for **tsquery** input, as described in [Text Search Types](#). The difference is that while basic **tsquery** input takes the tokens at face value, **to\_tsquery** normalizes each token to a lexeme using the specified or default configuration, and discards any tokens that are stop words according to the configuration. For example:

```
SELECT to_tsquery('english', 'The & Fat & Rats');
to_tsquery
-----
'fat' & 'rat'
(1 row)
```

As in basic **tsquery** input, **weight(s)** can be attached to each lexeme to restrict it to match only **tsvector** lexemes of those **weight(s)**. For example:

```
SELECT to_tsquery('english', 'Fat | Rats:AB');
to_tsquery
-----
```

```
'fat' | 'rat':AB  
(1 row)
```

Also, the asterisk (\*) can be attached to a lexeme to specify prefix matching:

```
SELECT to_tsquery('supern:*A & star:A*B');  
to_tsquery  
-----  
'supern':*A & 'star':*AB  
(1 row)
```

Such a lexeme will match any word having the specified string and weight in a **tsquery**.

**plainto\_tsquery**([ config regconfig, ] querytext text) returns tsquery

**plainto\_tsquery** transforms unformatted text **querytext** to **tsquery**. The text is parsed and normalized much as for **to\_tsvector**, then the **&** (AND) Boolean operator is inserted between surviving words.

For example:

```
SELECT plainto_tsquery('english', 'The Fat Rats');  
plainto_tsquery  
-----  
'fat' & 'rat'  
(1 row)
```

Note that **plainto\_tsquery** cannot recognize Boolean operators, weight labels, or prefix-match labels in its input:

```
SELECT plainto_tsquery('english', 'The Fat & Rats:C');  
plainto_tsquery  
-----  
'fat' & 'rat' & 'c'  
(1 row)
```

Here, all the input punctuation was discarded as being space symbols.

### 9.3.3 Ranking Search Results

Ranking attempts to measure how relevant documents are to a particular query, so that when there are many matches the most relevant ones can be shown first. GaussDB(DWS) provides two predefined ranking functions, which take into account lexical, proximity, and structural information; that is, they consider how often the query terms appear in the document, how close together the terms are in the document, and how important is the part of the document where they occur. However, the concept of relevancy is vague and application-specific. Different applications might require additional information for ranking, for example, document modification time. The built-in ranking functions are only examples. You can write your own ranking functions and/or combine their results with additional factors to fit your specific needs.

The two ranking functions currently available are:

**ts\_rank**([ weights float4[], ] vector tsvector, query tsquery [, normalization integer ]) returns float4

Ranks vectors based on the frequency of their matching lexemes.

**ts\_rank\_cd**([ weights float4[], ] vector tsvector, query tsquery [, normalization integer ]) returns float4

This function requires positional information in its input. Therefore, it will not work on "stripped" **tsvector** values. It will always return zero.

For both these functions, the optional **weights** argument offers the ability to weigh word instances more or less heavily depending on how they are labeled. The weight arrays specify how heavily to weigh each category of word, in the order:

```
{D-weight, C-weight, B-weight, A-weight}
```

If no **weights** are provided, then these defaults are used: {0.1, 0.2, 0.4, 1.0}

Typically weights are used to mark words from special areas of the document, like the title or an initial abstract, so they can be treated with more or less importance than words in the document body.

Since a longer document has a greater chance of containing a query term it is reasonable to take into account document size. For example, a hundred-word document with five instances of a search word is probably more relevant than a thousand-word document with five instances. Both ranking functions take an integer **normalization** option that specifies whether and how a document's length should impact its rank. The integer option controls several behaviors, so it is a bit mask: you can specify one or more behaviors using a vertical bar (|) (for example, 2|4).

- 0 (the default) ignores the document length
- 1 divides the rank by (1 + Logarithm of the document length)
- 2 divides the rank by the document length
- 4 divides the rank by the mean harmonic distance between extents (this is implemented only by ts\_rank\_cd)
- 8 divides the rank by the number of unique words in document
- 16 divides the rank by (1 + Logarithm of the number of unique words in document)
- 32 divides the rank by (itself + 1)

If more than one flag bit is specified, the transformations are applied in the order listed.

It is important to note that the ranking functions do not use any global information, so it is impossible to produce a fair normalization to 1% or 100% as sometimes desired. Normalization option 32 (**rank/(rank+1)**) can be applied to scale all ranks into the range zero to one, but of course this is just a cosmetic change; it will not affect the ordering of the search results.

Here is an example that selects only the ten highest-ranked matches:

```
SELECT id, title, ts_rank_cd(to_tsvector(body), query) AS rank
FROM tsearch.pgweb, to_tsquery('america') query
WHERE query @@ to_tsvector(body)
ORDER BY rank DESC
LIMIT 10;
id | title | rank
-----+-----+-----
11 | Philology | .2
 2 | Mathematics | .1
12 | Geography | .1
13 | Computer science | .1
(4 rows)
```

This is the same example using normalized ranking:

```
SELECT id, title, ts_rank_cd(to_tsvector(body), query, 32 /* rank/(rank+1) */) AS rank
FROM tsearch.pgweb, to_tsquery('america') query
```



```
WHERE query @@ to_tsvector(body)
ORDER BY rank DESC
LIMIT 10;
id | title | rank
-----+-----+-----
11 | Philology | .166667
 2 | Mathematics | .0909091
12 | Geography | .0909091
13 | Computer science | .0909091
(4 rows)
```

The following example sorts query by Chinese word segmentation:

```
CREATE TABLE tsearch.ts_zhparser(id int, body text);
INSERT INTO tsearch.ts_zhparser VALUES (1, 'Chinese');
INSERT INTO tsearch.ts_zhparser VALUES (2, 'Chinese search');
INSERT INTO tsearch.ts_zhparser VALUES (3 'Search Chinese');
-- Accurate match
SELECT id, body, ts_rank_cd (to_tsvector ('zhparser', body), query) AS rank FROM tsearch.ts_zhparser,
to_tsquery ('Chinese') query WHERE query @@ to_tsvector (body);
id | body | rank
-----+-----+-----
 1 | Chinese | .1
(1 row)

-- Fuzzy match
SELECT id, body, ts_rank_cd (to_tsvector ('zhparser', body), query) AS rank FROM tsearch.ts_zhparser,
to_tsquery ('Chinese') query WHERE query @@ to_tsvector ('zhparser', body);
id | body | rank
-----+-----+-----
 3 | Search Chinese | .1
 1 | Chinese | .1
 2 | Chinese search | .1
(3 rows)
```

Ranking can be expensive since it requires consulting the **tsvector** of each matching document, which can be I/O bound and therefore slow. Unfortunately, it is almost impossible to avoid since practical queries often result in large numbers of matches.

### 9.3.4 Highlighting Results

To present search results it is ideal to show a part of each document and how it is related to the query. Usually, search engines show fragments of the document with marked search terms. GaussDB(DWS) provides function **ts\_headline** that implements this functionality.

```
ts_headline([ config regconfig, ] document text, query tsquery [, options text ]) returns text
```

**ts\_headline** accepts a document along with a query, and returns an excerpt from the document in which terms from the query are highlighted. The configuration to be used to parse the document can be specified by **config**. If **config** is omitted, the **default\_text\_search\_config** configuration is used.

If an options string is specified it must consist of a comma-separated list of one or more **option=value** pairs. The available options are:

- **StartSel, StopSel**: The strings with which to delimit query words appearing in the document, to distinguish them from other excerpted words. You must double-quote these strings if they contain spaces or commas.
- **MaxWords, MinWords**: These numbers determine the longest and shortest headlines to output.

- **ShortWord**: Words of this length or less will be dropped at the start and end of a headline. The default value of three eliminates common English articles.
- **HighlightAll**: Boolean flag. If **true** the whole document will be used as the headline, ignoring the preceding three parameters.
- **MaxFragments**: Maximum number of text excerpts or fragments to display. The default value of zero selects a non-fragment-oriented headline generation method. A value greater than zero selects fragment-based headline generation. This method finds text fragments with as many query words as possible and stretches those fragments around the query words. As a result query words are close to the middle of each fragment and have words on each side. Each fragment will be of at most **MaxWords** and words of length **ShortWord** or less are dropped at the start and end of each fragment. If not all query words are found in the document, then a single fragment of the first **MinWords** in the document will be displayed.
- **FragmentDelimiter**: When more than one fragment is displayed, the fragments will be separated by this string.

Any unspecified options receive these defaults:

```
StartSel=<b>, StopSel=</b>,
MaxWords=35, MinWords=15, ShortWord=3, HighlightAll=FALSE,
MaxFragments=0, FragmentDelimiter=" ... "
```

For example:

```
SELECT ts_headline('english',
'The most common type of search
is to find all documents containing given query terms
and return them in order of their similarity to the
query.',
to_tsquery('english', 'query & similarity'));
      ts_headline
-----
containing given <b>query</b> terms
and return them in order of their <b>similarity</b> to the
<b>query</b>.
(1 row)

SELECT ts_headline('english',
'The most common type of search
is to find all documents containing given query terms
and return them in order of their similarity to the
query.',
to_tsquery('english', 'query & similarity'),
'StartSel = <, StopSel = >');
      ts_headline
-----
containing given <query> terms
and return them in order of their <similarity> to the
<query>.
(1 row)
```

**ts\_headline** uses the original document, not a **tsvector** summary, so it can be slow and should be used with care.

## 9.4 Additional Features

## 9.4.1 Manipulating tsvector

GaussDB(DWS) provides functions and operators that can be used to manipulate documents that are already in tsvector type.

- `tsvector || tsvector`

The tsvector concatenation operator returns a new tsvector which combines the lexemes and positional information of the two tsvectors given as arguments. Positions and weight labels are retained during the concatenation. Positions appearing in the right-hand tsvector are offset by the largest position mentioned in the left-hand tsvector, so that the result is nearly equivalent to the result of performing **to\_tsvector** on the concatenation of the two original document strings. (The equivalence is not exact, because any stop-words removed from the end of the left-hand argument will not affect the result, whereas they would have affected the positions of the lexemes in the right-hand argument if textual concatenation were used.)

One advantage of using concatenation in the tsvector form, rather than concatenating text before applying **to\_tsvector**, is that you can use different configurations to parse different sections of the document. Also, because the **setweight** function marks all lexemes of the given tsvector the same way, it is necessary to parse the text and do **setweight** before concatenating if you want to label different parts of the document with different weights.

- `setweight(vector tsvector, weight "char")` returns tsvector

**setweight** returns a copy of the input tsvector in which every position has been labeled with the given weight, either **A**, **B**, **C**, or **D**. (**D** is the default for new tsvectors and as such is not displayed on output.) These labels are retained when tsvectors are concatenated, allowing words from different parts of a document to be weighted differently by ranking functions.

---

### NOTICE

Note that weight labels apply to positions, not lexemes. If the input tsvector has been stripped of positions then **setweight** does nothing.

---

- `length(vector tsvector)` returns integer

Returns the number of lexemes stored in the vector.

- `strip(vector tsvector)` returns tsvector

Returns a tsvector which lists the same lexemes as the given tsvector, but which lacks any position or weight information. While the returned tsvector is much less useful than an unstripped tsvector for relevance ranking, it will usually be much smaller.

## 9.4.2 Manipulating Queries

GaussDB(DWS) provides functions and operators that can be used to manipulate queries that are already in tsquery type.

- `tsquery && tsquery`

Returns the AND-combination of the two given tsqueries.

- `tsquery || tsquery`  
Returns the OR-combination of the two given `tsqueries`.
- `!! tsquery`  
Returns the negation (NOT) of the given `tsquery`.
- `numnode(query tsquery)` returns integer  
Returns the number of nodes (lexemes plus operators) in a **tsquery**. This function is useful to determine if the query is meaningful (returns > 0), or contains only stop words (returns 0). For example:

```
SELECT numnode(plainto_tsquery('the any'));
NOTICE: text-search query contains only stop words or doesn't contain lexemes, ignored
CONTEXT: referenced column: numnode
 numnode
-----
      0

SELECT numnode('foo & bar)::tsquery);
 numnode
-----
      3
```

- `querytree(query tsquery)` returns text  
Returns the portion of a **tsquery** that can be used for searching an index. This function is useful for detecting unindexable queries, for example those containing only stop words or only negated terms. For example:

```
SELECT querytree(to_tsquery('!defined'));
 querytree
-----
      T
(1 row)
```

### 9.4.3 Rewriting Queries

The **ts\_rewrite** family of functions searches a given **tsquery** for occurrences of a target subquery, and replace each occurrence with a substitute subquery. In essence this operation is a **tsquery** specific version of substring replacement. A target and substitute combination can be thought of as a query rewrite rule. A collection of such rewrite rules can be a powerful search aid. For example, you can expand the search using synonyms (that is, new york, big apple, nyc, gotham) or narrow the search to direct the user to some hot topic.

- `ts_rewrite (query tsquery, target tsquery, substitute tsquery)` returns `tsquery`  
This form of **ts\_rewrite** simply applies a single rewrite rule: **target** is replaced by **substitute** wherever it appears in query. For example:

```
SELECT ts_rewrite('a & b)::tsquery, 'a)::tsquery, 'c)::tsquery);
 ts_rewrite
-----
 'b' & 'c'
```

- `ts_rewrite (query tsquery, select text)` returns `tsquery`  
This form of **ts\_rewrite** accepts a starting query and a SQL select command, which is given as a text string. The **select** must yield two columns of **tsquery** type. For each row of the select result, occurrences of the first column value (the target) are replaced by the second column value (the substitute) within the current **query** value.

 **NOTE**

Note that when multiple rewrite rules are applied in this way, the order of application can be important; so in practice you will want the source query to **ORDER BY** some ordering key.

Consider a real-life astronomical example. We will expand query supernovae using table-driven rewriting rules:

```
CREATE TABLE tsearch.aliaes (id int, t tsquery, s tsquery);
INSERT INTO tsearch.aliaes VALUES(1, to_tsquery('supernovae'), to_tsquery('supernovae|sn'));
SELECT ts_rewrite(to_tsquery('supernovae & crab'), 'SELECT t, s FROM tsearch.aliaes');
      ts_rewrite
-----
'crab' & ( 'supernova' | 'sn' )
```

We can change the rewriting rules just by updating the table:

```
UPDATE tsearch.aliaes
SET s = to_tsquery('supernovae|sn & !nebulae')
WHERE t = to_tsquery('supernovae');
SELECT ts_rewrite(to_tsquery('supernovae & crab'), 'SELECT t, s FROM tsearch.aliaes');
      ts_rewrite
-----
'crab' & ( 'supernova' | 'sn' & '!nebula' )
```

Rewriting can be slow when there are many rewriting rules, since it checks every rule for a possible match. To filter out obvious non-candidate rules we can use the containment operators for the **tsquery** type. In the example below, we select only those rules which might match the original query:

```
SELECT ts_rewrite('a & b':tsquery, 'SELECT t,s FROM tsearch.aliaes WHERE "a & b":tsquery @> t');
      ts_rewrite
-----
'b' & 'a'
(1 row)
DROP TABLE ts_rewrite;
```

## 9.4.4 Gathering Document Statistics

The function **ts\_stat** is useful for checking your configuration and for finding stop-word candidates.

```
ts_stat(sqlquery text, [ weights text, ]
      OUT word text, OUT ndoc integer,
      OUT nentry integer) returns setof record
```

**sqlquery** is a text value containing an SQL query which must return a single **tsvector** column. **ts\_stat** executes the query and returns statistics about each distinct lexeme (word) contained in the **tsvector** data. The columns returned are

- **word text**: the value of a lexeme
- **ndoc integer**: number of documents (**tsvectors**) the word occurred in
- **nentry integer**: total number of occurrences of the word

If **weights** are supplied, only occurrences having one of those weights are counted. For example, to find the ten most frequent words in a document collection:

```
SELECT * FROM ts_stat('SELECT to_tsvector("english", sr_reason_sk) FROM tpccds.store_returns WHERE sr_customer_sk < 10') ORDER BY nentry DESC, ndoc DESC, word LIMIT 10;;
```

```

word | ndoc | nentry
-----+-----
32 | 2 | 2
33 | 2 | 2
1 | 1 | 1
10 | 1 | 1
13 | 1 | 1
14 | 1 | 1
15 | 1 | 1
17 | 1 | 1
20 | 1 | 1
22 | 1 | 1
(10 rows)

```

The same, but counting only word occurrences with weight **A** or **B**:

```

SELECT * FROM ts_stat('SELECT to_tsvector("english", sr_reason_sk) FROM tpeds.store_returns WHERE
sr_customer_sk < 10', 'a') ORDER BY nentry DESC, ndoc DESC, word LIMIT 10;
word | ndoc | nentry
-----+-----
(0 rows)

```

## 9.5 Parsers

Text search parsers are responsible for splitting raw document text into tokens and identifying each token's type, where the set of types is defined by the parser itself. Note that a parser does not modify the text at all — it simply identifies plausible word boundaries. Because of this limited scope, there is less need for application-specific custom parsers than there is for custom dictionaries.

Currently, GaussDB(DWS) provides the following built-in parsers: `pg_catalog.default` for English configuration, and `pg_catalog.ngram`, `pg_catalog.zhparser`, and `pg_catalog.pound` for full text search in texts containing Chinese, or both Chinese and English.

The built-in parser is named `pg_catalog.default`. It recognizes 23 token types, shown in [Table 9-1](#).

**Table 9-1** Default parser's token types

| Alias                         | Description                         | Examples                                      |
|-------------------------------|-------------------------------------|-----------------------------------------------|
| <code>asciiword</code>        | Word, all ASCII letters             | elephant                                      |
| <code>word</code>             | Word, all letters                   | mañana                                        |
| <code>numword</code>          | Word, letters and digits            | beta1                                         |
| <code>asciihword</code>       | Hyphenated word, all ASCII          | up-to-date                                    |
| <code>hword</code>            | Hyphenated word, all letters        | lógico-matemática                             |
| <code>numhword</code>         | Hyphenated word, letters and digits | postgresql-beta1                              |
| <code>hword_ascii part</code> | Hyphenated word part, all ASCII     | postgresql in the context<br>postgresql-beta1 |

| Alias         | Description                              | Examples                                                 |
|---------------|------------------------------------------|----------------------------------------------------------|
| hword_part    | Hyphenated word part, all letters        | lógico or matemática in the context lógico-matemática    |
| hword_numpart | Hyphenated word part, letters and digits | beta1 in the context postgresql-beta1                    |
| email         | Email address                            | foo@example.com                                          |
| protocol      | Protocol head                            | http://                                                  |
| url           | URL                                      | example.com/stuff/index.html                             |
| host          | Host                                     | example.com                                              |
| url_path      | URL path                                 | /stuff/index.html, in the context of a URL               |
| file          | File or path name                        | /usr/local/foo.txt, if not within a URL                  |
| sfloat        | Scientific notation                      | -1.23E+56                                                |
| float         | Decimal notation                         | -1.234                                                   |
| int           | Signed integer                           | -1234                                                    |
| uint          | Unsigned integer                         | 1234                                                     |
| version       | Version number                           | 8.3.0                                                    |
| tag           | XML tag                                  | <a href="dictionaries.html">                             |
| entity        | XML entity                               | &amp;                                                    |
| blank         | Space symbols                            | (any whitespace or punctuation not otherwise recognized) |

Note: The parser's notion of a "letter" is determined by the database's locale setting, specifically **lc\_ctype**. Words containing only the basic ASCII letters are reported as a separate token type, since it is sometimes useful to distinguish them. In most European languages, token types word and asciiword should be treated alike.

**email** does not support all valid email characters as defined by RFC 5322. Specifically, the only non-alphanumeric characters supported for email user names are period, dash, and underscore.

It is possible for the parser to identify overlapping tokens in the same piece of text. As an example, a hyphenated word will be reported both as the entire word and as each component:

```
SELECT alias, description, token FROM ts_debug('english','foo-bar-beta1');
  alias | description | token
-----+-----+-----
numhword | Hyphenated word, letters and digits | foo-bar-beta1
```

```

hword_asciipart | Hyphenated word part, all ASCII | foo
blank | Space symbols | -
hword_asciipart | Hyphenated word part, all ASCII | bar
blank | Space symbols | -
hword_numpart | Hyphenated word part, letters and digits | beta1
    
```

This behavior is desirable since it allows searches to work for both the whole compound word and for components. Here is another instructive example:

```

SELECT alias, description, token FROM ts_debug('english','http://example.com/stuff/index.html');
alias | description | token
-----+-----+-----
protocol | Protocol head | http://
url | URL | example.com/stuff/index.html
host | Host | example.com
url_path | URL path | /stuff/index.html
    
```

N-gram is a mechanical word segmentation method, and applies to no semantic Chinese segmentation scenarios. The N-gram segmentation method ensures the completeness of the segmentation. However, to cover all the possibilities, it but adds unnecessary words to the index, resulting in a large number of index items. N-gram supports Chinese coding, including GBK and UTF-8. Six built-in token types are shown in [Table 9-2](#).

**Table 9-2** Token types

| Alias       | Description     |
|-------------|-----------------|
| zh_words    | chinese words   |
| en_word     | english word    |
| numeric     | numeric data    |
| alnum       | alnum string    |
| grapsymbol  | graphic symbol  |
| multisymbol | multiple symbol |

Zhparser is a dictionary-based semantic word segmentation method. The bottom-layer calls the Simple Chinese Word Segmentation (SCWS) algorithm (<https://github.com/hightman/scws>), which applies to Chinese segmentation scenarios. SCWS is a term frequency and dictionary-based mechanical Chinese words engine. It can split a whole paragraph Chinese text into words. The two Chinese coding formats, GBK and UTF-8, are supported. The 26 built-in token types are shown in [Table 9-3](#).

**Table 9-3** Token types

| Alias | Description     |
|-------|-----------------|
| A     | Adjective       |
| B     | Differentiation |
| C     | Conjunction     |



| Alias | Description                |
|-------|----------------------------|
| D     | Adverb                     |
| E     | Exclamation                |
| F     | Position                   |
| G     | Lexeme                     |
| H     | Preceding element          |
| I     | Idiom                      |
| J     | Acronyms and abbreviations |
| K     | Subsequent element         |
| L     | Common words               |
| M     | Numeral                    |
| N     | Noun                       |
| O     | Onomatopoeia               |
| P     | Preposition                |
| Q     | Quantifiers                |
| R     | Pronoun                    |
| S     | Space                      |
| T     | Time                       |
| U     | Auxiliary word             |
| V     | Verb                       |
| W     | Punctuation                |
| X     | Unknown                    |
| Y     | Interjection               |
| Z     | Status words               |

Pound segments words in a fixed format. It is used to segment to-be-parsed nonsense Chinese and English words that are separated by fixed separators. It supports Chinese encoding (including GBK and UTF8) and English encoding (including ASCII). Pound has six pre-configured token types (as listed in [Table 9-4](#)) and supports five separators (as listed in [Table 9-5](#)). The default, the separator is #. Pound The maximum length of a token is 256 characters.

**Table 9-4** Token types

| Alias       | Description     |
|-------------|-----------------|
| zh_words    | chinese words   |
| en_word     | english word    |
| numeric     | numeric data    |
| alnum       | alnum string    |
| grapsymbol  | graphic symbol  |
| multisymbol | multiple symbol |

**Table 9-5** Separator types

| Delimiter | Description       |
|-----------|-------------------|
| @         | Special character |
| #         | Special character |
| \$        | Special character |
| %         | Special character |
| /         | Special character |

## 9.6 Dictionaries

### 9.6.1 Overview

A dictionary is used to define stop words, that is, words to be ignored in full-text retrieval.

A dictionary can also be used to normalize words so that different derived forms of the same word will match. A normalized word is called a lexeme.

In addition to improving retrieval quality, normalization and removal of stop words can reduce the size of the **tsvector** representation of a document, thereby improving performance. Normalization and removal of stop words do not always have linguistic meaning. Users can define normalization and removal rules in dictionary definition files based on application environments.

A dictionary is a program that receives a token as input and returns:

- An array of lexemes if the input token is known to the dictionary (note that one token can produce more than one lexeme).
- A single lexeme to replace the original token with a new token to be passed to subsequent dictionaries (a dictionary that does this is called a filtering dictionary).

- An empty array if the input token is known to the dictionary but is a stop word.
- **NULL** if the dictionary does not recognize the token.

GaussDB(DWS) provides predefined dictionaries for many languages and also provides five predefined dictionary templates, **Simple**, **Synonym**, **Thesaurus**, **Ispell**, and **Snowball**. These templates can be used to create new dictionaries with custom parameters.

When using full-text retrieval, you are advised to:

- In the text search configuration, configure a parser together with a set of dictionaries to process the parser's output tokens. For each token type that the parser can return, a separate list of dictionaries is specified by the configuration. When a token of that type is found by the parser, each dictionary in the list is consulted in turn, until a dictionary recognizes it as a known word. If it is identified as a stop word, or no dictionary recognizes the token, it will be discarded and not indexed or searched for. Generally, the first dictionary that returns a non-**NULL** output determines the result, and any remaining dictionaries are not consulted. However, a filtering dictionary can replace the input token with a modified one, which is then passed to subsequent dictionaries.
- The general rule for configuring a list of dictionaries is to place first the most narrow, most specific dictionary, then the more general dictionaries, finishing with a very general dictionary, like a **Snowball** stemmer dictionary or a **Simple** dictionary, which recognizes everything. In the following example, for an astronomy-specific search (**astro\_en** configuration), you can configure the token type **asciiword** (ASCII word) with a **Synonym** dictionary of astronomical terms, a general English **Ispell** dictionary, and a **Snowball** English stemmer dictionary:

```
ALTER TEXT SEARCH CONFIGURATION astro_en
ADD MAPPING FOR asciiword WITH astro_syn, english_ispell, english_stem;
```

A filtering dictionary can be placed anywhere in the list, except at the end where it would be useless. Filtering dictionaries are useful to partially normalize words to simplify the task of later dictionaries.

## 9.6.2 Stop Words

Stop words are words that are very common, appear in almost every document, and have no discrimination value. Therefore, they can be ignored in the context of full text searching. Each type of dictionaries treats stop words in different ways. For example, **Ispell** dictionaries first normalize words and then check the list of stop words, while **Snowball** dictionaries first check the list of stop words.

For example, every English text contains words like **a** and **the**, so it is useless to store them in an index. However, stop words affect the positions in **tsvector**, which in turn affect ranking.

```
SELECT to_tsvector('english','in the list of stop words');
to_tsvector
-----
'list':3 'stop':5 'word':6
```

The missing positions 1, 2, and 4 are because of stop words. Ranks calculated for documents with and without stop words are quite different:

```
SELECT ts_rank_cd (to_tsvector('english','in the list of stop words'), to_tsquery('list & stop'));
ts_rank_cd
```

```
-----  
.05  
  
SELECT ts_rank_cd (to_tsvector('english','list stop words'), to_tsquery('list & stop'));  
ts_rank_cd  
-----  
.1
```

## 9.6.3 Simple Dictionary

A **Simple** dictionary operates by converting the input token to lower case and checking it against a list of stop words. If the token is found in the list, an empty array will be returned, causing the token to be discarded. If it is not found, the lower-cased form of the word is returned as the normalized lexeme. In addition, you can set **Accept** to **false** for **Simple** dictionaries (default: **true**) to report non-stop-words as unrecognized, allowing them to be passed on to the next dictionary in the list.

### Precautions

- Most types of dictionaries rely on dictionary configuration files. The name of a configuration file can only be lowercase letters, digits, and underscores (\_).
- A dictionary cannot be created in **pg\_temp** mode.
- Dictionary configuration files must be stored in UTF-8 encoding. They will be translated to the actual database encoding, if that is different, when they are read into the server.
- Generally, a session will read a dictionary configuration file only once, when it is first used within the session. To modify a configuration file, run the **ALTER TEXT SEARCH DICTIONARY** statement to update and reload the file.

### Procedure

**Step 1** Create a **Simple** dictionary.

```
CREATE TEXT SEARCH DICTIONARY public.simple_dict (  
    TEMPLATE = pg_catalog.simple,  
    STOPWORDS = english  
);
```

**english.stop** is the full name of a file of stop words. For details about the syntax and parameters for creating a **Simple** dictionary, see [CREATE TEXT SEARCH DICTIONARY](#).

**Step 2** Use the **Simple** dictionary.

```
SELECT ts_lexize('public.simple_dict','Yes');  
ts_lexize  
-----  
{yes}  
(1 row)  
  
SELECT ts_lexize('public.simple_dict','The');  
ts_lexize  
-----  
{}  
(1 row)
```

**Step 3** Set **Accept=false** so that the **Simple** dictionary returns **NULL** instead of a lower-cased non-stop word.

```
ALTER TEXT SEARCH DICTIONARY public.simple_dict ( Accept = false );  
ALTER TEXT SEARCH DICTIONARY
```

```
SELECT ts_lexize('public.simple_dict','YeS');
ts_lexize
-----
(1 row)

SELECT ts_lexize('public.simple_dict','The');
ts_lexize
-----
{}
(1 row)
```

----End

## 9.6.4 Synonym Dictionary

A **Synonym** dictionary is used to define, identify, and convert synonyms of a token. Phrases are not supported. Synonyms of phrases can be defined in a **Thesaurus** dictionary. For details, see [Thesaurus Dictionary](#).

### Examples

- A **Synonym** dictionary can be used to overcome linguistic problems. For example, to prevent an English stemmer dictionary from reducing the word 'Paris' to 'pari', define a **Paris paris** line in the **Synonym** dictionary and put it before the **english\_stem** dictionary.

```
SELECT * FROM ts_debug('english', 'Paris');
alias | description | token | dictionaries | dictionary | lexemes
-----+-----+-----+-----+-----+-----
asciiword | Word, all ASCII | Paris | {english_stem} | english_stem | {pari}
(1 row)

CREATE TEXT SEARCH DICTIONARY my_synonym (
  TEMPLATE = synonym,
  SYNONYMS = my_synonyms,
  FILEPATH = 'obs://bucket_name/path accesskey=ak secretkey=sk region=rg'
);

ALTER TEXT SEARCH CONFIGURATION english
ALTER MAPPING FOR asciiword
WITH my_synonym, english_stem;

SELECT * FROM ts_debug('english', 'Paris');
alias | description | token | dictionaries | dictionary | lexemes
-----+-----+-----+-----+-----+-----
asciiword | Word, all ASCII | Paris | {my_synonym,english_stem} | my_synonym | {paris}
(1 row)

SELECT * FROM ts_debug('english', 'paris');
alias | description | token | dictionaries | dictionary | lexemes
-----+-----+-----+-----+-----+-----
asciiword | Word, all ASCII | Paris | {my_synonym,english_stem} | my_synonym | {paris}
(1 row)

ALTER TEXT SEARCH DICTIONARY my_synonym ( CASESENSITIVE=true);

SELECT * FROM ts_debug('english', 'Paris');
alias | description | token | dictionaries | dictionary | lexemes
-----+-----+-----+-----+-----+-----
asciiword | Word, all ASCII | Paris | {my_synonym,english_stem} | my_synonym | {paris}
(1 row)

SELECT * FROM ts_debug('english', 'paris');
alias | description | token | dictionaries | dictionary | lexemes
-----+-----+-----+-----+-----+-----
```

```
asciiword | Word, all ASCII | Paris | {my_synonym,english_stem} | my_synonym | {pari}
(1 row)
```

The full name of the **Synonym** dictionary file is **my\_synonyms.syn**, and the dictionary is stored in the **obs://bucket\_name/path accesskey=ak secretkey=sk region=rg** directory. For details about the syntax and parameters for creating a **Synonym** dictionary, see [CREATE TEXT SEARCH DICTIONARY](#).

- An asterisk (\*) can be placed at the end of a synonym in the configuration file. This indicates that the synonym is a prefix. The asterisk is ignored when the entry is used in `to_tsvector()`, but when it is used in `to_tsquery()`, the result will be a query item with the prefix match marker (see [Manipulating Queries](#)).

Assume that the content in the dictionary file **synonym\_sample.syn** is as follows:

```
postgres    pgsql
postgresql  pgsql
postgre pgsql
gogle    googl
indices index*
```

Create and use a dictionary.

```
CREATE TEXT SEARCH DICTIONARY syn (
    TEMPLATE = synonym,
    SYNONYMS = synonym_sample
);
```

```
SELECT ts_lexize('syn','indices');
ts_lexize
-----
{index}
(1 row)
```

```
CREATE TEXT SEARCH CONFIGURATION tst (copy=simple);
```

```
ALTER TEXT SEARCH CONFIGURATION tst ALTER MAPPING FOR asciiword WITH syn;
```

```
SELECT to_tsvector('tst','indices');
to_tsvector
-----
'index':1
(1 row)
```

```
SELECT to_tsquery('tst','indices');
to_tsquery
-----
'index':*
(1 row)
```

```
SELECT 'indexes are very useful':tsvector;
tsvector
-----
'are' 'indexes' 'useful' 'very'
(1 row)
```

```
SELECT 'indexes are very useful':tsvector @@ to_tsquery('tst','indices');
?column?
-----
t
(1 row)
```

## 9.6.5 Thesaurus Dictionary

A **Thesaurus** dictionary (sometimes abbreviated as TZ) is a collection of relationships between words and phrases, such as broader terms (BT), narrower terms (NT), preferred terms, non-preferred terms, and related terms. Based on definitions in the dictionary file, a TZ replaces all non-preferred terms by one preferred term and, optionally, preserves the original terms for indexing as well. A TZ is an extension of a **Synonym** dictionary with added phrase support.

### Precautions

- A TZ has the capability to recognize phrases and therefore it must remember its state and interact with the parser to determine whether to handle the next token or stop accumulation. A TZ must be configured carefully. For example, if an AZ is configured to handle only **asciword** tokens, a TZ definition like **one 7** will not work because the token type **uint** is not assigned to the TZ.
- TZs are used during indexing, so any change in the TZ's parameters requires reindexing. For most other dictionary types, small changes such as adding or removing stop words does not force reindexing.

### Procedure

#### Step 1 Create a TZ named **thesaurus\_astro**.

**thesaurus\_astro** is a simple astronomical TZ that defines two astronomical word combinations (word+synonym).

```
supernovae stars : sn  
crab nebulae : crab
```

Run the following statement to create the TZ:

```
CREATE TEXT SEARCH DICTIONARY thesaurus_astro (  
  TEMPLATE = thesaurus,  
  DictFile = thesaurus_astro,  
  Dictionary = pg_catalog.english_stem,  
  FILEPATH = 'obs://bucket_name/path accesskey=ak secretkey=sk region=rg'  
);
```

The full name of the TZ file is **thesaurus\_astro.ths**, and the TZ is stored in the **obs://bucket\_name/path accesskey=ak secretkey=sk region=rg** directory. **pg\_catalog.english\_stem** is the subdictionary (a **Snowball** English stemmer) used for input normalization. The subdictionary has its own configuration (for example, stop words), which is not shown here. For details about the syntax and parameters for creating a TZ, see [CREATE TEXT SEARCH DICTIONARY](#).

#### Step 2 Bind the TZ to the desired token types in the text search configuration.

```
ALTER TEXT SEARCH CONFIGURATION russian  
  ALTER MAPPING FOR asciword, asciihword, hword_asciipart  
  WITH thesaurus_astro, english_stem;
```

#### Step 3 Use the TZ.

- Test the TZ.

The **ts\_lexize** function is not very useful for testing the TZ because the function processes its input as a single token. Instead, you can use the **plainto\_tsquery**, **to\_tsvector**, or **to\_tsquery** function which will break their input strings into multiple tokens.

```
SELECT plainto_tsquery('russian','supernova star');  
plainto_tsquery
```

```
-----  
'sn'  
(1 row)  
  
SELECT to_tsvector('russian','supernova star');  
to_tsvector  
-----  
'sn':1  
(1 row)  
  
SELECT to_tsquery('russian','"supernova star"');  
to_tsquery  
-----  
'sn'  
(1 row)
```

**supernova star** matches **supernovae stars** in **thesaurus\_astro** because the **english\_stem** stemmer is specified in the **thesaurus\_astro** definition. The stemmer removed **e** and **s**.

- To index the original phrase, include it in the right-hand part of the definition.  
supernovae stars : sn supernovae stars

```
ALTER TEXT SEARCH DICTIONARY thesaurus_astro (  
  DictFile = thesaurus_astro,  
  FILEPATH = 'file:///home/dicts/');  
  
SELECT plainto_tsquery('russian','supernova star');  
plainto_tsquery  
-----  
'sn' & 'supernova' & 'star'  
(1 row)
```

----End

## 9.6.6 Ispell Dictionary

An **Ispell** dictionary is a morphological dictionary, which can normalize different linguistic forms of a word into the same lexeme. For example, an English **Ispell** dictionary can match all declensions and conjugations of the search term **bank**, such as, **banking**, **banked**, **banks**, **banks'**, and **bank's**.

GaussDB(DWS) does not provide any predefined **Ispell** dictionaries or dictionary files. The **.dict** files and **.affix** files support multiple open-source dictionary formats, including **Ispell**, **MySpell**, and **Hunspell**.

### Procedure

- Step 1** Obtain the dictionary definition file (**.dict**) and affix file (**.affix**).

You can use an open-source dictionary. The name extensions of the open-source dictionary may be **.aff** and **.dic**. In this case, you need to change them to **.affix** and **.dict**. In addition, for some dictionary files (for example, Norwegian dictionary files), you need to run the following commands to convert the character encoding to UTF-8:

```
iconv -f ISO_8859-1 -t UTF-8 -o nn_no.affix nn_NO.aff  
iconv -f ISO_8859-1 -t UTF-8 -o nn_no.dict nn_NO.dic
```

- Step 2** Create an **Ispell** dictionary.

```
CREATE TEXT SEARCH DICTIONARY norwegian_ispell (  
  TEMPLATE = ispell,  
  DictFile = nn_no,  
  AffFile = nn_no,
```



```
FilePath = 'file:///home/dicts'obs://bucket_name/path accesskey=ak secretkey=sk region=rg'  
);
```

The full names of the **Ispell** dictionary files are **nn\_no.dict** and **nn\_no.affix**, and the dictionary is stored in the **obs://bucket\_name/path accesskey=ak secretkey=sk region=rg** directory. For details about the syntax and parameters for creating an **Ispell** dictionary, see [CREATE TEXT SEARCH DICTIONARY](#).

**Step 3** Use the **Ispell** dictionary to split compound words.

```
SELECT ts_lexize('norwegian_isspell', 'sjokoladefabrikk');  
   ts_lexize  
-----  
{sjokolade,fabrikk}  
(1 row)
```

**MySpell** does not support compound words. **Hunspell** supports compound words. GaussDB(DWS) supports only the basic compound word operations of **Hunspell**. Generally, **Ispell** dictionaries recognize a limited set of words, so they should be followed by another broader dictionary, for example, a **Snowball** dictionary, which recognizes everything.

----End

## 9.6.7 Snowball Dictionary

A **Snowball** dictionary is based on a project by Martin Porter and is used for stem analysis, providing stemming algorithms for many languages. GaussDB(DWS) provides predefined **Snowball** dictionaries of many languages. You can query the **PG\_TS\_DICT** system catalog to view the predefined **Snowball** dictionaries and supported stemming algorithms.

A **Snowball** dictionary recognizes everything, no matter whether it is able to simplify the word. Therefore, it should be placed at the end of the dictionary list. It is useless to place it before any other dictionary because a token will never pass it through to the next dictionary.

For details about the syntax of **Snowball** dictionaries, see [CREATE TEXT SEARCH DICTIONARY](#).

## 9.7 Configuration Examples

Text search configuration specifies the following components required for converting a document into a **tsvector**:

- A parser, decomposes a text into tokens.
- Dictionary list, converts each token into a lexeme.

Each time when the **to\_tsvector** or **to\_tsquery** function is invoked, a text search configuration is required to specify a processing procedure. The GUC parameter **default\_text\_search\_config** specifies the default text search configuration, which will be used if the text search function does not explicitly specify a text search configuration.

GaussDB(DWS) provides some predefined text search configurations. You can also create user-defined text search configurations. In addition, to facilitate the management of text search objects, multiple **gsql** meta-commands are provided to display information about text search objects.

## Procedure

- Step 1** Create a text search configuration **ts\_conf** by copying the predefined text search configuration **english**.

```
CREATE TEXT SEARCH CONFIGURATION ts_conf ( COPY = pg_catalog.english );  
CREATE TEXT SEARCH CONFIGURATION
```

- Step 2** Create a **Synonym** dictionary.

Assume that the definition file **pg\_dict.syn** of the **Synonym** dictionary contains the following contents:

```
postgres pg  
pgsql pg  
postgresql pg
```

Run the following statement to create the **Synonym** dictionary:

```
CREATE TEXT SEARCH DICTIONARY pg_dict (  
    TEMPLATE = synonym,  
    SYNONYMS = pg_dict,  
    FILEPATH = 'obs://bucket_name/path accesskey=ak secretkey=sk region=rg'  
);
```

- Step 3** Create an **IsPELL** dictionary **english\_ispell** (the dictionary definition file is from the open source dictionary).

```
CREATE TEXT SEARCH DICTIONARY english_ispell (  
    TEMPLATE = ispell,  
    DictFile = english,  
    AffFile = english,  
    StopWords = english,  
    FILEPATH = 'obs://bucket_name/path accesskey=ak secretkey=sk region=rg'  
);
```

- Step 4** Modify the text search configuration **ts\_conf** and change the dictionary list for tokens of certain types. For details about token types, see [Parsers](#).

```
ALTER TEXT SEARCH CONFIGURATION ts_conf  
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart,  
        word, hword, hword_part  
    WITH pg_dict, english_ispell, english_stem;
```

- Step 5** In the text search configuration, set non-index or set the search for tokens of certain types.

```
ALTER TEXT SEARCH CONFIGURATION ts_conf  
    DROP MAPPING FOR email, url, url_path, sfloat, float;
```

- Step 6** Use the text retrieval commissioning function **ts\_debug()** to test the text search configuration **ts\_conf**.

```
SELECT * FROM ts_debug('ts_conf', '  
PostgreSQL, the highly scalable, SQL compliant, open source object-relational  
database management system, is now undergoing beta testing of the next  
version of our software.  
' );
```

- Step 7** You can set the default text search configuration of the current session to **ts\_conf**. This setting is valid only for the current session.

```
\dF+ ts_conf  
    Text search configuration "public.ts_conf"  
Parser: "pg_catalog.default"  
Token  | Dictionaries  
-----+-----  
asciihword | pg_dict,english_ispell,english_stem  
asciiword  | pg_dict,english_ispell,english_stem  
file       | simple  
host       | simple
```

```

hword          | pg_dict,english_ispell,english_stem
hword_asciipart | pg_dict,english_ispell,english_stem
hword_numpart  | simple
hword_part     | pg_dict,english_ispell,english_stem
int            | simple
numhword       | simple
numword        | simple
uint          | simple
version        | simple
word           | pg_dict,english_ispell,english_stem

```

```

SET default_text_search_config = 'public.ts_conf';
SET
SHOW default_text_search_config;
default_text_search_config

```

```

-----
public.ts_conf
(1 row)

```

----End

## 9.8 Testing and Debugging Text Search

### 9.8.1 Testing a Configuration

The function `ts_debug` allows easy testing of a text search configuration.

```

ts_debug([ config regconfig, ] document text,
         OUT alias text,
         OUT description text,
         OUT token text,
         OUT dictionaries regdictionary[],
         OUT dictionary regdictionary,
         OUT lexemes text[])
returns setof record

```

**ts\_debug** displays information about every token of document as produced by the parser and processed by the configured dictionaries. It uses the configuration specified by **config**, or **default\_text\_search\_config** if that argument is omitted.

**ts\_debug** returns one row for each token identified in the text by the parser. The columns returned are:

- alias text — short name of the token type
- description text — description of the token type
- token text — text of the token
- **dictionaries regdictionary[]** — the dictionaries selected by the configuration for this token type
- **dictionary regdictionary**: the dictionary that recognized the token, or NULL if none did
- **lexemes text[]**: the lexeme(s) produced by the dictionary that recognized the token, or NULL if none did; an empty array ({} ) means the token was recognized as a stop word

Here is a simple example:

```

SELECT * FROM ts_debug('english','a fat cat sat on a mat - it ate a fat rats');
alias | description | token | dictionaries | dictionary | lexemes

```

```

asciiword | Word, all ASCII | a | {english_stem} | english_stem | {}
blank | Space symbols | | {}
asciiword | Word, all ASCII | fat | {english_stem} | english_stem | {fat}
blank | Space symbols | | {}
asciiword | Word, all ASCII | cat | {english_stem} | english_stem | {cat}
blank | Space symbols | | {}
asciiword | Word, all ASCII | sat | {english_stem} | english_stem | {sat}
blank | Space symbols | | {}
asciiword | Word, all ASCII | on | {english_stem} | english_stem | {}
blank | Space symbols | | {}
asciiword | Word, all ASCII | a | {english_stem} | english_stem | {}
blank | Space symbols | | {}
asciiword | Word, all ASCII | mat | {english_stem} | english_stem | {mat}
blank | Space symbols | | {}
blank | Space symbols | - | {}
asciiword | Word, all ASCII | it | {english_stem} | english_stem | {}
blank | Space symbols | | {}
asciiword | Word, all ASCII | ate | {english_stem} | english_stem | {ate}
blank | Space symbols | | {}
asciiword | Word, all ASCII | a | {english_stem} | english_stem | {}
blank | Space symbols | | {}
asciiword | Word, all ASCII | fat | {english_stem} | english_stem | {fat}
blank | Space symbols | | {}
asciiword | Word, all ASCII | rats | {english_stem} | english_stem | {rat}
(24 rows)

```

## 9.8.2 Testing a Parser

The `ts_parse` function allows direct testing of a text search parser.

```

ts_parse(parser_name text, document text,
         OUT tokid integer, OUT token text) returns setof record

```

`ts_parse` parses the given **document** and returns a series of records, one for each token produced by parsing. Each record includes a **tokid** showing the assigned token type and a **token** which is the text of the token. For example:

```

SELECT * FROM ts_parse('default', '123 - a number');
tokid | token
-----+-----
    22 | 123
    12 |
    12 | -
     1 | a
    12 |
     1 | number
(6 rows)

```

```

ts_token_type(parser_name text, OUT tokid integer,
             OUT alias text, OUT description text) returns setof record

```

`ts_token_type` returns a table which describes each type of token the specified parser can recognize. For each token type, the table gives the integer **tokid** that the parser uses to label a token of that type, the **alias** that names the token type in configuration commands, and a short description. For example:

```

SELECT * FROM ts_token_type('default');
tokid | alias | description
-----+-----
     1 | asciiword | Word, all ASCII
     2 | word | Word, all letters
     3 | numword | Word, letters and digits
     4 | email | Email address
     5 | url | URL
     6 | host | Host
     7 | sfloat | Scientific notation
     8 | version | Version number
     9 | hword_numpart | Hyphenated word part, letters and digits

```

|    |  |                 |  |                                     |
|----|--|-----------------|--|-------------------------------------|
| 10 |  | hword_part      |  | Hyphenated word part, all letters   |
| 11 |  | hword_asciipart |  | Hyphenated word part, all ASCII     |
| 12 |  | blank           |  | Space symbols                       |
| 13 |  | tag             |  | XML tag                             |
| 14 |  | protocol        |  | Protocol head                       |
| 15 |  | numhword        |  | Hyphenated word, letters and digits |
| 16 |  | asciihword      |  | Hyphenated word, all ASCII          |
| 17 |  | hword           |  | Hyphenated word, all letters        |
| 18 |  | url_path        |  | URL path                            |
| 19 |  | file            |  | File or path name                   |
| 20 |  | float           |  | Decimal notation                    |
| 21 |  | int             |  | Signed integer                      |
| 22 |  | uint            |  | Unsigned integer                    |
| 23 |  | entity          |  | XML entity                          |

(23 rows)

### 9.8.3 Testing a Dictionary

The `ts_lexize` function facilitates dictionary testing.

`ts_lexize(dict regdictionary, token text)` returns `text[]` `ts_lexize` returns an array of lexemes if the input `token` is known to the dictionary, or an empty array if the token is known to the dictionary but it is a stop word, or `NULL` if it is an unknown word.

For example:

```
SELECT ts_lexize('english_stem', 'stars');
ts_lexize
-----
{star}

SELECT ts_lexize('english_stem', 'a');
ts_lexize
-----
{}
```

#### NOTICE

The `ts_lexize` function expects a single `token`, not text.

## 9.9 Limitations

The current limitations of GaussDB(DWS)'s full text search are:

- The length of each lexeme must be less than 2 KB.
- The length of a `tsvector` (lexemes + positions) must be less than 1 megabyte.
- Position values in `tsvector` must be greater than 0 and no more than 16383.
- No more than 256 positions per lexeme. Excessive positions, if any, will be discarded.
- The number of nodes (lexemes + operators) in a `tsquery` must be less than 32768.

---

# 10 System Operation

---

GaussDB(DWS) runs SQL statements to perform different system operations, such as setting variables, displaying the execution plan, and collecting garbage data.

## Setting Variables

For details about how to set various parameters for a session or transaction, see [SET](#).

## Displaying the Execution Plan

For details about how to display the execution plan that GaussDB(DWS) makes for SQL statements, see [EXPLAIN](#).

## Specifying a Checkpoint in Transaction Logs

By default, WALs periodically specify checkpoints in a transaction log. **CHECKPOINT** forces an immediate checkpoint when the related command is issued, without waiting for a regular checkpoint scheduled by the system. For details, see [CHECKPOINT](#).

## Collecting Unnecessary Data

For details about how to collect garbage data and analyze a database as required, For details, see [VACUUM](#).

## Collecting statistics

For details about how to collect statistics on tables in databases, For details, see [ANALYZE | ANALYSE](#).

## Setting the Constraint Check Mode for the Current Transaction

For details about how to set the constraint check mode for the current transaction, For details, see [SET CONSTRAINTS](#).

# 11 Controlling Transactions

---

A transaction is a user-defined sequence of database operations, which form an integral unit of work.

## Starting transactions

GaussDB(DWS) starts a transaction using **START TRANSACTION** and **BEGIN**. For details, see [START TRANSACTION](#) and [BEGIN](#).

## Setting transactions

GaussDB(DWS) sets a transaction using **SET TRANSACTION** or **SET LOCAL TRANSACTION**. For details, see [SET TRANSACTION](#).

## Submitting a transaction

GaussDB(DWS) commits all operations of a transaction using **COMMIT** or **END**. For details, see [COMMIT | END](#).

## Rolling back transactions

If a fault occurs during a transaction and the transaction cannot proceed, the system performs rollback to cancel all the completed database operations related to the transaction. For details, see [ROLLBACK](#).

### NOTE

If an execution request (not in a transaction block) received in the database contains multiple statements, the statements will be packed into a transaction. If one of the statements fails, the entire request will be rolled back.

# 12 DDL Syntax

---

## 12.1 DDL Syntax Overview

Data definition language (DDL) is used to define or modify an object in a database, such as a table, index, or view.

 **NOTE**

GaussDB(DWS) does not support DDL if its CN is unavailable. For example, if a CN in the cluster is faulty, creating a database or table will fail.

### Defining a Database

A database is the warehouse for organizing, storing, and managing data. Defining a database includes: creating a database, altering the database attributes, and dropping the database. The following table lists the related SQL statements.

**Table 12-1** SQL statements for defining a database

| Function                  | SQL Statement          |
|---------------------------|------------------------|
| Create a database         | <b>CREATE DATABASE</b> |
| Alter database attributes | <b>ALTER DATABASE</b>  |
| Delete a database         | <b>DROP DATABASE</b>   |

### Defining a Schema

A schema is the set of a group of database objects and is used to control the access to the database objects. The following table lists the related SQL statements.



**Table 12-2** SQL statements for defining a schema

| Function                | SQL Statement        |
|-------------------------|----------------------|
| Create a schema         | <b>CREATE SCHEMA</b> |
| Alter schema attributes | <b>ALTER SCHEMA</b>  |
| Delete a schema         | <b>DROP SCHEMA</b>   |

## Defining a Table

A table is a special data structure in a database and is used to store data objects and the relationship between data objects. The following table lists the related SQL statements.

**Table 12-3** SQL statements for defining a table

| Function               | SQL Statement       |
|------------------------|---------------------|
| Create a table         | <b>CREATE TABLE</b> |
| Alter table attributes | <b>ALTER TABLE</b>  |
| Delete a table         | <b>DROP TABLE</b>   |

## Defining a Partitioned Table

A partitioned table is a special data structure in a database and is used to store data objects and the relationship between data objects. The following table lists the related SQL statements.

**Table 12-4** SQL statements for defining a partitioned table

| Function                           | SQL Statement                 |
|------------------------------------|-------------------------------|
| Create a partitioned table         | <b>CREATE TABLE PARTITION</b> |
| Create a partition                 | <b>ALTER TABLE PARTITION</b>  |
| Alter partitioned table attributes | <b>ALTER TABLE PARTITION</b>  |
| Delete a partition                 | <b>ALTER TABLE PARTITION</b>  |
| Delete a partitioned table         | <b>DROP TABLE</b>             |

## Defining an Index

An index indicates the sequence of values in one or more columns in the database table. The database index is a data structure that improves the speed of data

access to specific information in a database table. The following table lists the related SQL statements.

**Table 12-5** SQL statements for defining an index

| Function               | SQL Statement       |
|------------------------|---------------------|
| Create an index        | <b>CREATE INDEX</b> |
| Alter index attributes | <b>ALTER INDEX</b>  |
| Delete an index        | <b>DROP INDEX</b>   |
| Rebuild an index       | <b>REINDEX</b>      |

## Defining a Role

A role is used to manage rights. For database security, all management and operation rights can be assigned to different roles. The following table lists the related SQL statements.

**Table 12-6** SQL statements for defining a role

| Function              | SQL Statement      |
|-----------------------|--------------------|
| Create a role         | <b>CREATE ROLE</b> |
| Alter role attributes | <b>ALTER ROLE</b>  |
| Delete a role         | <b>DROP ROLE</b>   |

## Defining a User

A user is used to log in to a database. Different rights can be assigned to users for managing data accesses and operations of users. The following table lists the related SQL statements.

**Table 12-7** SQL statements for defining a user

| Function              | SQL Statement      |
|-----------------------|--------------------|
| Create a user         | <b>CREATE USER</b> |
| Alter user attributes | <b>ALTER USER</b>  |
| Delete a user         | <b>DROP USER</b>   |

## Defining a Stored Procedure

A stored procedure is a set of SQL statements for achieving specific functions and is stored in the database after compiling. Users can specify a name and provide

parameters (if necessary) to execute the stored procedure. The following table lists the related SQL statements.

**Table 12-8** SQL statements for defining a stored procedure

| Function                  | SQL Statement           |
|---------------------------|-------------------------|
| Create a stored procedure | <b>CREATE PROCEDURE</b> |
| Delete a stored procedure | <b>DROP PROCEDURE</b>   |

## Define a Function

In GaussDB(DWS), a function is similar to a stored procedure, which is a set of SQL statements. The function and stored procedure are used the same. The following table lists the related SQL statements.

**Table 12-9** SQL statements for defining a function

| Function                  | SQL Statement          |
|---------------------------|------------------------|
| Create a function         | <b>CREATE FUNCTION</b> |
| Alter function attributes | <b>ALTER FUNCTION</b>  |
| Delete a function         | <b>DROP FUNCTION</b>   |

## Defining a View

A view is a virtual table exported from one or several basic tables. The view is used to control data accesses for users. The following table lists the related SQL statements.

**Table 12-10** SQL statements for defining a view

| Function      | SQL Statement      |
|---------------|--------------------|
| Create a view | <b>CREATE VIEW</b> |
| Delete a view | <b>DROP VIEW</b>   |

## Defining a Cursor

To process SQL statements, the stored procedure process assigns a memory segment to store context association. Cursors are handles or pointers to context regions. With a cursor, the stored procedure can control alterations in context areas.

**Table 12-11** SQL statements for defining a cursor

| Function                   | SQL Statement |
|----------------------------|---------------|
| Create a cursor            | <b>CURSOR</b> |
| Move a cursor              | <b>MOVE</b>   |
| Extract data from a cursor | <b>FETCH</b>  |
| Close a cursor             | <b>CLOSE</b>  |

## Altering or Ending a Session

A session is a connection established between the user and the database. The following table lists the related SQL statements.

**Table 12-12** SQL statements related to sessions

| Function        | SQL Statement                    |
|-----------------|----------------------------------|
| Alter a session | <b>ALTER SESSION</b>             |
| End a session   | <b>ALTER SYSTEM KILL SESSION</b> |

## Defining a Resource Pool

A resource pool is a system catalog used by the resource load management module to specify attributes related to resource management, such as Cgroups. The following table lists the related SQL statements.

**Table 12-13** SQL statements for defining a resource pool

| Function                   | SQL Statement               |
|----------------------------|-----------------------------|
| Create a resource pool     | <b>CREATE RESOURCE POOL</b> |
| Change resource attributes | <b>ALTER RESOURCE POOL</b>  |
| Delete a resource pool     | <b>DROP RESOURCE POOL</b>   |

## Defining Synonyms

A synonym is a special database object compatible with Oracle. It is used to store the mapping between a database object and another. Currently, only synonyms can be used to associate the following database objects: tables, views, functions, and stored procedures. The following table lists the related SQL statements.

**Table 12-14** SQL statements for defining a resource pool

| Function            | SQL Statement                             |
|---------------------|-------------------------------------------|
| Creating a synonym  | <a href="#">4.18.14.66-CREATE SYNONYM</a> |
| Modifying a synonym | <a href="#">4.18.14.23-ALTER SYNONYM</a>  |
| Deleting a synonym  | <a href="#">4.18.14.101-DROP SYNONYM</a>  |

## 12.2 ALTER DATABASE

### Function

**ALTER DATABASE** modifies a database, including its name, owner, object isolation, and connection limitation.

### Precautions

- Only the owner of a database or a system administrator has the permission to run the **ALTER DATABASE** statement. Users other than system administrators may have the following permission constraints depending on the attributes to be modified:
  - To modify the database name, you must have the CREATEDB permission.
  - To modify a database owner, you must be a database owner and a member of the new owner, and have the CREATEDB permission.
  - To change the default tablespace, you must be a database owner or a system administrator, and must have the CREATE permission on the new tablespace. This statement physically migrates tables and indexes in a default tablespace to a new tablespace. Note that tables and indexes outside the default tablespace are not affected.
  - Only a database owner or a system administrator can modify GUC parameters for the database.
  - Only database owners and system administrators can modify the object isolation attribute of a database.
- You are not allowed to rename a database in use. To rename it, connect to another database.

### Syntax

- Modify the maximum number of connections of the database.  

```
ALTER DATABASE database_name
  [ [ WITH ] CONNECTION LIMIT connlimit ];
```
- Rename the database.  

```
ALTER DATABASE database_name
  RENAME TO new_name;
```
- Change the database owner.  

```
ALTER DATABASE database_name
  OWNER TO new_owner;
```
- Modify the session parameter value of the database.

```
ALTER DATABASE database_name  
SET configuration_parameter { { TO | = } { value | DEFAULT } | FROM CURRENT };
```

- Reset the database configuration parameter.

```
ALTER DATABASE database_name RESET  
{ configuration_parameter | ALL };
```

- Modify the object isolation attribute of a database.

```
ALTER DATABASE database_name [ WITH ] { ENABLE | DISABLE } PRIVATE OBJECT;
```

#### NOTE

- To modify the object isolation attribute of a database, the database must be connected. Otherwise, the modification will fail.
- For a new database, the object isolation attribute is disabled by default. After this attribute is enabled, common users can view only the objects (such as tables, functions, views, and columns) that they have the permission to access. This attribute does not take effect for administrators. After this attribute is enabled, administrators can still view all database objects.

## Parameter Description

- **database\_name**  
Specifies the name of the database whose attributes are to be modified.  
Value range: a string compliant with the identifier naming rules
- **connlimit**  
Specifies the maximum number of concurrent connections that can be made to this database (excluding administrators' connections).  
Value range: The value must be an integer, preferably between **1** and **50**. The default value **-1** indicates no restrictions.
- **new\_name**  
Specifies the new name of a database.  
Value range: a string compliant with the identifier naming rules
- **new\_owner**  
Specifies the new owner of a database.  
Value range: a string indicating a valid user name
- **configuration\_parameter**  
**value**  
Sets a specified database session parameter. If the value is **DEFAULT** or **RESET**, the default setting is used in the new session. **OFF** closes the setting.  
Value range: A string. It can be set to:
  - DEFAULT
  - OFF
  - RESET
- **FROM CURRENT**  
Sets the value based on the database connected to the current session.
- **RESET configuration\_parameter**  
Resets the specified database session parameter.
- **RESET ALL**  
Resets all database session parameters.

 NOTE

- Modifies the default tablespace of a database by moving all the tables or indexes from the old tablespace to the new one. This operation does not affect the tables or indexes in other non-default tablespaces.
- The modified database session parameter values will take effect in the next session.

## Examples

See the [Examples](#) in **CREATE DATABASE**.

## Helpful Links

[CREATE DATABASE, DROP DATABASE](#)

# 12.3 ALTER FOREIGN TABLE (For GDS)

## Function

**ALTER FOREIGN TABLE** modifies a foreign table.

## Precautions

None

## Syntax

- Set the attributes of a foreign table.  

```
ALTER FOREIGN TABLE [ IF EXISTS ] table_name  
  OPTIONS ( {[ ADD | SET | DROP ] option ['value']}, ... );
```
- Set a new owner.  

```
ALTER FOREIGN TABLE [ IF EXISTS ] tablename  
  OWNER TO new_owner;
```

## Parameter Description

- **table\_name**  
Specifies the name of an existing foreign table to be modified.  
Value range: an existing foreign table name.
- **option**  
Specifies the name of the option to be modified.  
Value range: see [Parameter Description](#) in **CREATE FOREIGN TABLE**.
- **value**  
Specifies the new value of **option**.

## Examples

```
-- Create a foreign table:  
CREATE FOREIGN TABLE tpods.customer_ft  
(  
  c_customer_sk      integer      ,  
  c_customer_id     char(16)    ,
```

```
c_current_demo_sk      integer      ,
c_current_hdemo_sk     integer      ,
c_current_addr_sk      integer      ,
c_first_shipto_date_sk integer      ,
c_first_sales_date_sk  integer      ,
c_salutation           char(10)       ,
c_first_name           char(20)       ,
c_last_name            char(30)       ,
c_preferred_cust_flag  char(1)        ,
c_birth_day            integer      ,
c_birth_month          integer      ,
c_birth_year           integer      ,
c_birth_country        varchar(20)    ,
c_login                char(13)       ,
c_email_address        char(50)       ,
c_last_review_date    char(10)
)
SERVER gsmpp_server
OPTIONS
(
  location 'gsfs://10.185.179.143:5000/customer1*.dat',
  FORMAT 'TEXT' ,
  DELIMITER '|',
  encoding 'utf8',
  mode 'Normal')
READ ONLY;

-- Modify foreign table attributes and delete the mode option:
ALTER FOREIGN TABLE tpods.customer_ft options(drop mode);

-- Delete the foreign table:
DROP FOREIGN TABLE tpods.customer_ft;
```

## Helpful Links

[CREATE FOREIGN TABLE \(for GDS Import and Export\)](#), [DROP FOREIGN TABLE](#)

# 12.4 ALTER FOREIGN TABLE (For HDFS or OBS)

## Function

**ALTER FOREIGN TABLE For HDFS** modifies an HDFS or OBS foreign table.

## Precautions

None.

## Syntax

- Set a foreign table's attributes.  
ALTER FOREIGN TABLE [ IF EXISTS ] table\_name  
OPTIONS ( {[ ADD | SET | DROP ] option ['value']} [, ... ] );
- Set the owner of the foreign table.  
ALTER FOREIGN TABLE [ IF EXISTS ] tablename  
OWNER TO new\_owner;
- Update a foreign table column.  
ALTER FOREIGN TABLE [ IF EXISTS ] table\_name  
MODIFY ( { column\_name data\_type | column\_name [ CONSTRAINT constraint\_name ] NOT NULL  
[ ENABLE ] | column\_name [ CONSTRAINT constraint\_name ] NULL } [, ... ] );
- Modify the column of the foreign table.



```
ALTER FOREIGN TABLE [ IF EXISTS ] tablename
    action [, ... ];
```

The **action** syntax is as follows:

```
ALTER [ COLUMN ] column_name [ SET DATA ] TYPE data_type
| ALTER [ COLUMN ] column_name { SET | DROP } NOT NULL
| ALTER [ COLUMN ] column_name SET STATISTICS [PERCENT] integer
| ALTER [ COLUMN ] column_name OPTIONS ( {[ ADD | SET | DROP ] option ['value'] } [, ... ] )
| MODIFY column_name data_type
| MODIFY column_name [ CONSTRAINT constraint_name ] NOT NULL [ ENABLE ]
| MODIFY column_name [ CONSTRAINT constraint_name ] NULL
```

For details, see [ALTER TABLE](#).

- Add a foreign table informational constraint.

```
ALTER FOREIGN TABLE [ IF EXISTS ] tablename
    ADD [ CONSTRAINT constraint_name ]
    { PRIMARY KEY | UNIQUE } ( column_name )
    [ NOT ENFORCED [ ENABLE QUERY OPTIMIZATION | DISABLE QUERY OPTIMIZATION ] |
    ENFORCED ];
```

For parameters about adding an informational constraint to a foreign table, see [Parameter Description](#) in CREATE FOREIGN TABLE (For HDFS).

- Remove a foreign table informational constraint.

```
ALTER FOREIGN TABLE [ IF EXISTS ] tablename
    DROP CONSTRAINT constraint_name ;
```

## Parameter Description

- **IF EXISTS**

Sends a notification instead of an error if no tables have identical names. The notification prompts that the table you are querying does not exist.

- **tablename**

Specifies the name of an existing foreign table to be modified.

Value range: an existing foreign table name

- **new\_owner**

Specifies the new owner of the foreign table.

Value range: A string indicating a valid user name.

- **data\_type**

Specifies the new type for an existing column.

Value range: A string compliant with the identifier naming rules.

- **constraint\_name**

Specifies the name of a constraint to add or delete.

- **column\_name**

Specifies the name of an existing column.

Value range: a string. It must comply with the naming convention.

For details on how to modify other parameters in the foreign table, such as **IF EXISTS**, see [Parameter Description](#) in [ALTER TABLE](#).

## Examples

See [Example](#) in CREATE FOREIGN TABLE (for HDFS).

## Helpful Links

[CREATE FOREIGN TABLE \(SQL on Hadoop or OBS\), DROP FOREIGN TABLE](#)

# 12.5 ALTER FUNCTION

## Function

**ALTER FUNCTION** modifies the attributes of a customized function.

## Precautions

Only the owner of a function or a system administrator can run this statement. If a function involves operations on temporary tables, the **ALTER FUNCTION** cannot be used.

## Syntax

- Modify the additional parameter of the customized function.  

```
ALTER FUNCTION function_name ( [ { [ argmode ] [ argname ] argtype } [, ...] ] )  
    action [ ... ] [ RESTRICT ];
```

The syntax of the **action** clause is as follows:

```
{ CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT }  
{ IMMUTABLE | STABLE | VOLATILE }  
{ SHIPPABLE | NOT SHIPPABLE }  
{ NOT FENCED | FENCED }  
[ NOT ] LEAKPROOF  
{ [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER }  
AUTHID { DEFINER | CURRENT_USER }  
COST execution_cost  
ROWS result_rows  
SET configuration_parameter { { TO | = } { value | DEFAULT } FROM CURRENT }  
RESET { configuration_parameter | ALL }
```

- Modify the name of the customized function.  

```
ALTER FUNCTION funname ( [ { [ argmode ] [ argname ] argtype } [, ...] ] )  
    RENAME TO new_name;
```
- Modify the owner of the customized function.  

```
ALTER FUNCTION funname ( [ { [ argmode ] [ argname ] argtype } [, ...] ] )  
    OWNER TO new_owner;
```
- Modify the schema of the customized function.  

```
ALTER FUNCTION funname ( [ { [ argmode ] [ argname ] argtype } [, ...] ] )  
    SET SCHEMA new_schema;
```

## Parameter Description

- **function\_name**  
Specifies the function name to be modified.  
Value range: An existing function name.
- **argmode**  
Specifies whether a parameter is an input or output parameter.  
Value range: **IN**, **OUT**, **IN OUT**
- **argname**  
Indicates the parameter name.

Value range: A string. It must comply with the naming convention.

- **argtype**  
Specifies the parameter type.  
Value range: A valid type. For details, see [Data Types](#).
- **CALLED ON NULL INPUT**  
Declares that some parameters of the function can be invoked in normal mode if the parameter values are **NULL**. By default, the usage is the same as specifying the parameters.
- **RETURNS NULL ON NULL INPUT**  
**STRICT**  
Indicates that the function always returns **NULL** whenever any of its arguments are **NULL**. If this parameter is specified, the function is not executed when there are null arguments; instead a null result is assumed automatically.  
The usage of **RETURNS NULL ON NULL INPUT** is the same as that of **STRICT**.
- **IMMUTABLE**  
Indicates that the function always returns the same result if the parameter values are the same.
- **STABLE**  
Indicates that the function cannot modify the database, and that within a single table scan it will consistently return the same result for the same parameter values, but that its result varies by SQL statements.
- **VOLATILE**  
Indicates that the function value can change in one table scanning and no optimization is performed.
- **SHIPPABLE**
- **NOT SHIPPABLE**  
Indicates whether the function can be pushed down to DN for execution.  
Functions of the **IMMUTABLE** type can always be pushed down to the DN.  
Functions of the **STABLE** or **VOLATILE** type can be pushed down to DN only if their attribute is **SHIPPABLE**.
- **LEAKPROOF**  
Indicates that the function has no side effect and specifies that the parameter includes only the returned value. **LEAKPROOF** can be set only by a system administrator.
- **EXTERNAL**  
(Optional) The objective is to be compatible with SQL. This feature applies to all functions, including external functions.
- **SECURITY INVOKER**  
**AUTHID CURREN\_USER**  
Declares that the function will be executed according to the permission of the user that invokes it. By default, the usage is the same as specifying the parameters.  
**SECURITY INVOKER** and **AUTHID CURREN\_USER** have the same functions.

- **SECURITY DEFINER**  
**AUTHID DEFINER**  
Specifies that the function is to be executed with the permissions of the user that created it.  
The usage of **AUTHID DEFINER** is the same as that of **SECURITY DEFINER**.
- **COST execution\_cost**  
A positive number giving the estimated execution cost for the function.  
The unit of **execution\_cost** is `cpu_operator_cost`.  
Value range: A positive number.
- **ROWS result\_rows**  
Estimates the number of rows returned by the function. This is only allowed when the function is declared to return a set.  
Value range: A positive number. The default is 1000 rows.
- **configuration\_parameter**
  - **value**  
Sets a specified database session parameter to a specified value. If the value is **DEFAULT** or **RESET**, the default setting is used in the new session. **OFF** closes the setting.  
Value range: a string
    - **DEFAULT**
    - **OFF**
    - **RESET**Specifies the default value.
  - **from current**  
Uses the value of **configuration\_parameter** of the current session.
- **new\_name**  
Specifies the new name of a function. To change a function's schema, you must also have the CREATE permission on the new schema.  
Value range: A string. It must comply with the naming convention.
- **new\_owner**  
Specifies the new owner of a function. To alter the owner, the new owner must also be a direct or indirect member of the new owning role, and that role must have CREATE permission on the function's schema.  
Value range: Existing user roles.
- **new\_schema**  
Specifies the new schema of a function.  
Value range: Existing schemas.

## Examples

See [Examples](#) in **CREATE FUNCTION**.

## Helpful Links

[CREATE FUNCTION, DROP FUNCTION](#)

# 12.6 ALTER GROUP

## Function

**ALTER GROUP** modifies the attributes of a user group.

## Precautions

**ALTER GROUP** is an alias for **ALTER ROLE**, and it is not a standard SQL command and not recommended. Users can use **ALTER ROLE** directly.

## Syntax

- Add users to a group.  

```
ALTER GROUP group_name  
  ADD USER user_name [, ... ];
```
- Remove users from a group.  

```
ALTER GROUP group_name  
  DROP USER user_name [, ... ];
```
- Change the name of the group.  

```
ALTER GROUP group_name  
  RENAME TO new_name;
```

## Parameter Description

See the [Parameter Description](#) in **ALTER ROLE**.

## Helpful Links

[CREATE GROUP, DROP GROUP, ALTER ROLE](#)

# 12.7 ALTER INDEX

## Function

**ALTER INDEX** modifies the definition of an existing index.

There are several sub-forms:

- IF EXISTS  
If the specified index does not exist, a notice instead of an error is sent.
- RENAME TO  
Changes only the name of the index. There is no effect on the stored data.
- SET TABLESPACE  
This option changes the index tablespace to the specified tablespace, and moves the index-related data files to the new tablespace.

- **SET ( { STORAGE\_PARAMETER = value } [, ...] )**  
Change one or more index-method-specific storage parameters. Note that the index contents will not be modified immediately by this command. You might need to rebuild the index with **REINDEX** to get the desired effects depending on parameters.
- **RESET ( { storage\_parameter } [, ...] )**  
Reset one or more index-method-specific storage parameters to the default value. Similar to the **SET** statement, **REINDEX** may be used to completely update the index.
- **[ MODIFY PARTITION index\_partition\_name ] UNUSABLE**  
Sets the index on a table or index partition to be unavailable.
- **REBUILD [ PARTITION index\_partition\_name ]**  
Recreates the index on a table or index partition.
- **RENAME PARTITION**  
Renames an index partition.
- **MOVE PARTITION**  
Modifies the tablespace of an index partition.

## Precautions

- Only the owner of an index or a system administrator can run this statement.
- Tablespaces of partitioned table indexes cannot be changed. However, tablespaces of partition indexes can be changed.

## Syntax

- **Rename a table index.**  

```
ALTER INDEX [ IF EXISTS ] index_name  
    RENAME TO new_name;
```
- **Modify the storage parameter of a table index.**  

```
ALTER INDEX [ IF EXISTS ] index_name  
    SET ( {storage_parameter = value} [, ... ] );
```
- **Reset the storage parameter of a table index.**  

```
ALTER INDEX [ IF EXISTS ] index_name  
    RESET ( storage_parameter [, ... ] );
```
- **Set a table index or an index partition to be unavailable.**  

```
ALTER INDEX [ IF EXISTS ] index_name  
    [ MODIFY PARTITION index_partition_name ] UNUSABLE;
```

### NOTE

The syntax cannot be used for column-store tables.

- **Rebuild a table index or index partition.**  

```
ALTER INDEX index_name  
    REBUILD [ PARTITION index_partition_name ];
```
- **Rename an index partition.**  

```
ALTER INDEX [ IF EXISTS ] index_name  
    RENAME PARTITION index_partition_name TO new_index_partition_name;
```

### NOTE

**PG\_OBJECT** does not support the record of the syntax when the last modification time of the index is recorded.

## Parameter Description

- **index\_name**  
Specifies the index name to be modified.
- **new\_name**  
Specifies the new name for the index.  
Value range: A string that must comply with the identifier naming rules.
- **storage\_parameter**  
Specifies the name of an index-method-specific parameter.
- **value**  
Specifies the new value for an index-method-specific storage parameter. This might be a number or a word depending on the parameter.
- **new\_index\_partition\_name**  
Specifies the new name of the index partition.
- **index\_partition\_name**  
Specifies the name of the index partition.

## Examples

See [Examples](#) in the CREATE INDEX.

## Helpful Links

[CREATE INDEX](#), [DROP INDEX](#), [REINDEX](#)

# 12.8 ALTER LARGE OBJECT

## Function

**ALTER LARGE OBJECT** changes the owner of a large object.

## Precautions

Only the administrator or the owner of the to-be-modified large object can run **ALTER LARGE OBJECT**.

## Syntax

```
ALTER LARGE OBJECT large_object_oid  
OWNER TO new_owner;
```

## Parameter Description

- **large\_object\_oid**  
Specifies the OID of the large object to be modified.  
Value range: An existing large object name.
- **OWNER TO new\_owner**  
Specifies the new owner of the large object.

Value range: An existing user name/role.

## Examples

None.

# 12.9 ALTER NODE

## Function

**ALTER NODE** modifies the definition of an existing node.

## Precautions

**ALTER NODE** is the internal interface of cluster management tool. You are not advised to use this interface, because doing so affects the cluster.

## Syntax

```
ALTER NODE nodename WITH
(
  [ TYPE = nodetype,]
  [ HOST = hostname,]
  [ PORT = portnum,]
  [ HOST1 = 'hostname',]
  [ PORT1 = portnum,]
  [ HOSTPRIMARY [ = boolean ],]
  [ PRIMARY [ = boolean ],]
  [ PREFERRED [ = boolean ],]
  [ SCTP_PORT = portnum,]
  [ CONTROL_PORT = portnum,]
  [ SCTP_PORT1 = portnum,]
  [ CONTROL_PORT1 = portnum, ]
  [ NODEIS_CENTRAL [ = boolean ]]
);
```

### NOTE

The port whose number is specified by **PORT** is used for internal communications between nodes. Unlike the port number which connect node by external client, it can be queried in the `pgxc_node` table.

## Parameter Description

See [Parameter description](#) in **CREATE NODE**.

## Helpful Links

[CREATE NODE, DROP NODE](#)

# 12.10 ALTER NODE GROUP

## Function

**ALTER NODE GROUP** modifies the information about a Node Group.



## Precautions

- Only a system administrator is allowed to modify Node Group information.
- Node Group modification operations (excluding SET DEFAULT) are internal and need to be performed in maintenance mode (by invoking **set xc\_maintenance\_mode=on;**).
- **ALTER NODE GROUP** can be used only within a database. To avoid data inconsistency in DBMS, do not manually run this SQL statement.

## Syntax

```
ALTER NODE GROUP groupname
| SET DEFAULT
| RENAME TO new_group_name
| SET VCGROUP RENAME TO new_group_name
| SET NOT VCGROUP
| SET TABLE GROUP new_group_name
| COPY BUCKETS FROM src_group_name
| ADD NODE ( nodename [, ... ] )
| DELETE NODE ( nodename [, ... ] )
| RESIZE TO dest_group_name
| SET VCGROUP WITH GROUP new_group_name
```

## Parameter Description

- **groupname**  
Specifies the Node Group to be renamed.  
Value range: a string compliant with the naming convention
- **SET DEFAULT**  
Sets **in\_redistribution** to 'y' for all Node Groups excluding the one specified by **groupname**. To be compatible with earlier versions, this syntax is retained and does not need to be executed in maintenance mode.
- **RENAME TO new\_group\_name**  
Renames the Node Group specified by **groupname** to *new\_group\_name*.
- **SET VCGROUP RENAME TO new\_group\_name**  
Converts the entire physical cluster into a logical cluster. After the conversion, **groupname** is the logical cluster name, and the original physical cluster is changed to **new\_group\_name**.
- **SET NOT VCGROUP**  
Converts all logical clusters to common Node Groups and changes **group\_kind** from 'v' to 'n' for all of them.
- **SET TABLE GROUP new\_group\_name**  
Changes all the **group\_names** in the **pgroup** columns of the **pgxc\_class** tables on all CNs to **new\_group\_name**.
- **COPY BUCKETS FROM src\_group\_name**  
Copies values in the **group\_members** and **group\_bucketcontent** columns from the Node Group specified by **src\_group\_name** to the Node Group specified by **groupname**.
- **ADD NODE ( nodename [, ... ] )**  
Adds nodes from the Node Group specified by **groupname**. After the statement execution, the new nodes are registered with the **pgxc\_node**

system catalog. This statement only modifies the system catalog and does not add nodes or redistribute data. Do not invoke this statement.

- **DELETE NODE ( nodename [, ... ] )**

Deletes nodes from the Node Group specified by **groupname**. The deleted nodes still exist in the **pgxc\_node** system catalog. This statement only modifies the system catalog and does not delete nodes or redistribute data. Do not invoke this statement.

- **RESIZE TO dest\_group\_name**

Specifies a resize flag for the cluster. Set the Node Group specified by **groupname** to the Node Group before data redistribution and set **is\_installation** of the Node Group to **FALSE**. Set **desst\_group\_name** to the Node Group after data redistribution and set **is\_installation** of the Node Group to **TRUE**.

- **SET VCGROUP WITH GROUP new\_group\_name**

Converts a physical cluster into a logical cluster. After the conversion, **groupname** is still the physical cluster, and **new\_group\_name** is the name of the logical cluster.

## 12.11 ALTER RESOURCE POOL

### Function

**ALTER RESOURCE POOL** changes the Cgroup of a resource pool.

### Precautions

Users having the ALTER permission can modify resource pools.

### Syntax

```
ALTER RESOURCE POOL pool_name  
WITH ({MEM_PERCENT= pct | CONTROL_GROUP="group_name" | ACTIVE_STATEMENTS=stmt |  
MAX_DOP = dop | MEMORY_LIMIT='memory_size' | io_limits=io_limits | io_priority='io_priority'}[, ... ]);
```

### Parameter Description

- **pool\_name**  
Specifies the name of the resource pool.  
The name of the resource pool is the name of an existing resource pool.  
Value range: a string. It must comply with the naming convention rule.
- **group\_name**  
Specifies the name of a Cgroup.

 **NOTE**

- You can use either double quotation marks ("" ) or single quotation mark (') in the syntax when setting the name of a Cgroup.
- The value of **group\_name** is case-sensitive.
- When **group\_name** is not specified, the default value "Medium" is used. It is the "Medium" Timeshare Cgroup of the DefaultClass Cgroup.
- If a database user specifies the Timeshare string (**Rush**, **High**, **Medium**, or **Low**) in the syntax, for example, if **control\_group** is set to **High**, the resource pool will be associated with the **High** Timeshare Cgroup under **DefaultClass**.

Value range: an existing control group.

- **stmt**

Specifies the maximum number of statements that can be concurrently executed in a resource pool.

Value range: Numeric data ranging from **-1** to **INT\_MAX**.

- **dop**

This is a reserved parameter.

Value range: Numeric data ranging from **-1** to **INT\_MAX**.

- **memory\_size**

Specifies the maximum storage for a resource pool.

Value range: a string, from **1KB** to **2047GB**.

- **mem\_percent**

Specifies the proportion of available resource pool memory to the total memory or group user memory.

The value of **mem\_percent** for a common user is an integer ranging from 0 to 100. The default value is **0**.

 **NOTE**

When both of **mem\_percent** and **memory\_limit** are specified, only **mem\_percent** takes effect.

- **io\_limits**

Specifies the upper limit of IOPS in a resource pool.

The IOPS is counted by ones for column storage and by 10 thousands for row storage.

- **io\_priority**

Specifies the I/O priority for jobs that consume many I/O resources. It takes effect when the I/O usage reaches 90%.

There are three priorities: **Low**, **Medium**, and **High**. If you do not want to control I/O resources, set this parameter to **None**, which is the default value.

 **NOTE**

The settings of **io\_limits** and **io\_priority** are valid only for complex jobs, such as batch import (using **INSERT INTO SELECT**, **COPY FROM**, or **CREATE TABLE AS**), complex queries involving over 500 MB data on each DN, and **VACUUM FULL**.

## Examples

The example assumes that the user has created the **class1** Cgroup and three Workload Cgroups under **class1**: **Low**, **wg1**, and **wg2**. For details, see Priority Control.

```
-- Create a resource pool:  
CREATE RESOURCE POOL pool1;  
  
-- Specify "High" Timeshare Workload under "DefaultClass" as the Cgroup for a resource pool:  
ALTER RESOURCE POOL pool1 WITH (CONTROL_GROUP="High");  
  
-- Remove resource pool 1:  
DROP RESOURCE POOL pool1;
```

## Helpful Links

[CREATE RESOURCE POOL, DROP RESOURCE POOL](#)

# 12.12 ALTER ROLE

## Function

**ALTER ROLE** changes the attributes of a role.

## Precautions

None

## Syntax

- Modifying the Rights of a Role

```
ALTER ROLE role_name [ [ WITH ] option [ ... ] ];
```

The **option** clause for granting rights is as follows:

```
{CREATEDB | NOCREATEDB}  
| {CREATEROLE | NOCREATEROLE}  
| {INHERIT | NOINHERIT}  
| {AUDITADMIN | NOAUDITADMIN}  
| {SYSADMIN | NOSYSADMIN}  
| {USEFT | NOUSEFT}  
| {LOGIN | NOLOGIN}  
| {REPLICATION | NOREPLICATION}  
| {INDEPENDENT | NOINDEPENDENT}  
| {VCADMIN | NOVCADMIN}  
| CONNECTION LIMIT connlimit  
| [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'  
| [ ENCRYPTED | UNENCRYPTED ] IDENTIFIED BY 'password' [ REPLACE 'old_password' ]  
| [ ENCRYPTED | UNENCRYPTED ] PASSWORD { 'password' | DISABLE }  
| [ ENCRYPTED | UNENCRYPTED ] IDENTIFIED BY { 'password' [ REPLACE 'old_password' ] |  
DISABLE }  
| VALID BEGIN 'timestamp'  
| VALID UNTIL 'timestamp'  
| RESOURCE POOL 'respool'  
| USER GROUP 'groupuser'  
| PERM SPACE 'spacelimit'  
| NODE GROUP logic_cluster_name  
| ACCOUNT { LOCK | UNLOCK }  
| PGUSER  
| LDAP
```

- Rename a role.  

```
ALTER ROLE role_name  
  RENAME TO new_name;
```
- Set parameters for a role.  

```
ALTER ROLE role_name [ IN DATABASE database_name ]  
  SET configuration_parameter {{ TO | = } { value | DEFAULT } | FROM CURRENT};
```
- Reset parameters for a role.  

```
ALTER ROLE role_name  
  [ IN DATABASE database_name ] RESET {configuration_parameter|ALL};
```

## Parameter Description

- **role\_name**  
Indicates a role name.  
Value range: an existing user name
- **IN DATABASE database\_name**  
Modifies the parameters of a role on a specified database.
- **SET configuration\_parameter**  
Sets parameters for a role. The session parameters modified using the **ALTER ROLE** command is only for a specific role and is valid in the next session triggered by the role.  
Valid value:  
Values of **configuration\_parameter** and **value** are listed in [SET](#).  
**DEFAULT** clears the value of **configuration\_parameter**. The value of the **configuration\_parameter** parameter will inherit the default value of the new session generated for the role.  
**FROM CURRENT** uses the value of **configuration\_parameter** of the current session.
- **RESET configuration\_parameter/ALL**  
The effect of clearing the **configuration\_parameter** value is the same as setting it to **DEFAULT**.  
Value range: **ALL** indicates that all parameter values are cleared.
- **ACCOUNT LOCK | ACCOUNT UNLOCK**
  - **ACCOUNT LOCK**: locks an account to forbid login to databases.
  - **ACCOUNT UNLOCK**: unlocks an account to allow login to databases.
- **PGUSER**  
**PGUSER** of a role cannot be modified in the current version.

For details about other parameters, see [Parameter Description](#) in **CREATE ROLE**.

## Examples

See [Examples](#) in **CREATE ROLE**.

## Helpful Links

[CREATE ROLE](#), [DROP ROLE](#), [SET](#)

## 12.13 ALTER ROW LEVEL SECURITY POLICY

### Function

**ALTER ROW LEVEL SECURITY POLICY** modifies an existing row-level access control policy, including the policy name and the users and expressions affected by the policy.

### Precautions

Only the table owner or administrators can perform this operation.

### Syntax

```
ALTER [ ROW LEVEL SECURITY ] POLICY [ IF EXISTS ] policy_name ON table_name RENAME TO
new_policy_name

ALTER [ ROW LEVEL SECURITY ] POLICY policy_name ON table_name
[ TO { role_name | PUBLIC } [, ...] ]
[ USING ( using_expression ) ]
```

### Parameter Description

- *policy\_name*  
Specifies the name of a row-level access control policy to be modified.
- *table\_name*  
Specifies the name of a table to which a row-level access control policy is applied.
- *new\_policy\_name*  
Specifies the new name of a row-level access control policy.
- *role\_name*  
Specifies names of users affected by a row-level access control policy will be applied. **PUBLIC** indicates that the row-level access control policy will affect all users.
- *using\_expression*  
Specifies an expression defined for a row-level access control policy. The return value is of the boolean type.

### Examples

```
-- Create the data table all_data.
CREATE TABLE all_data(id int, role varchar(100), data varchar(100));

-- Create a row-level access control policy to specify that the current user can view only their own data.
CREATE ROW LEVEL SECURITY POLICY all_data_rls ON all_data USING(role = CURRENT_USER);
\d+ all_data
```

| Table "public.all_data" |                        |           |          |              |             |
|-------------------------|------------------------|-----------|----------|--------------|-------------|
| Column                  | Type                   | Modifiers | Storage  | Stats target | Description |
| id                      | integer                |           | plain    |              |             |
| role                    | character varying(100) |           | extended |              |             |
| data                    | character varying(100) |           | extended |              |             |

Row Level Security Policies:

```

POLICY "all_data_rls"
  USING (((role)::name = "current_user"()))
Has OIDs: no
Distribute By: HASH(id)
Location Nodes: ALL DATANODES
Options: orientation=row, compression=no

-- Change the name of the all_data_rls policy.
ALTER ROW LEVEL SECURITY POLICY all_data_rls ON all_data RENAME TO all_data_new_rls;

-- Change the users affected by the row-level access control policy.
ALTER ROW LEVEL SECURITY POLICY all_data_new_rls ON all_data TO alice, bob;
\d+ all_data

```

| Table "public.all_data" |                        |           |          |              |             |
|-------------------------|------------------------|-----------|----------|--------------|-------------|
| Column                  | Type                   | Modifiers | Storage  | Stats target | Description |
| id                      | integer                |           | plain    |              |             |
| role                    | character varying(100) |           | extended |              |             |
| data                    | character varying(100) |           | extended |              |             |

```

Row Level Security Policies:
  POLICY "all_data_new_rls"
    TO alice,bob
    USING (((role)::name = "current_user"()))
Has OIDs: no
Distribute By: HASH(id)
Location Nodes: ALL DATANODES
Options: orientation=row, compression=no, enable_rowsecurity=true

-- Modify the expression defined for the access control policy.
ALTER ROW LEVEL SECURITY POLICY all_data_new_rls ON all_data USING (id > 100 AND role =
current_user);
\d+ all_data

```

| Table "public.all_data" |                        |           |          |              |             |
|-------------------------|------------------------|-----------|----------|--------------|-------------|
| Column                  | Type                   | Modifiers | Storage  | Stats target | Description |
| id                      | integer                |           | plain    |              |             |
| role                    | character varying(100) |           | extended |              |             |
| data                    | character varying(100) |           | extended |              |             |

```

Row Level Security Policies:
  POLICY "all_data_new_rls"
    TO alice,bob
    USING (((id > 100) AND ((role)::name = "current_user"()))))
Has OIDs: no
Distribute By: HASH(id)
Location Nodes: ALL DATANODES
Options: orientation=row, compression=no, enable_rowsecurity=true

```

## Helpful Links

[CREATE ROW LEVEL SECURITY POLICY, DROP ROW LEVEL SECURITY POLICY](#)

## 12.14 ALTER SCHEMA

### Function

**ALTER SCHEMA** changes the attributes of a schema.

### Precautions

Only the owner of an index or a system administrator can run this statement.

## Syntax

- Rename a schema.  

```
ALTER SCHEMA schema_name  
  RENAME TO new_name;
```
- Changes the owner of a schema.  

```
ALTER SCHEMA schema_name  
  OWNER TO new_owner;
```

## Parameter Description

- **schema\_name**  
Indicates the name of the current schema.  
Value range: An existing schema name.
- **RENAME TO new\_name**  
Renames a schema.  
**new\_name**: new name of the schema  
Value range: A string. It must comply with the naming convention.
- **OWNER TO new\_owner**  
Changes the owner of a schema. To do this as a non-administrator, you must be a direct or indirect member of the new owning role, and that role must have CREATE permission in the database.  
**new\_owner**: new owner of a schema  
Value range: An existing user name/role.

## Examples

```
-- Create the ds schema:  
CREATE SCHEMA ds;  
  
-- Rename the ds schema to ds_new:  
ALTER SCHEMA ds RENAME TO ds_new;  
  
-- Create user jack:  
CREATE USER jack PASSWORD 'Bigdata123@';  
  
-- Change the owner of ds_new to jack:  
ALTER SCHEMA ds_new OWNER TO jack;  
  
-- Delete user jack and the ds_new schema:  
DROP SCHEMA ds_new;  
DROP USER jack;
```

## Helpful Links

[CREATE SCHEMA, DROP SCHEMA](#)

# 12.15 ALTER SEQUENCE

## Function

**ALTER SEQUENCE** modifies the parameters of an existing sequence.



## Precautions

- You must be the owner of the sequence to use **ALTER SEQUENCE**.
- In the current version, you can modify only the owner, home column, and the maximum value. To modify other parameters, delete the sequence and create it again. Then, use the `Setval` function to restore original parameter values.
- **ALTER SEQUENCE MAXVALUE** cannot be used in transactions, functions, and stored procedures.
- After the maximum value of a sequence is changed, the cache of the sequence in all sessions is cleared.
- **ALTER SEQUENCE** blocks the invocation of **nextval**, **setval**, **currval**, and **lastval**.

## Syntax

Change the maximum value or home column of the sequence.

```
ALTER SEQUENCE [ IF EXISTS ] name  
  [ MAXVALUE maxvalue | NO MAXVALUE | NOMAXVALUE ]  
  [ OWNED BY { table_name.column_name | NONE } ] ;
```

Change the owner of a sequence.

```
ALTER SEQUENCE [ IF EXISTS ] name OWNER TO new_owner;
```

## Parameter Description

- name  
Specifies the sequence name to be changed.
- IF EXISTS  
Sends a notification instead of an error when you are modifying a non-existing sequence.
- MAXVALUE maxvalue | NO MAXVALUE  
Maximum value of a sequence. If **NO MAXVALUE** is declared, the default value of the ascending sequence is  $2^{63}-1$ , and that of the descending sequence is **-1**. **NOMAXVALUE** is equivalent to **NO MAXVALUE**.
- OWNED BY  
Associates a sequence with a specified column included in a table. In this way, the sequence will be deleted when you delete its associated field or the table where the field belongs.  
  
If the sequence has been associated with another table before you use this parameter, the new association will overwrite the old one.  
  
The associated table and sequence must be owned by the same user and in the same schema.  
  
If **OWNED BY NONE** is used, existing associations will be deleted.
- new\_owner  
Specifies the user name of the new owner. To change the owner, you must also be a direct or indirect member of the new role, and this role must have CREATE permission on the sequence's schema.

## Examples

```
-- Create a sequence named serial that starts from 101 and increases in ascending order:  
CREATE SEQUENCE serial START 101;  
  
-- Change the maximum value of serial to 200:  
ALTER SEQUENCE serial MAXVALUE 200;  
  
-- Create a table, and specify default values for the sequence:  
CREATE TABLE T1(C1 bigint default nextval('serial'));  
  
-- Change the owning column of the serial sequence to T1.C1:  
ALTER SEQUENCE serial OWNED BY T1.C1;  
  
-- Delete the sequence.  
DROP SEQUENCE serial cascade;  
DROP TABLE T1;
```

## Helpful Links

[CREATE SEQUENCE, DROP SEQUENCE](#)

# 12.16 ALTER SERVER

## Function

**ALTER SERVER** adds, modifies, or deletes the parameters of an existing server. You can query existing servers from the **pg\_foreign\_server** system catalog.

## Precautions

Only the owner of a server or a system administrator can run this statement.

## Syntax

- Change the parameters for an external server.

```
ALTER SERVER server_name [ VERSION 'new_version' ]  
[ OPTIONS ( {[ ADD | SET | DROP ] option ['value']} [, ... ] ) ];
```

In **OPTIONS**, **ADD**, **SET**, and **DROP** are operations to be executed. If these operations are not specified, **ADD** operations will be performed by default. **option** and **value** are corresponding operation parameters.

Currently, only **SET** is supported on an HDFS server. **ADD** and **DROP** are not supported. The syntax for SET and DROP operations is retained for later use.

- Change the owner of an external server.

```
ALTER SERVER server_name  
OWNER TO new_owner;
```

- Change the name of an external server.


```
ALTER SERVER server_name  
RENAME TO new_name;
```

## Parameter Description

The parameters for modifying the server are as follows:

- **server\_name**  
Specifies the name of the server to be modified.
- **new\_version**  
Indicates the new version of the server.
- The server parameters in **OPTIONS** are as follows:
  - address  
Specifies the IP address and port number of the primary and standby nodes of the HDFS cluster.

 **NOTE**

- **address** is mandatory for HDFS servers. Therefore, **ADD** and **DROP** operations are not supported.
  - **address** only supports IPv4 addresses in dot-decimal notation, and an address string cannot contain spaces. Groups of addresses are separated by commas (,). An IP address and a port number are separated by a colon (:). You are advised to configure two IP address and port pairs in an HDFS cluster. One is used as the socket address of the primary HDFS NameNode and another is used as that of the secondary HDFS NameNode.
  - hdfsconfigpath  
Specifies the HDFS cluster configuration file.
-  **NOTE**
- If HDFS is in security mode, **hdfsconfigpath** is mandatory.
  - If you set **hdfsconfigpath**, you can only set one value for **path**.
  - region  
Indicates the IP address or domain name of the OBS server. This parameter is available only when **type** is **OBS**.

- **new\_owner**  
Indicates the new owner of the server. To change the owner, you must be the owner of the external server and a direct or indirect member of the new owner role, and must have the USAGE permission on the encapsulator of the external server.
- **new\_name**  
Indicates the new name of the server.

## Examples

```

Create the hdfs_server server, and hdfs_fdw is the built-in foreign data wrapper.
CREATE SERVER hdfs_server FOREIGN DATA WRAPPER HDFS_FDW OPTIONS (address
'10.10.0.100:25000,10.10.0.101:25000', hdfsconfigpath '/opt/hadoop_client/HDFS/hadoop/etc/
hadoop',type'HDFS');

SELECT * FROM pg_foreign_server WHERE srvname='hdfs_server';
  srvname | srvowner | srvfwd | srvtype | srvversion | srvacl |
sroptions
-----+-----+-----+-----+-----+-----+-----
-----+-----+-----+-----+-----+-----+-----
-----
 hdfs_server |      10 | 13332 |         |           | {"address=
10.10.0.100:25000,10.10.0.101:25000",hdfsconfigpath=/opt/hadoop_client/HDFS/hadoop/etc/hadoop}
(1 row)
Change the current name to the IP address of the HDFS server.

```

```
ALTER SERVER hdfs_server OPTIONS ( SET address '10.10.0.110:25000,10.10.0.120:25000');

SELECT * FROM pg_foreign_server WHERE srvname='hdfs_server';
   srvname | srvowner | srvfdw | srvtype | srvversion | srvacl |          srvoptions
-----+-----+-----+-----+-----+-----+-----
 hdfs_server |      10 | 13167 |         |           |        | {"address=10.10.0.110:25000,10.10.0.120:25000",hdfscfgpath=/opt/hadoop_client/HDFS/hadoop/etc/hadoop}
(1 row)
--Change the current name to hdfscfgpath of the HDFS server.
ALTER SERVER hdfs_server OPTIONS ( SET hdfscfgpath '/opt/bigdata/hadoop');

SELECT * FROM pg_foreign_server WHERE srvname='hdfs_server';
   srvname | srvowner | srvfdw | srvtype | srvversion | srvacl |          srvoptions
-----+-----+-----+-----+-----+-----+-----
 hdfs_server |      10 | 13332 |         |           |        | {"address=10.10.0.110:25000,10.10.0.120:25000",hdfscfgpath=/opt/bigdata/hadoop}
(1 row)
-- Delete hdfs_server.
DROP SERVER hdfs_server;
```

## Helpful Links

[CREATE SERVER DROP SERVER](#)

# 12.17 ALTER SESSION

## Function

**ALTER SESSION** defines or modifies the conditions or parameters that affect the current session. Modified session parameters are kept until the current session is disconnected.

## Precautions

- If the **START TRANSACTION** command is not executed before the **SET TRANSACTION** command, the transaction is ended instantly and the command does not take effect.
- You can use the transaction\_mode(s) method declared in the **START TRANSACTION** command to avoid using the **SET TRANSACTION** command.

## Syntax

- Set transaction parameters of a session.

```
ALTER SESSION SET [ SESSION CHARACTERISTICS AS ] TRANSACTION
  { ISOLATION LEVEL { READ COMMITTED | READ UNCOMMITTED } | { READ ONLY | READ
WRITE } } [, ...];
```
- Set other running parameters of a session.

```
ALTER SESSION SET
  {{config_parameter {{ TO | = } { value | DEFAULT }
  | FROM CURRENT }} | CURRENT_SCHEMA [ TO | = ] { schema | DEFAULT }
  | TIME_ZONE time_zone
  | SCHEMA schema
  | NAMES encoding_name
  | ROLE role_name PASSWORD 'password'
  | SESSION AUTHORIZATION { role_name PASSWORD 'password' | DEFAULT }
  | XML OPTION { DOCUMENT | CONTENT }
  };
```

## Parameter Description

To modify the description of parameters related to the session, see [Parameter Description](#) of the **SET** syntax.

## Examples

```
-- Create the ds schema:
CREATE SCHEMA ds;

-- Set the search path of the schema:
SET SEARCH_PATH TO ds, public;

-- Set the time/date type to the traditional postgres format (date before month):
SET DATESTYLE TO postgres, dmy;

-- Set the character code of the current session to UTF8:
ALTER SESSION SET NAMES 'UTF8';

-- Set the time zone to Berkeley of California:
SET TIME_ZONE 'PST8PDT';

-- Set the time zone to Italy:
SET TIME_ZONE 'Europe/Rome';

-- Set the current schema:
ALTER SESSION SET CURRENT_SCHEMA TO tpceds;

-- Set XML OPTION to DOCUMENT:
ALTER SESSION SET XML_OPTION DOCUMENT;

-- Create the role joe, and set the session role to omm:
CREATE ROLE joe WITH PASSWORD 'Bigdata123@';
ALTER SESSION SET SESSION AUTHORIZATION joe PASSWORD 'Bigdata123@';

-- Switch to the default user:
ALTER SESSION SET SESSION AUTHORIZATION default;

-- Delete the ds schema:
DROP SCHEMA ds;

-- Delete the role joe:
DROP ROLE joe;
```

## Helpful Links

[SET](#)

# 12.18 ALTER SYNONYM

## Function

**ALTER SYNONYM** is used to modify the attribute of a synonym.

## Precautions

- Only the synonym owner can be changed.
- Only the system administrator has the permission to modify the synonym owner information.
- The new owner must have the CREATE permission on the schema where the synonym resides.

## Syntax

```
ALTER SYNONYM synonym_name  
OWNER TO new_owner;
```

## Parameter Description

- **synonym**  
Name of a synonym to be modified (optionally with schema names)  
Value range: A string compliant with the identifier naming rules
- **new\_owner**  
New owner of a synonym object  
Value range: A string. It must be a valid username.

## Examples

```
-- Create synonym t1.  
CREATE OR REPLACE SYNONYM t1 FOR ot.t1;  
  
-- Create user u1.  
CREATE USER u1 PASSWORD 'user@111';  
  
-- Change the owner of the synonym t1 to u1.  
ALTER SYNONYM t1 OWNER TO u1;  
  
-- Delete synonym t1.  
DROP SYNONYM t1;  
  
-- Delete user u1.  
DROP USER u1;
```

## Helpful Links

[CREATE SYNONYM](#) and [DROP SYNONYM](#)

# 12.19 ALTER SYSTEM KILL SESSION

## Function

**ALTER SYSTEM KILL SESSION** ends a session.

## Precautions

None

## Syntax

```
ALTER SYSTEM KILL SESSION 'session_sid, serial' [ IMMEDIATE ];
```

## Parameter Description

- **session\_sid, serial**  
Specifies **SID** and **SERIAL** of a session (see examples for format).  
Value range: The **SIDs** and **SERIALS** of all sessions that can be queried from the system catalog **V\$SESSION**.

- **IMMEDIATE**

Indicates that a session will be ended instantly after the command is executed.

## Examples

```
-- Query session information:
SELECT sid,serial#,username FROM V$SESSION;

  sid      | serial# | username
-----+-----+-----
140131075880720 | 0 | dbadmin
140131025549072 | 0 | dbadmin
140131073779472 | 0 | dbadmin
140131071678224 | 0 | dbadmin
140131125774096 | 0 |
140131127875344 | 0 |
140131113629456 | 0 |
140131094742800 | 0 |
(8 rows)

-- End the session whose SID is 140131075880720:
ALTER SYSTEM KILL SESSION '140131075880720,0' IMMEDIATE;
```

## 12.20 ALTER TABLE

### Function

**ALTER TABLE** is used to modify tables, including modifying table definitions, renaming tables, renaming specified columns in tables, renaming table constraints, setting table schemas, enabling or disabling row-level access control, and adding or updating multiple columns.

### Precautions

- You must own the table to use **ALTER TABLE**. A system administrator has the permission by default.
- The storage parameter **ORIENTATION** cannot be modified.
- Currently, **SET SCHEMA** can only set schemas to user schemas. It cannot set a schema to a system internal schema.
- The distribution column of a table cannot be modified.
- The column-store table only supports **PARTIAL CLUSTER KEY**.
- In a column-store table, you can perform **ADD COLUMN**, **ALTER TYPE**, **SET STATISTICS**, **DROP COLUMN** operations, and change table name and space. The types of new and modified columns should be the [Data Types](#) supported by column storage. The **USING** option of **ALTER TYPE** only supports constant expression and expression involved in the column.
- The column constraints supported by column-store table include **NULL**, **NOT NULL**, and **DEFAULT** constant values. Only **DEFAULT** value can be modified (**SET DEFAULT** and **DROP DEFAULT**). Currently, **NULL/NOT NULL** constraints cannot be modified.
- Auto-increment columns cannot be added, or a column in which the **DEFAULT** value contains the `nextval()` expression cannot be added either.

- Row-level access control cannot be enabled for HDFS tables, foreign tables, and temporary tables.
- If you delete the PRIMARY KEY constraint by specifying the constraint name, the NOT NULL constraint is not deleted. You can manually delete the NOT NULL constraint as needed.

## Syntax

- **ALTER TABLE** modifies the definition of a table.  
ALTER TABLE [ IF EXISTS ] { table\_name [\*] | ONLY table\_name | ONLY ( table\_name ) }  
action [, ... ];

There are several clauses of **action**:

```
column_clause
| ADD table_constraint [ NOT VALID ]
| ADD table_constraint_using_index
| VALIDATE CONSTRAINT constraint_name
| DROP CONSTRAINT [ IF EXISTS ] constraint_name [ RESTRICT | CASCADE ]
| CLUSTER ON index_name
| SET WITHOUT CLUSTER
| SET ( {storage_parameter = value} [, ... ] )
| RESET ( storage_parameter [, ... ] )
| OWNER TO new_owner
| SET TABLESPACE new_tablespace
| SET {COMPRESS|NOCOMPRESS}

| TO { GROUP groupname | NODE ( nodename [, ... ] ) }
| ADD NODE ( nodename [, ... ] )
| DELETE NODE ( nodename [, ... ] )
| DISABLE TRIGGER [ trigger_name | ALL | USER ]
| ENABLE TRIGGER [ trigger_name | ALL | USER ]
| ENABLE REPLICA TRIGGER trigger_name
| ENABLE ALWAYS TRIGGER trigger_name
| DISABLE ROW LEVEL SECURITY
| ENABLE ROW LEVEL SECURITY
| FORCE ROW LEVEL SECURITY
| NO FORCE ROW LEVEL SECURITY
```



 NOTE

- **ADD table\_constraint [ NOT VALID ]**  
Adds a new table constraint.
- **ADD table\_constraint\_using\_index**  
Adds primary key constraint or unique constraint based on the unique index.
- **VALIDATE CONSTRAINT constraint\_name**  
Validates a foreign key or check constraint that was previously created as **NOT VALID**, by scanning the table to ensure there are no rows for which the constraint is not satisfied. Nothing happens if the constraint is already marked valid.
- **DROP CONSTRAINT [ IF EXISTS ] constraint\_name [ RESTRICT | CASCADE ]**  
Drops a table constraint.
- **CLUSTER ON index\_name**  
Selects the default index for future **CLUSTER** operations. It does not actually re-cluster the table.
- **SET WITHOUT CLUSTER**  
Removes the most recently used **CLUSTER** index specification from the table. This operation affects future cluster operations that do not specify an index.
- **SET ( {storage\_parameter = value} [, ... ] )**  
Changes one or more storage parameters for the table.
- **RESET ( storage\_parameter [, ... ] )**  
Resets one or more storage parameters to their defaults. As with **SET**, a table rewrite might be needed to update the table entirely.
- **OWNER TO new\_owner**  
Changes the owner of the table, sequence, or view to the specified user.
- **SET {COMPRESS|NOCOMPRESS}**  
Sets the compression feature of a table. The table compression feature affects only the storage mode of data inserted in a batch subsequently and does not affect storage of existing data. Setting the table compression feature will result in the fact that there are both compressed and uncompressed data in the table.
- **TO { GROUP groupname | NODE ( nodename [, ... ] ) }**  
The syntax is only available in extended mode (when GUC parameter **support\_extended\_features** is **on**). Exercise caution when enabling the mode. It is used for tools like internal dilatation tools. Common users should not use the mode.
- **ADD NODE ( nodename [, ... ] )**  
It is only available for tools like internal dilatation. General users should not use the mode.
- **DELETE NODE ( nodename [, ... ] )**  
It is only available for internal scale-in tools. Common users should not use the syntax.
- **DISABLE TRIGGER [ trigger\_name | ALL | USER ]**  
Disables a single trigger specified by **trigger\_name**, disables all triggers, or disables only user triggers (excluding internally generated constraint triggers, for example, deferrable unique constraint triggers and exclusion constraints triggers).

 NOTE

Exercise caution when using this function because data integrity cannot be ensured as expected if the triggers are not executed.

- **| ENABLE TRIGGER [ trigger\_name | ALL | USER ]**

Enables a single trigger specified by **trigger\_name**, enables all triggers, or enables only user triggers.

- | **ENABLE REPLICA TRIGGER trigger\_name**

Determines that the trigger firing mechanism is affected by the configuration variable **session\_replication\_role**. When the replication role is **origin** (default value) or **local**, a simple trigger is fired.

When **ENABLE REPLICA** is configured for a trigger, it is fired only when the session is in **replica** mode.

- | **ENABLE ALWAYS TRIGGER trigger\_name**

Determines that all triggers are fired regardless of the current replication mode.

- | **DISABLE/ENABLE ROW LEVEL SECURITY**

Enables or disables row-level access control for a table.

If row-level access control is enabled for a data table but no row-level access control policy is defined, the row-level access to the data table is not affected. If row-level access control for a table is disabled, the row-level access to the table is not affected even if a row-level access control policy has been defined. For details, see [CREATE ROW LEVEL SECURITY POLICY](#).

- | **NO FORCE/FORCE ROW LEVEL SECURITY**

Forcibly enables or disables row-level access control for a table.

By default, the table owner is not affected by the row-level access control feature. However, if row-level access control is forcibly enabled, the table owner (excluding system administrators) will be affected. System administrators are not affected by any row-level access control policies.

- There are several clauses of **column\_clause**:

```
ADD [ COLUMN ] column_name data_type [ compress_mode ] [ COLLATE collation ]
[ column_constraint [ ... ] ]
| MODIFY column_name data_type
| MODIFY column_name [ CONSTRAINT constraint_name ] NOT NULL [ ENABLE ]
| MODIFY column_name [ CONSTRAINT constraint_name ] NULL
| DROP [ COLUMN ] [ IF EXISTS ] column_name [ RESTRICT | CASCADE ]
| ALTER [ COLUMN ] column_name [ SET DATA ] TYPE data_type [ COLLATE collation ] [ USING
expression ]
| ALTER [ COLUMN ] column_name { SET DEFAULT expression | DROP DEFAULT }
| ALTER [ COLUMN ] column_name { SET | DROP } NOT NULL
| ALTER [ COLUMN ] column_name SET STATISTICS [PERCENT] integer
ADD STATISTICS (( column_1_name, column_2_name [, ...] ))
DELETE STATISTICS (( column_1_name, column_2_name [, ...] ))
| ALTER [ COLUMN ] column_name SET ( {attribute_option = value} [, ...] )
| ALTER [ COLUMN ] column_name RESET ( attribute_option [, ...] )
| ALTER [ COLUMN ] column_name SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN }
```

 NOTE

- **ADD [ COLUMN ] column\_name data\_type [ compress\_mode ] [ COLLATE collation ] [ column\_constraint [ ... ] ]**  
Adds a column to a table. If a column is added with **ADD COLUMN**, all existing rows in the table are initialized with the column's default value (**NULL** if no **DEFAULT** clause is specified).
- **ADD ( { column\_name data\_type [ compress\_mode ] } [, ...] )**  
Adds columns in the table.
- **MODIFY column\_name data\_type**  
Modifies the data type of an existing column in the table.
- **MODIFY column\_name [ CONSTRAINT constraint\_name ] NOT NULL [ ENABLE ]**  
Adds the NOT NULL constraint to a certain column in the table.
- **MODIFY column\_name [ CONSTRAINT constraint\_name ] NULL**  
Deletes the NOT NULL constraint to a certain column in the table.
- **DROP [ COLUMN ] [ IF EXISTS ] column\_name [ RESTRICT | CASCADE ]**  
Drops a column from a table. Index and constraint related to the column are automatically dropped. If an object not belonging to the table depends on the column, **CASCADE** must be specified, such as foreign key reference and view.  
The **DROP COLUMN** form does not physically remove the column, but simply makes it invisible to SQL operations. Subsequent insert and update operations in the table will store a **NULL** value for the column. Therefore, column deletion takes a short period of time but does not immediately release the table space on the disks, because the space occupied by the deleted column is not reclaimed. The space will be reclaimed when **VACUUM** is executed.
- **ALTER [ COLUMN ] column\_name [ SET DATA ] TYPE data\_type [ COLLATE collation ] [ USING expression ]**  
Modifies the type of a column in a table. Indexes and simple table constraints on the column will automatically use the new data type by reparsing the originally supplied expression.  
**ALTER TYPE** requires an entire table be rewritten. This is an advantage sometimes, because it frees up unnecessary space from a table. For example, to reclaim the space occupied by a deleted column, the fastest method is to use the command.  

```
ALTER TABLE table ALTER COLUMN anycol TYPE anytype;
```

  
In this command, **anycol** indicates any column existing in the table and **anytype** indicates the type of the prototype of the column. **ALTER TYPE** does not change the table except that the table is forcibly rewritten. In this way, the data that is no longer used is deleted.
- **ALTER [ COLUMN ] column\_name { SET DEFAULT expression | DROP DEFAULT }**  
Sets or removes the default value for a column. The default values only apply to subsequent **INSERT** commands; they do not cause rows already in the table to change. Defaults can also be created for views, in which case they are inserted into **INSERT** statements on the view before the view's **ON INSERT** rule is applied.
- **ALTER [ COLUMN ] column\_name { SET | DROP } NOT NULL**  
Changes whether a column is marked to allow **NULL** values or to reject **NULL** values. You can only use **SET NOT NULL** when the column contains no **NULL** values.
- **ALTER [ COLUMN ] column\_name SET STATISTICS [PERCENT] integer**

Specifies the per-column statistics-gathering target for subsequent **ANALYZE** operations. The value ranges from **0** to **10000**. Set it to **-1** to revert to using the default system statistics target.

- **{ADD | DELETE} STATISTICS ((column\_1\_name, column\_2\_name [, ...]))**

Adds or deletes the declaration of collecting multi-column statistics to collect multi-column statistics as needed when **ANALYZE** is performed for a table or a database. The statistics about a maximum of 32 columns can be collected at a time. You are not allowed to add or delete the declaration for system tables or foreign tables

- **ALTER [ COLUMN ] column\_name SET ( {attribute\_option = value} [, ... ] )**

- **ALTER [ COLUMN ] column\_name RESET ( attribute\_option [, ... ] )**

Sets or resets per-attribute options.

Currently, the only defined per-attribute options are **n\_distinct** and **n\_distinct\_inherited**. **n\_distinct** affects statistics of table, while **n\_distinct\_inherited** affects the statistics of table and its subtables. Currently, only **SET/RESET n\_distinct** is supported, and **SET/RESET n\_distinct\_inherited** is forbidden.

- **ALTER [ COLUMN ] column\_name SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN }**

Sets the storage mode for a column. This clause specifies whether this column is held inline or in a secondary TOAST table, and whether the data should be compressed. This clause takes effect only for row-store tables and an error will be reported if it is executed for column-store tables. **SET STORAGE** does not change a table. It only specifies the recommended strategy for future table updates.

- **column\_constraint** is as follows:

```
[ CONSTRAINT constraint_name ]
{ NOT NULL |
  NULL |
  CHECK ( expression ) |
  DEFAULT default_expr |
  UNIQUE index_parameters |
  PRIMARY KEY index_parameters }
[ DEFERRABLE | NOT DEFERRABLE | INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

- **compress\_mode** of a column is as follows:

```
[ DELTA | PREFIX | DICTIONARY | NUMSTR | NOCOMPRESS ]
```

- **table\_constraint\_using\_index** used to add the primary key constraint or unique constraint based on the unique index is as follows:

```
[ CONSTRAINT constraint_name ]
{ UNIQUE | PRIMARY KEY } USING INDEX index_name
[ DEFERRABLE | NOT DEFERRABLE | INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

- **table\_constraint** is as follows:

```
[ CONSTRAINT constraint_name ]
{ CHECK ( expression ) |
  UNIQUE ( column_name [, ... ] ) index_parameters |
  PRIMARY KEY ( column_name [, ... ] ) index_parameters }
[ DEFERRABLE | NOT DEFERRABLE | INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

- **index\_parameters** is as follows:

```
[ WITH ( {storage_parameter = value} [, ... ] ) ]
[ USING INDEX TABLESPACE tablespace_name ]
```

- Rename the table. The renaming does not affect stored data.

```
ALTER TABLE [ IF EXISTS ] table_name
  RENAME TO new_table_name;
```

- Rename the specified column in the table.

```
ALTER TABLE [ IF EXISTS ] { table_name [*] | ONLY table_name | ONLY ( table_name ) }
  RENAME [ COLUMN ] column_name TO new_column_name;
```

- Rename the constraint of the table.  

```
ALTER TABLE [ IF EXISTS ] { table_name [*] | ONLY table_name | ONLY ( table_name ) }  
    RENAME CONSTRAINT constraint_name TO new_constraint_name;
```
- Set the schema of the table.  

```
ALTER TABLE [ IF EXISTS ] table_name  
    SET SCHEMA new_schema;
```

#### NOTE

- The schema setting moves the table into another schema. Associated indexes and constraints owned by table columns are migrated as well. Currently, the schema for sequences cannot be changed. If the table has sequences, delete the sequences, and create them again or delete the ownership between the table and sequences. In this way, the table schema can be changed.
  - To change the schema of a table, you must also have CREATE privilege on the new schema. To add the table as a new child of a parent table, you must own the parent table as well. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE permission on the table's schema. These restrictions enforce that altering the owner does not do anything you could not do by dropping and recreating the table. However, a system administrator can alter ownership of any table anyway.
  - All the actions except for **RENAME** and **SET SCHEMA** can be combined into a list of multiple alterations to apply in parallel. For example, it is possible to add several columns or alter the type of several columns in a single command. This is useful with large tables, since only one pass over the table need be made.
  - Adding a **CHECK** or **NOT NULL** constraint requires scanning the table to verify that existing rows meet the constraint.
  - Adding a column with a non-null default or changing the type of an existing column will require the entire table to be rewritten. Table rebuilding may take a significant amount of time for a large table; and will temporarily require as much as double the disk space.
- Add columns.  

```
ALTER TABLE [ IF EXISTS ] table_name  
    ADD ( { column_name data_type [ compress_mode ] [ COLLATE collation ] [ column_constraint  
[ ... ] } [, ...] );
```
  - Update columns.  

```
ALTER TABLE [ IF EXISTS ] table_name  
    MODIFY ( { column_name data_type | column_name [ CONSTRAINT constraint_name ] NOT NULL  
[ ENABLE ] | column_name [ CONSTRAINT constraint_name ] NULL } [, ...] );
```

## Parameter Description

- **IF EXISTS**  
Sends a notification instead of an error if no tables have identical names. The notification prompts that the table you are querying does not exist.
- **table\_name [\*] | ONLY table\_name | ONLY ( table\_name )**  
**table\_name** is the name of table that you need to modify.  
If **ONLY** is specified, only the table is modified. If **ONLY** is not specified, the table and all subtables will be modified. You can add the asterisk (\*) option following the table name to specify that all subtables are scanned, which is the default operation.
- **constraint\_name**  
Specifies the name of an existing constraint to drop.
- **index\_name**  
Specifies the name of this index.

- **storage\_parameter**  
Specifies the name of a storage parameter.
- **new\_owner**  
Specifies the name of the new table owner.
- **new\_tablespace**  
Specifies the new name of the tablespace to which the table belongs.
- **column\_name, column\_1\_name, column\_2\_name**  
Specifies the name of a new or an existing column.
- **data\_type**  
Specifies the type of a new column or a new type of an existing column.
- **compress\_mode**  
Specifies the compress options of the table, only available for row-store tables. The clause specifies the algorithm preferentially used by the column.
- **collation**  
Specifies the collation rule name of a column. The optional **COLLATE** clause specifies a collation for the new column; if omitted, the collation is the default for the new column.
- **USING expression**  
A **USING** clause specifies how to compute the new column value from the old; if omitted, the default conversion is an assignment cast from old data type to new. A **USING** clause must be provided if there is no implicit or assignment cast from the old to new type.

 **NOTE**

**USING** in **ALTER TYPE** can specify any expression involving the old values of the row; that is, it can refer to any columns other than the one being converted. This allows very general conversions to be done with the **ALTER TYPE** syntax. Because of this flexibility, the **USING** expression is not applied to the column's default value (if any); the result might not be a constant expression as required for a default. This means that when there is no implicit or assignment cast from old to new type, **ALTER TYPE** might fail to convert the default even though a **USING** clause is supplied. In such cases, drop the default with **DROP DEFAULT**, perform the **ALTER TYPE**, and then use **SET DEFAULT** to add a suitable new default. Similar considerations apply to indexes and constraints involving the column.

- **NOT NULL | NULL**  
Sets whether the column allows null values.
- **integer**  
Specifies the constant value of an integer with a sign. If **PERCENT** is used, the range of **integer** is from 0 to 100.
- **attribute\_option**  
Specifies an attribute option.
- **PLAIN | EXTERNAL | EXTENDED | MAIN**  
Specifies a column storage mode.
  - **PLAIN** must be used for fixed-length values (such as integers). It must be inline and uncompressed.
  - **MAIN** is for inline, compressible data.

- **EXTERNAL** is for external, uncompressed data. Use of **EXTERNAL** will make substring operations on **text** and **bytea** values run faster, at the penalty of increased storage space.
- **EXTENDED** is for external, compressed data. **EXTENDED** is the default for most data types that support non-**PLAIN** storage.
- **CHECK ( expression )**

New or updated rows must satisfy for an insert or update operation to succeed. Expressions evaluating to TRUE succeed. If any row of an insert or update operation produces a FALSE result, an error exception is raised and the insert or update does not alter the database.

A check constraint specified as a column constraint should reference only the column's values, while an expression appearing in a table constraint can reference multiple columns.

Currently, **CHECK** expression does not include subqueries and cannot use variables apart from the current column.
- **DEFAULT default\_expr**

Assigns a default data value for a column.

The data type of the default expression must match the data type of the column.

The default expression will be used in any insert operation that does not specify a value for the column. If there is no default value for a column, then the default value is **NULL**.
- **UNIQUE index\_parameters**

**UNIQUE ( column\_name [, ... ] ) index\_parameters**

The **UNIQUE** constraint specifies that a group of one or more columns of a table can contain only unique values.
- **PRIMARY KEY index\_parameters**

**PRIMARY KEY ( column\_name [, ... ] ) index\_parameters**

The primary key constraint specifies that a column or columns of a table can contain only unique (non-duplicate) and non-null values.
- **DEFERRABLE | NOT DEFERRABLE | INITIALLY DEFERRED | INITIALLY IMMEDIATE**

Sets whether the constraint can be deferrable.

  - **DEFERRABLE**: deferrable can be postponed until the end of the transaction using the **SET CONSTRAINTS** command.
  - **NOT DEFERRABLE**: checks immediately after the execution of each command.
  - **INITIALLY IMMEDIATE**: checks immediately after the execution of each statement.
  - **INITIALLY DEFERRED**: checks when the transaction ends.
- **WITH ( {storage\_parameter = value} [, ... ] )**

Specifies an optional storage parameter for a table or an index.
- **COMPRESS|NOCOMPRESS**
  - **NOCOMPRESS**: If the **NOCOMPRESS** keyword is specified, the existing compression feature of the table is not changed.

- **COMPRESS**: If the **COMPRESS** keyword is specified, the table compression feature is triggered if tuples are inserted in a batch.
- **new\_table\_name**  
Specifies the new table name.
- **new\_column\_name**  
Specifies the new name of a specific column in a table.
- **new\_constraint\_name**  
Specifies the new name of a table constraint.
- **new\_schema**  
Specifies the new schema name.
- **CASCADE**  
Automatically drops objects that depend on the dropped column or constraint (for example, views referencing the column).
- **RESTRICT**  
Refuses to drop the column or constraint if there are any dependent objects. This is the default behavior.
- **schema\_name**  
Specifies the schema name of a table.

## Examples

See [Examples](#).

## Helpful Links

[CREATE TABLE, DROP TABLE](#)

# 12.21 ALTER TABLE PARTITION

## Function

**ALTER TABLE PARTITION** modifies table partition, including add, delete, split, merge partitions, and modify partition attributes.

## Precautions

- The name of the added partition must be different from names of existing partitions in the partition table.
- The partition key of the added partition must be the same type as that of the partition table. The key value of the added partition must exceed the upper limit of the last partition range.
- If the number of partitions in the target partition table has reached the maximum (**32767**), partitions cannot be added.
- If a partition table has only one partition, the partition cannot be deleted.
- Use **PARTITION FOR()** to choose partitions. The number of specified values in the brackets should be the same as the column number in customized partition, and they must be consistent.



- The **Value** partition table does not support the **Alter Partition** operation.

## Syntax

- Modify the syntax of the table partition.

```
ALTER TABLE [ IF EXISTS ] { table_name [*] | ONLY table_name | ONLY ( table_name ) }
action [, ... ];
```

**action** indicates the following clauses for maintaining partitions. For the partition continuity when multiple clauses are used for partition maintenance, GaussDB(DWS) does DROP PARTITION and then ADD PARTITION, and finally runs the rest clauses in sequence.

```
move_clause |
exchange_clause |
row_clause |
merge_clause |
modify_clause |
split_clause |
add_clause |
drop_clause
```

- The **move\_clause** syntax is used to move the partition to a new tablespace.

```
MOVE PARTITION { partition_name | FOR ( partition_value [, ...] ) } TABLESPACE tablespacename
```

- The **exchange\_clause** syntax is used to move the data from a general table to a specified partition.

```
EXCHANGE PARTITION { ( partition_name ) | FOR ( partition_value [, ...] ) }
WITH TABLE {[ ONLY ] ordinary_table_name | ordinary_table_name * | ONLY
( ordinary_table_name )}
[ { WITH | WITHOUT } VALIDATION ] [ VERBOSE ]
```

The ordinary table and partition whose data is to be exchanged must meet the following requirements:

- The number of columns of the ordinary table is the same as that of the partition, and their information should be consistent, including: column name, data type, constraint, collation information, storage parameter, and compress information.
- The compressed information of the ordinary table and partition table should be consistent.
- The distribution column information of the ordinary table and partition should be consistent.
- The number and information of indexes of the ordinary table and partition should be consistent.
- The number and information of constraints of the ordinary table and partition should be consistent.
- The ordinary table cannot be a temporary table or unlogged table.

When the exchange is done, the data and tablespace of the ordinary table and partition are exchanged. The statistics about ordinary tables and partitions become unreliable, and they should be analyzed again.

- The syntax of **row\_clause** is used to set the row movement switch of a partitioned table.

```
{ ENABLE | DISABLE } ROW MOVEMENT
```

- The **merge\_clause** syntax is used to merge partitions into one.

```
MERGE PARTITIONS { partition_name } [, ...] INTO PARTITION partition_name
```

- The syntax of **modify\_clause** is used to set whether a partition index is usable.

```
MODIFY PARTITION partition_name { UNUSABLE LOCAL INDEXES | REBUILD UNUSABLE LOCAL INDEXES }
```

- The **split\_clause** syntax is used to split one partition into partitions.

```
SPLIT PARTITION { partition_name | FOR ( partition_value [, ...] ) } { split_point_clause | no_split_point_clause }
```

- The syntax of specified **split\_point\_clause** is as follows:

```
AT ( partition_value ) INTO ( PARTITION partition_name , PARTITION partition_name )
```

---

#### NOTICE

- The size of split point should be in the range of splitting partition key. The split point can only split one partition into two.

- The syntax of **no\_split\_point\_clause** is as follows:

```
INTO { ( partition_less_than_item [, ...] ) | ( partition_start_end_item [, ...] ) }
```

---

#### NOTICE

- The first new partition key specified by **partition\_less\_than\_item** must be larger than that of the former partition (if any), and the last partition key specified by **partition\_less\_than\_item** must be equal to that of the splitting partition.
- The start point (if any) of the first new partition specified by **partition\_start\_end\_item** must be equal to the partition key (if any) of the previous partition. The end point (if any) of the last partition specified by **partition\_start\_end\_item** must be equal to the partition key of the splitting partition.
- **partition\_less\_than\_item** supports a maximum of four partition keys and **partition\_start\_end\_item** supports only one partition key. For details about the supported data types, see [partition key](#).

- The syntax of **partition\_less\_than\_item** is as follows:

```
PARTITION partition_name VALUES LESS THAN ( { partition_value | MAXVALUE } [, ...] )  
[ TABLESPACE tablespace_name ]
```

- The syntax of **partition\_start\_end\_item** is as follows. For details about the constraints, see [partition\\_start\\_end\\_item syntax](#).

```
PARTITION partition_name {  
  {START(partition_value) END (partition_value) EVERY (interval_value)} |  
  {START(partition_value) END ({partition_value | MAXVALUE})} |  
  {START(partition_value)} |  
  {END({partition_value | MAXVALUE})}  
} [TABLESPACE tablespace_name]
```

- The syntax of **add\_clause** is used to add a partition to one or more specified partitioned tables.

```
ADD {partition_less_than_item | partition_start_end_item}
```

- The syntax of **drop\_clause** is used to remove a specified partition from a partitioned table.

```
DROP PARTITION { partition_name | FOR ( partition_value [, ...] ) }
```

- The syntax of modifying a table partition name is as follows:

```
ALTER TABLE [ IF EXISTS ] { table_name [*] | ONLY table_name | ONLY ( table_name ) }
```

```
RENAME PARTITION { partition_name | FOR ( partition_value [, ...] ) } TO partition_new_name;
```

## Parameter Description

- **table\_name**  
Specifies the name of a partition table.  
Value range: An existing partition table name.
- **partition\_name**  
Specifies the name of a partition.  
Value range: An existing partition name.
- **partition\_value**  
Specifies the key value of a partition.  
The value specified by **PARTITION FOR ( partition\_value [, ...] )** can uniquely identify a partition.  
Value range: Value range of the partition key for the partition to be renamed.
- **UNUSABLE LOCAL INDEXES**  
Sets all the indexes unusable in the partition.
- **REBUILD UNUSABLE LOCAL INDEXES**  
Rebuilds all the indexes in the partition.
- **ENABLE/DISABLE ROW MOVEMENT**  
Specifies the row movement switch.  
If the tuple value is updated on the partition key during the **UPDATE** action, the partition where the tuple is located is altered. Setting of this parameter enables error messages to be reported or movement of the tuple between partitions.  
Valid value:
  - **ENABLE**: The row movement switch is enabled.
  - **DISABLE**: The row movement switch is disabled.
 The switch is disabled by default.
- **ordinary\_table\_name**  
Specifies the name of the ordinary table whose data is to be migrated.  
Value range: An existing ordinary table name.
- **{ WITH | WITHOUT } VALIDATION**  
Checks whether the ordinary table data meets the specified partition key range of the partition to be migrated.  
Valid value:
  - **WITH**: checks whether the common table data meets the partition key range of the partition to be exchanged. If any data does not meet the required range, an error is reported.
  - **WITHOUT**: does not check whether the common table data meets the partition key range of the partition to be exchanged.
 The default value is **WITH**.

The check is time consuming, especially when the data volume is large. Therefore, use **WITHOUT** when you are sure that the current common table data meets the partition key range of the partition to be exchanged.

- **VERBOSE**

When **VALIDATION** is **WITH**, if the ordinary table contains data that is out of the partition key range, insert the data to the correct partition. If there is no correct partition where the data can be route to, an error is reported.

---

**NOTICE**

Only when **VALIDATION** is **WITH**, **VERBOSE** can be specified.

- **partition\_new\_name**

Specifies the new name of a partition.

Value range: a string. It must comply with the naming convention rule.

## Examples

See [Examples](#) in **CREATE TABLE PARTITION**.

## Helpful Links

[CREATE TABLE PARTITION](#), [DROP TABLE](#)

# 12.22 ALTER TEXT SEARCH CONFIGURATION

## Function

**ALTER TEXT SEARCH CONFIGURATION** modifies the definition of a text search configuration. You can modify its mappings from token types to dictionaries, change the configuration's name or owner, or modify the parameters.

The **ADD MAPPING FOR** form installs a list of dictionaries to be consulted for the specified token types; an error will be generated if there is already a mapping for any of the token types.

The **ALTER MAPPING FOR** form removes existing mapping for those token types and then adds specified mappings.

**ALTER MAPPING REPLACE ... WITH ...** and **ALTER MAPPING FOR... REPLACE ... WITH ...** options replace **old\_dictionary** with **new\_dictionary**. Note that only when **pg\_ts\_config\_map** has tuples corresponding to **maptokentype** and **old\_dictionary**, the update will succeed. If the update fails, no messages are returned.

The **DROP MAPPING FOR** form deletes all dictionaries for the specified token types in the text search configuration. If **IF EXISTS** is not specified and the string type mapping specified by **DROP MAPPING FOR** does not exist in text search configuration, an error will occur in database.

## Precautions

- If a search configuration is referenced (to create an index), users are not allowed to modify it.
- To use **ALTER TEXT SEARCH CONFIGURATION**, you must be the owner of the configuration.

## Syntax

- Add text search configuration string mapping.

```
ALTER TEXT SEARCH CONFIGURATION name  
  ADD MAPPING FOR token_type [, ... ] WITH dictionary_name [, ... ];
```

- Modify the text search configuration dictionary syntax.

```
ALTER TEXT SEARCH CONFIGURATION name  
  ALTER MAPPING FOR token_type [, ... ] REPLACE old_dictionary WITH new_dictionary;
```

- Modify the text search configuration string.

```
ALTER TEXT SEARCH CONFIGURATION name  
  ALTER MAPPING FOR token_type [, ... ] WITH dictionary_name [, ... ];
```

- Change the text search configuration dictionary.

```
ALTER TEXT SEARCH CONFIGURATION name  
  ALTER MAPPING REPLACE old_dictionary WITH new_dictionary;
```

- Remove text search configuration string mapping.

```
ALTER TEXT SEARCH CONFIGURATION name  
  DROP MAPPING [ IF EXISTS ] FOR token_type [, ... ];
```

- Rename the owner of text search configuration.

```
ALTER TEXT SEARCH CONFIGURATION name OWNER TO new_owner;
```

- Rename the name of text search configuration.

```
ALTER TEXT SEARCH CONFIGURATION name RENAME TO new_name;
```

- Rename the namespace of text search configuration.

```
ALTER TEXT SEARCH CONFIGURATION name SET SCHEMA new_schema;
```

- Modify the attributes of text search configuration.

```
ALTER TEXT SEARCH CONFIGURATION name SET ( { configuration_option = value } [, ...] );
```

- Reset the attributes of text search configuration.

```
ALTER TEXT SEARCH CONFIGURATION name RESET ( {configuration_option} [, ...] );
```

## Parameter description

- **name**  
Specifies the name (optionally schema-qualified) of an existing text search configuration.
- **token\_type**  
Specifies the name of a token type that is emitted by the configuration's parser. For details, see [Parsers](#).
- **dictionary\_name**  
Specifies the name of a text search dictionary to be consulted for the specified token types. If multiple dictionaries are listed, they are consulted in the specified order.
- **old\_dictionary**  
Specifies the name of a text search dictionary to be replaced in the mapping.

- **new\_dictionary**  
Specifies the name of a text search dictionary to be substituted for **old\_dictionary**.
- **new\_owner**  
Specifies the new owner of the text search configuration.
- **new\_name**  
Specifies the new name of the text search configuration.
- **new\_schema**  
Specifies the new schema for the text search configuration.
- **configuration\_option**  
Text search configuration option. For details, see [CREATE TEXT SEARCH CONFIGURATION](#).
- **value**  
Specifies the value of text search configuration option.

## Examples

```
-- Create a text search configuration:
CREATE TEXT SEARCH CONFIGURATION english_1 (parser=default);
CREATE TEXT SEARCH CONFIGURATION

-- Add text search configuration string mapping:
ALTER TEXT SEARCH CONFIGURATION english_1 ADD MAPPING FOR word WITH simple,english_stem;
ALTER TEXT SEARCH CONFIGURATION

-- Add text search configuration string mapping:
ALTER TEXT SEARCH CONFIGURATION english_1 ADD MAPPING FOR email WITH english_stem,
french_stem;
ALTER TEXT SEARCH CONFIGURATION

-- Query information about the text search configuration:
SELECT b.cfgname,a.maptokentype,a.mapseqno,a.mapdict,c.dictname FROM pg_ts_config_map
a,pg_ts_config b, pg_ts_dict c WHERE a.mapcfg=b.oid AND a.mapdict=c.oid AND b.cfgname='english_1'
ORDER BY 1,2,3,4,5;
  cfgname | maptokentype | mapseqno | mapdict | dictname
-----+-----+-----+-----+-----
english_1 | 2 | 1 | 3765 | simple
english_1 | 2 | 2 | 12960 | english_stem
english_1 | 4 | 1 | 12960 | english_stem
english_1 | 4 | 2 | 12964 | french_stem
(4 rows)

-- Add text search configuration string mapping:
ALTER TEXT SEARCH CONFIGURATION english_1 ALTER MAPPING REPLACE french_stem with german_stem;
ALTER TEXT SEARCH CONFIGURATION

-- Query information about the text search configuration:
SELECT b.cfgname,a.maptokentype,a.mapseqno,a.mapdict,c.dictname FROM pg_ts_config_map
a,pg_ts_config b, pg_ts_dict c WHERE a.mapcfg=b.oid AND a.mapdict=c.oid AND b.cfgname='english_1'
ORDER BY 1,2,3,4,5;
  cfgname | maptokentype | mapseqno | mapdict | dictname
-----+-----+-----+-----+-----
english_1 | 2 | 1 | 3765 | simple
english_1 | 2 | 2 | 12960 | english_stem
english_1 | 4 | 1 | 12960 | english_stem
english_1 | 4 | 2 | 12966 | german_stem
(4 rows)
```

See [Examples](#) in **CREATE TEXT SEARCH CONFIGURATION**.

## Helpful Links

[CREATE TEXT SEARCH CONFIGURATION, DROP TEXT SEARCH CONFIGURATION](#)

# 12.23 ALTER TEXT SEARCH DICTIONARY

## Function

**ALTER TEXT SEARCH DICTIONARY** modifies the definition of a full-text retrieval dictionary, including its parameters, name, owner, and schema.

## Precautions

- **ALTER** is not supported by predefined dictionaries.
- Only the owner of a dictionary can do **ALTER** to the dictionary. System administrators have this permission by default.
- After a dictionary is created or modified, any modification to the user-defined dictionary definition file in the directory specified by **FilePath** will not affect the dictionary in the database. To make such modifications take effect in the dictionary in the database, run the **ALTER TEXT SEARCH DICTIONARY** statement to update the definition file of the dictionary.

## Syntax

- Modify the dictionary definition.  

```
ALTER TEXT SEARCH DICTIONARY name (  
    option [ = value ] [, ... ]  
);
```
- Rename a dictionary.  

```
ALTER TEXT SEARCH DICTIONARY name RENAME TO new_name;
```
- Set the schema of a dictionary.  

```
ALTER TEXT SEARCH DICTIONARY name SET SCHEMA new_schema;
```
- Change the owner of a dictionary.  

```
ALTER TEXT SEARCH DICTIONARY name OWNER TO new_owner;
```

## Parameter Description

- *name*  
Specifies the name of an existing dictionary. (If you do not specify a schema name, the dictionary in the current schema will be used.)  
Value range: name of an existing dictionary
- *option*  
Specifies the name of a parameter to be modified. Each type of dictionaries has a template containing their custom parameters. Parameters function in a way irrelevant to their setting sequence. For details about parameters, see [option](#).

 **NOTE**

- The **TEMPLATE** parameter in a dictionary cannot be modified.
- To specify a dictionary, specify both the dictionary definition file path (**FILEPATH**) and the file name (the parameter varies based on dictionary types).
- The name of a dictionary definition file can contain only lowercase letters, digits, and underscores (\_).
- *value*  
Specifies the new value of a parameter. If = and *value* are omitted, the previous settings of the parameter will be deleted and the default value will be used.  
Value range: valid values defined for the parameter
- *new\_name*  
Specifies the new name of a dictionary.  
Value range: a string, which complies with the identifier naming convention. A value can contain a maximum of 63 characters.
- *new\_owner*  
Specifies the new owner of a dictionary.  
Value range: an existing user name
- *new\_schema*  
Specifies the new schema of a dictionary.  
Value range: an existing schema name

## Examples

```
-- Modify the definition of stop words in Snowball dictionaries. Retain the values of other parameters.  
ALTER TEXT SEARCH DICTIONARY my_dict ( StopWords = newrussian, FilePath = 'obs://bucket_name/path  
accesskey=ak secretkey=sk region=rg' );  
  
-- Modify the Language parameter in Snowball dictionaries and delete the definition of stop words.  
ALTER TEXT SEARCH DICTIONARY my_dict ( Language = dutch, StopWords );  
  
-- Update the dictionary definition and do not change any other content.  
ALTER TEXT SEARCH DICTIONARY my_dict ( dummy );
```

## Helpful Links

[CREATE TEXT SEARCH DICTIONARY, DROP TEXT SEARCH DICTIONARY](#)

# 12.24 ALTER TRIGGER

## Function

**ALTER TRIGGER** modifies the definition of a trigger.

## Precautions

Only the owner of a table where a trigger is created and system administrators can run the **ALTER TRIGGER** statement.



## Syntax

```
ALTER TRIGGER trigger_name ON table_name RENAME TO new_name;
```

### Parameter Description

- **trigger\_name**  
Specifies the name of the trigger to be modified.  
Value range: an existing trigger
- **table\_name**  
Specifies the name of the table where the trigger to be modified is located.  
Value range: an existing table having a trigger
- **new\_name**  
Specifies the new name after modification.  
Value range: a string that complies with the identifier naming convention. A value contains a maximum of 63 characters and cannot be the same as other triggers on the same table.

### Examples

For details, see [CREATE TRIGGER](#).

### Helpful Links

[CREATE TRIGGER](#), [DROP TRIGGER](#), [ALTER TABLE](#)

## 12.25 ALTER TYPE

### Function

**ALTER TYPE** modifies the definition of a type.

### Syntax

- **Modify a type.**  

```
ALTER TYPE name action [ , ... ]  
ALTER TYPE name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }  
ALTER TYPE name RENAME ATTRIBUTE attribute_name TO new_attribute_name [ CASCADE |  
RESTRICT ]  
ALTER TYPE name RENAME TO new_name  
ALTER TYPE name SET SCHEMA new_schema  
ALTER TYPE name ADD VALUE [ IF NOT EXISTS ] new_enum_value [ { BEFORE | AFTER }  
neighbor_enum_value ]  
ALTER TYPE name RENAME VALUE existing_enum_value TO new_enum_value
```

where action is one of:

```
ADD ATTRIBUTE attribute_name data_type [ COLLATE collation ] [ CASCADE | RESTRICT ]  
DROP ATTRIBUTE [ IF EXISTS ] attribute_name [ CASCADE | RESTRICT ]  
ALTER ATTRIBUTE attribute_name [ SET DATA ] TYPE data_type [ COLLATE collation ] [ CASCADE |  
RESTRICT ]
```
- **Add a new attribute to a composite type.**  

```
ALTER TYPE name ADD ATTRIBUTE attribute_name data_type [ COLLATE collation ] [ CASCADE |  
RESTRICT ]
```

- Delete an attribute from a composite type.  
ALTER TYPE name DROP ATTRIBUTE [ IF EXISTS ] attribute\_name [ CASCADE | RESTRICT ]
- Change the type of an attribute in a composite type.  
ALTER TYPE name ALTER ATTRIBUTE attribute\_name [ SET DATA ] TYPE data\_type [ COLLATE collation ] [ CASCADE | RESTRICT ]
- Change the owner of a type.  
ALTER TYPE name OWNER TO { new\_owner | CURRENT\_USER | SESSION\_USER }
- Change the name of a type or the name of an attribute in a composite type.  
ALTER TYPE name RENAME TO new\_name  
ALTER TYPE name RENAME ATTRIBUTE attribute\_name TO new\_attribute\_name [ CASCADE | RESTRICT ]
- Move a type to a new schema.  
ALTER TYPE name SET SCHEMA new\_schema
- Add a new value to an enumerated type.  
ALTER TYPE name ADD VALUE [ IF NOT EXISTS ] new\_enum\_value [ { BEFORE | AFTER } neighbor\_enum\_value ]
- Change an enumerated value in the value list.  
ALTER TYPE name RENAME VALUE existing\_enum\_value TO new\_enum\_value

## Parameter Description

- **name**  
Specifies the name of an existing type that needs to be modified (schema-qualified).
- **new\_name**  
Specifies the new name of the type.
- **new\_owner**  
Specifies the new owner of the type.
- **new\_schema**  
Specifies the new schema of the type.
- **attribute\_name**  
Specifies the name of the attribute to be added, modified, or deleted.
- **new\_attribute\_name**  
Specifies the new name of the attribute to be renamed.
- **data\_type**  
Specifies the data type of the attribute to be added, or the new type of the attribute to be modified.
- **new\_enum\_value**  
Specifies a new enumerated value. It is a non-empty string with a maximum length of 64 bytes.
- **neighbor\_enum\_value**  
Specifies an existing enumerated value before or after which a new enumerated value will be added.
- **existing\_enum\_value**  
Specifies an enumerated value to be changed. It is a non-empty string with a maximum length of 64 bytes.
- **CASCADE**

Determines that the type to be modified, its associated records, and subtables that inherit the type will all be updated.

- **RESTRICT**

Refuses to update the association record of the modified type. This is the default.

---

**NOTICE**

- **ADD ATTRIBUTE, DROP ATTRIBUTE, and ALTER ATTRIBUTE** can be combined for processing. For example, it is possible to add several attributes or change the types of several attributes at the same time in one command.
  - Only type owners can run **ALTER TYPE**. To modify the schema of a type, you must also have the **CREATE** permission for the new schema. To modify the owner of a type, you must be a direct or indirect member of the new owner and have the **CREATE** permission for the schema of this type. (These restrictions force modification owners not to do anything that cannot be done by deleting and rebuilding types. However, system administrators can modify the ownership of any type in any way.) To add an attribute or modify the type of an attribute, you must also have the **USAGE** permission for this type.
- 

## Examples

For details, see [Examples](#) for **CREATE TYPE**.

## Helpful Links

[CREATE TYPE, DROP TYPE](#)

# 12.26 ALTER USER

## Function

**ALTER USER** modifies the attributes of a database user.

## Precautions

Session parameters modified by **ALTER USER** apply to a specified user and take effect in the next session.

## Syntax

- Modify user rights or other information.  
ALTER USER user\_name [ [ WITH ] option [ ... ] ];

The **option** clause is as follows:

```
{ CREATEDB | NOCREATEDB }  
| { CREATEROLE | NOCREATEROLE }  
| { INHERIT | NOINHERIT }  
| { AUDITADMIN | NOAUDITADMIN }
```

```

| { SYSADMIN | NOSYSADMIN }
| { USEFT | NOUSEFT }
| { LOGIN | NOLOGIN }
| { REPLICATION | NOREPLICATION }
| { INDEPENDENT | NOINDEPENDENT }
| { VCADMIN | NOVCADMIN }
| CONNECTION LIMIT connlimit
| [ ENCRYPTED | UNENCRYPTED ] PASSWORD { 'password' | DISABLE }
| [ ENCRYPTED | UNENCRYPTED ] IDENTIFIED BY { 'password' [ REPLACE 'old_password' ] |
DISABLE }
| VALID BEGIN 'timestamp'
| VALID UNTIL 'timestamp'
| RESOURCE POOL 'respool'
| USER GROUP 'groupuser'
| PERM SPACE 'spacelimit'
| TEMP SPACE 'tmpspacelimit'
| SPILL SPACE 'spillspacelimit'
| NODE GROUP logic_cluster_name
| ACCOUNT { LOCK | UNLOCK }
| PGUSER
| LDAP

```

- Change the user name.  
ALTER USER user\_name  
RENAME TO new\_name;
- Change the value of a specified parameter associated with the user.  
ALTER USER user\_name  
SET configuration\_parameter { { TO | = } { value | DEFAULT } | FROM CURRENT };
- Reset the value of a specified parameter associated with the user.  
ALTER USER user\_name  
RESET { configuration\_parameter | ALL };

## Parameter description

- **user\_name**  
Specifies the current user name.  
Value range: an existing user name
- **new\_password**  
Indicates a new password.  
A new password must:
  - Differ from the old password.
  - Contain at least eight characters. This is the default length.
  - Differ from the user name or the user name spelled backward.
  - Contain at least three of the following four character types: uppercase letters, lowercase letters, digits, and special characters, including: ~!@#\$%^&\*()-\_+=\|[]{};,:<.>/? . If you use characters other than the four types, a warning is displayed, but you can still create the password.
Value range: a string.
- **old\_password**  
Indicates the old password.
- **ACCOUNT LOCK | ACCOUNT UNLOCK**
  - **ACCOUNT LOCK:** locks an account to forbid login to databases.
  - **ACCOUNT UNLOCK:** unlocks an account to allow login to databases.
- **PGUSER**  
**PGUSER** of a user cannot be modified in the current version.

For details about other parameters, see "Parameter Description" in [CREATE ROLE](#) and [ALTER ROLE](#).

## Examples

See [Examples](#) in [CREATE USER](#).

## Helpful Links

[CREATE ROLE](#), [CREATE USER](#), [DROP USER](#)

# 12.27 ALTER VIEW

## Function

**ALTER VIEW** modifies all auxiliary attributes of a view. (To modify the query definition of a view, use **CREATE OR REPLACE VIEW**.)

## Precautions

- Only the view owner can modify a view by running **ALTER VIEW**.
- To change a view's schema, you must also have the CREATE permission on the new schema.
- To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the view's schema.
- An administrator can change the owner relationship of any view.

## Syntax

- Set the default value of the view column.  

```
ALTER VIEW [ IF EXISTS ] view_name  
ALTER [ COLUMN ] column_name SET DEFAULT expression;
```
- Remove the default value of the view column.  

```
ALTER VIEW [ IF EXISTS ] view_name  
ALTER [ COLUMN ] column_name DROP DEFAULT;
```
- Change the owner of a view.  

```
ALTER VIEW [ IF EXISTS ] view_name  
OWNER TO new_owner;
```
- Rename a view.  

```
ALTER VIEW [ IF EXISTS ] view_name  
RENAME TO new_name;
```
- Set the schema of the view.  

```
ALTER VIEW [ IF EXISTS ] view_name  
SET SCHEMA new_schema;
```
- Set the options of the view.  

```
ALTER VIEW [ IF EXISTS ] view_name  
SET ( { view_option_name [= view_option_value] } [, ...] );
```
- Reset the options of the view.  

```
ALTER VIEW [ IF EXISTS ] view_name  
RESET ( view_option_name [, ...] );
```

## Parameter Description

- **IF EXISTS**  
If this option is specified, no error is reported if the view does not exist. Only a message is displayed.
- **view\_name**  
Specifies the view name, which can be schema-qualified.  
Value range: A string. It must comply with the naming convention rule.
- **column\_name**  
Indicates an optional list of names to be used for columns of the view. If not given, the column names are deduced from the query.  
Value range: A string. It must comply with the naming convention rule.
- **SET/DROP DEFAULT**  
Sets or deletes the default value of a column. Currently, this parameter does not take effect.
- **new\_owner**  
Specifies the new owner of a view.
- **new\_name**  
Specifies the new view name.
- **new\_schema**  
Specifies the new schema of the view.
- **view\_option\_name [ = view\_option\_value ]**  
This clause specifies optional parameters for a view.  
Currently, the only parameter supported by **view\_option\_name** is **security\_barrier**, which should be enabled when a view is intended to provide row-level security.  
Value range: Boolean type. It can be **TRUE** or **FALSE**.

## Examples

```
-- Create a view consisting of rows with c_customer_sk smaller than 150:
CREATE VIEW tpcds.customer_details_view_v1 AS
  SELECT * FROM tpcds.customer
  WHERE c_customer_sk < 150;

-- Rename the view:
ALTER VIEW tpcds.customer_details_view_v1 RENAME TO customer_details_view_v2;

-- Change the schema of the view:
ALTER VIEW tpcds.customer_details_view_v2 SET schema public;

-- Delete the view:
DROP VIEW public.customer_details_view_v2;
```

## Helpful Links

[CREATE VIEW, DROP VIEW](#)

## 12.28 CLEAN CONNECTION

### Function

**CLEAN CONNECTION** clears database connections when a database is abnormal. You may use this statement to delete a specific user's connections to a specified database.

### Precautions

None

### Syntax

```
CLEAN CONNECTION  
TO { COORDINATOR ( nodename [, ... ] ) | NODE ( nodename [, ... ] ) | ALL [ CHECK ] [ FORCE ] }  
[ FOR DATABASE dbname ]  
[ TO USER username ];
```

### Parameter Description

- **CHECK**  
This parameter can be specified only when the node list is specified as **TO ALL**. Setting this parameter will check whether a database is accessed by other sessions before its connections are cleared. If any sessions are detected before **DROP DATABASE** is executed, an error will be reported and the database will not be deleted.
- **FORCE**  
This parameter can be specified only when the node list is specified as **TO ALL**. Setting this parameter will send SIGTERM signals to all the threads related to the specified **dbname** and **username** and forcibly shut them down.
- **COORDINATOR ( nodename ,nodename ... } ) | NODE ( nodename , nodename ... ) | ALL**  
Deletes connections on a specified node. There are three scenarios:
  - Deletes connections to a specified CN.
  - Deletes connections to a specified DN.
  - Deletes connections to all CNs and DNs.Value range: **nodename** is an existing node name.
- **dbname**  
Deletes connections to a specific database. If this parameter is not specified, connections to all databases will be deleted.  
Value range: An existing database name.
- **username**  
Deletes connections of a specific user. If this parameter is not specified, connections of all users will be deleted.  
Value range: An existing user name.

## Examples

```
-- Create user jack:  
CREATE USER jack PASSWORD 'Bigdata123@';  
  
-- Clean connections to nodes dn1 and dn2 for the template1 database:  
CLEAN CONNECTION TO NODE (dn_6001_6002,dn_6003_6004) FOR DATABASE template1;  
  
-- Clean user jack's connections to dn1:  
CLEAN CONNECTION TO NODE (dn_6001_6002) TO USER jack;  
  
-- Clean all the connections to the postgres database:  
CLEAN CONNECTION TO ALL FORCE FOR DATABASE postgres;  
  
-- Delete user jack:  
DROP USER jack;
```

## 12.29 CLOSE

### Function

**CLOSE** frees the resources associated with an open cursor.

### Precautions

- After a cursor is closed, no subsequent operations are allowed on it.
- A cursor should be closed when it is no longer needed.
- Every non-holdable open cursor is implicitly closed when a transaction is terminated by **COMMIT** or **ROLLBACK**.
- A holdable cursor is implicitly closed if the transaction that created it aborts via **ROLLBACK**.
- If the creating transaction successfully commits, the holdable cursor remains open until an explicit **CLOSE** is executed, or the client disconnects.
- GaussDB(DWS) does not have an explicit **OPEN** cursor statement. A cursor is considered open when it is declared. You can see all available cursors by querying the **pg\_cursors** system view.

### Syntax

```
CLOSE { cursor_name | ALL } ;
```

### Parameter Description

- **cursor\_name**  
Specifies the name of a cursor to be closed.
- **ALL**  
Closes all open cursors.

### Examples

See [Examples](#) in **FETCH**.

### Helpful Links

[FETCH](#), [MOVE](#)



## 12.30 CLUSTER

### Function

Cluster a table according to an index.

**CLUSTER** instructs GaussDB(DWS) to cluster the table specified by **table\_name** based on the index specified by **index\_name**. The index must have been defined on **table\_name**.

When a table is clustered, it is physically reordered based on the index information. Clustering is a one-time operation: when the table is subsequently updated, the changes are not clustered. That is, no attempt is made to store new or updated rows according to their index order.

When a table is clustered, GaussDB(DWS) records which index the table was clustered by. The form **CLUSTER table\_name** reclusters the table using the same index as before. You can also use the **CLUSTER** or **SET WITHOUT CLUSTER** forms of **ALTER TABLE** to set the index to be used for future cluster operations, or to clear any previous setting.

**CLUSTER** without any parameter reclusters all the previously-clustered tables in the current database that the calling user owns, or all such tables if called by an administrator.

When a table is being clustered, an **ACCESS EXCLUSIVE** lock is acquired on it. This prevents any other database operations (both reads and writes) from operating on the table until the **CLUSTER** is finished.

### Precautions

Only row-store B-tree indexes support **CLUSTER**.

In cases where you are accessing single rows randomly within a table, the actual order of the data in the table is unimportant. However, if you tend to access some data more than others, and there is an index that groups them together, you will benefit from using **CLUSTER**. If you are requesting a range of indexed values from a table, or a single indexed value that has multiple rows that match, **CLUSTER** will help because once the index identifies the table page for the first row that matches, all other rows that match are probably already on the same table page, and so you save disk accesses and speed up the query.

When an index scan is used, a temporary copy of the table is created that contains the table data in the index order. Temporary copies of each index on the table are created as well. Therefore, you need free space on disk at least equal to the sum of the table size and the index sizes.

Because **CLUSTER** remembers which indexes are clustered, one can cluster the tables manually the first time, then set up a time like **VACUUM** without any parameters, so that the desired tables are periodically reclustered.

Because the optimizer records statistics about the ordering of tables, it is advisable to run **ANALYZE** on the newly clustered table. Otherwise, the optimizer might make poor choices of query plans.

**CLUSTER** cannot be executed in transactions.

## Syntax

- Cluster a table.  
`CLUSTER [ VERBOSE ] table_name [ USING index_name ];`
- Cluster a partition.  
`CLUSTER [ VERBOSE ] table_name PARTITION ( partition_name ) [ USING index_name ];`
- Cluster the table that has previously been clustered.  
`CLUSTER [ VERBOSE ];`

## Parameter Description

- **VERBOSE**  
Enables the display of progress messages.
- **table\_name**  
Specifies the name of the table.  
Value range: an existing table name
- **index\_name**  
Name of this index  
Value range: An existing index name.
- **partition\_name**  
Specifies the partition name.  
Value range: An existing partition name.

## Examples

```
-- Create a partitioned table:
CREATE TABLE tpcds.inventory_p1
(
  INV_DATE_SK          INTEGER          NOT NULL,
  INV_ITEM_SK          INTEGER          NOT NULL,
  INV_WAREHOUSE_SK    INTEGER          NOT NULL,
  INV_QUANTITY_ON_HAND INTEGER
)
DISTRIBUTE BY HASH(INV_ITEM_SK)
PARTITION BY RANGE(INV_DATE_SK)
(
  PARTITION P1 VALUES LESS THAN(2451179),
  PARTITION P2 VALUES LESS THAN(2451544),
  PARTITION P3 VALUES LESS THAN(2451910),
  PARTITION P4 VALUES LESS THAN(2452275),
  PARTITION P5 VALUES LESS THAN(2452640),
  PARTITION P6 VALUES LESS THAN(2453005),
  PARTITION P7 VALUES LESS THAN(MAXVALUE)
);

-- Create an index named ds_inventory_p1_index1:
CREATE INDEX ds_inventory_p1_index1 ON tpcds.inventory_p1 (INV_ITEM_SK) LOCAL;

-- Cluster the tpcds.inventory_p1 table:
CLUSTER tpcds.inventory_p1 USING ds_inventory_p1_index1;

-- Cluster the p3 partition:
CLUSTER tpcds.inventory_p1 PARTITION (p3) USING ds_inventory_p1_index1;

-- Cluster the tables that can be clustered in the database:
CLUSTER;
```

```
-- Drop an index:  
DROP INDEX tpcds.ds_inventory_p1_index1;  
  
-- Drop the partitioned table:  
DROP TABLE tpcds.inventory_p1;
```

## 12.31 COMMENT

### Function

**COMMENT** defines or changes the comment of an object.

### Precautions

- Only one comment string is stored for each object. To modify a comment, issue a new **COMMENT** command for the same object. To remove a comment, write **NULL** in place of the text string. Comments are automatically deleted when their objects are deleted.
- Currently, there is no security protection for viewing comments. Any user connected to a database can view all the comments for objects in the database. For shared objects such as databases, roles, and tablespaces, comments are stored globally so any user connected to any database in the cluster can see all the comments for shared objects. Therefore, do not put security-critical information in comments.
- For most kinds of objects, only the owner of objects can set the comment. Roles do not have owners, so the rule for **COMMENT ON ROLE** is that you must be administrator to comment on an administrator role, or have the **CREATEROLE** permission to comment on non-administrator roles. An administrator can comment on anything.

### Syntax

```
COMMENT ON  
{  
  AGGREGATE agg_name (agg_type [, ...] ) |  
  CAST (source_type AS target_type) |  
  COLLATION object_name |  
  COLUMN { table_name.column_name | view_name.column_name } |  
  CONSTRAINT constraint_name ON table_name |  
  CONVERSION object_name |  
  DATABASE object_name |  
  DOMAIN object_name |  
  EXTENSION object_name |  
  FOREIGN DATA WRAPPER object_name |  
  FOREIGN TABLE object_name |  
  FUNCTION function_name ( [ [ argmode ] [ argname ] argtype] [, ...] ) |  
  INDEX object_name |  
  LARGE OBJECT large_object_oid |  
  OPERATOR operator_name (left_type, right_type) |  
  OPERATOR CLASS object_name USING index_method |  
  OPERATOR FAMILY object_name USING index_method |  
  [ PROCEDURAL ] LANGUAGE object_name |  
  ROLE object_name |  
  RULE rule_name ON table_name |  
  SCHEMA object_name |  
  SERVER object_name |  
  TABLE object_name |  
  TABLESPACE object_name |  
  TEXT SEARCH CONFIGURATION object_name |
```

```
TEXT SEARCH DICTIONARY object_name |  
TEXT SEARCH PARSER object_name |  
TEXT SEARCH TEMPLATE object_name |  
TYPE object_name |  
VIEW object_name  
}  
IS 'text';
```

## Parameter Description

- **agg\_name**  
Specifies the new name of an aggregation function.
- **agg\_type**  
Specifies the data types of the aggregation function parameters.
- **source\_type**  
Specifies the name of the source data type of the cast.
- **target\_type**  
Specifies the name of the target data type of the cast.
- **object\_name**  
Specifies the name of the object to be commented.
- **table\_name.column\_name**  
**view\_name.column\_name**  
Specifies the column whose comment is defined or modified. You can add the table name or view name as the prefix.
- **constraint\_name**  
Specifies the table constraints whose comment is defined or modified.
- **table\_name**  
Specifies the table name.
- **function\_name**  
Specifies the function whose comment is defined or modified.
- **argmode,argname,argtype**  
Specifies the schema, name, and type of the function parameters.
- **large\_object\_oid**  
Specifies the OID of the large object whose comment is defined or modified.
- **operator\_name**  
Specifies the name of the operator.
- **left\_type,right\_type**  
The data type(s) of the operator's arguments (optionally schema-qualified). Write NONE for the missing argument of a prefix or postfix operator.
- **text**  
Specifies the new comment, written as a string literal; or **NULL** to drop the comment.

## Examples

```
CREATE TABLE tpceds.customer_demographics_t2  
(
```

```
CD_DEMO_SK          INTEGER          NOT NULL,
CD_GENDER           CHAR(1)
CD_MARITAL_STATUS  CHAR(1)
CD_EDUCATION_STATUS CHAR(20)
CD_PURCHASE_ESTIMATE INTEGER
CD_CREDIT_RATING   CHAR(10)
CD_DEP_COUNT       INTEGER
CD_DEP_EMPLOYED_COUNT INTEGER
CD_DEP_COLLEGE_COUNT INTEGER
)
WITH (ORIENTATION = COLUMN,COMPRESSION=MIDDLE)
DISTRIBUTE BY HASH (CD_DEMO_SK);

-- Add a comment to the tpcds.customer_demographics_t2.cd_demo_sk column:
COMMENT ON COLUMN tpcds.customer_demographics_t2.cd_demo_sk IS 'Primary key of customer
demographics table.';

-- Create a view consisting of rows with c_customer_sk smaller than 150:
CREATE VIEW tpcds.customer_details_view_v2 AS
SELECT *
FROM tpcds.customer
WHERE c_customer_sk < 150;

-- Add a comment to the tpcds.customer_details_view_v2 view:
COMMENT ON VIEW tpcds.customer_details_view_v2 IS 'View of customer detail';

-- Delete the view:
DROP VIEW tpcds.customer_details_view_v2;

-- Delete the tpcds.customer_demographics_t2 table:
DROP TABLE tpcds.customer_demographics_t2;
```

## 12.32 CREATE BARRIER

### Function

Creates a barrier for cluster nodes. The barrier can be used for data restoration.

### Precautions

Before creating a barrier, ensure that **gtm\_backup\_barrier** and **enable\_cbm\_tracking** are set to **on** for CNs and DN in the cluster.

### Syntax

```
CREATE BARRIER [ barrier_name ] ;
```

### Parameter Description

#### **barrier\_name**

(Optional) Indicates the name of a barrier.

Value range: a string. It must comply with the naming convention.

### Examples

```
-- Create a barrier without specifying its name:
CREATE BARRIER;

-- Specify a barrier name:
CREATE BARRIER 'barrier1';
```

## 12.33 CREATE DATABASE

### Function

**CREATE DATABASE** creates a database. By default, the new database will be created by cloning the standard system database template1. A different template can be specified using **TEMPLATE** *template name*.

### Precautions

- A user that has the **CREATEDB** permission or a sysadmin can create a database.
- **CREATE DATABASE** cannot be executed inside a transaction block.
- Errors along the line of "could not initialize database directory" are most likely related to insufficient permissions on the data directory, a full disk, or other file system problems.

### Syntax

```
CREATE DATABASE database_name
  [ [ WITH ] { [ OWNER [=] user_name ] |
    [ TEMPLATE [=] template ] |
    [ ENCODING [=] encoding ] |
    [ LC_COLLATE [=] lc_collate ] |
    [ LC_CTYPE [=] lc_ctype ] |
    [ DBCOMPATIBILITY [=] compatibilty_type ] |
    [ CONNECTION LIMIT [=] connlimit ] }[...];
```

### Parameter Description

- **database\_name**  
Indicates the database name.  
Value range: a string. It must comply with the naming convention rule.
- **OWNER [=] user\_name**  
Indicates the owner of the new database. By default, the owner of the database is the current user.  
Value range: an existing user name
- **TEMPLATE [=] template**  
Indicates the template name, that is, the name of the template to be used to create the database. GaussDB(DWS) creates a database by coping a database template. GaussDB(DWS) has two default template databases **template0** and **template1** and a default user database **postgres**.  
Value range: An existing database name. If this it is not specified, the system copies **template1** by default. Its value cannot be **postgres**.

---

**NOTICE**

Currently, database templates cannot contain sequences. If a database template contains sequences, database creation using this template will fail.

---

- **ENCODING [ = ] encoding**

Specifies the encoding format used by the new database. The value can be a string (for example, **SQL\_ASCII**) or an integer.

By default, the encoding format of the template database is used. The encoding formats of the template databases **template0** and **template1** vary based on OS environments by default. The encoding format of **template1** cannot be changed. If you need to change the encoding format of a database later, use **template0** to create the database.

Common values: **GBK**, **UTF8**, and **Latin1**

---

**NOTICE**

The character set encoding of the new database must be compatible with the local settings (**LC\_COLLATE** and **LC\_CTYPE**).

---

- **LC\_COLLATE [ = ] lc\_collate**

Specifies the collation order to use in the new database. For example, this parameter can be set using `lc_collate = 'zh_CN.gbk'`.

The use of this parameter affects the sort order applied to strings, for example, in queries with **ORDER BY**, as well as the order used in indexes on text columns. The default is to use the collation order of the template database.

Value range: A valid order type.

- **LC\_CTYPE [ = ] lc\_ctype**

Specifies the character classification to use in the new database. For example, this parameter can be set using `lc_ctype = 'zh_CN.gbk'`. The use of this parameter affects the categorization of characters, for example, lower, upper and digit. The default is to use the character classification of the template database.

Value range: A valid character type.

- **DBCMPATIBILITY [ = ] compatilbty\_type**

Specifies the compatible database type.

Valid values: **TD** and **ORA**, indicating the compatibility with Teradata and Oracle, respectively. If this parameter is not specified, the default value **ORA** is used.

- **TABLESPACE [ = ] tablespace\_name**

Specifies the name of the tablespace that will be associated with the new database.

Value range: An existing tablespace name.

- **CONNECTION LIMIT [ = ] connlimit**

Indicates the maximum number of concurrent connections that can be made to the new database.

Value range: An integer greater than or equal to **-1**. The default value **-1** means no limit.

**NOTICE**

- This limit does not apply to sysadmin.
- To ensure the proper running of a cluster, the minimum value of **CONNECTION LIMIT** is the number of CNs in the cluster, because when a cluster runs ANALYZE on a CN, other CNs will connect with the running CN for metadata synchronization. For example, if there are three CNs in the cluster, set **CONNECTION LIMIT** to 3 or a greater value.

The following are limitations on character encoding:

- If the locale is **C** (or equivalently **POSIX**), then all encoding modes are allowed, but for other locale settings only the encoding consistent with that of the locale will work properly.
- The encoding and locale settings must match those of the template database, except when template0 is used as template. This is because other databases might contain data that does not match the specified encoding, or might contain indexes whose sort ordering is affected by **LC\_COLLATE** and **LC\_CTYPE**. Copying such data would result in a database that is corrupt according to the new settings. template0, however, is known to not contain any data or indexes that would be affected.
- Supported encoding depends on the environment. If the message "invalid locale name" is displayed, run the **locale -a** command to check the encoding set supported by the environment.

## Examples

```
-- Create users jim and tom:
CREATE USER jim PASSWORD 'Bigdata123@';
CREATE USER tom PASSWORD 'Bigdata123@';

-- Create database music using GBK (the local encoding type is also GBK):
CREATE DATABASE music ENCODING 'GBK' template = template0;

-- Create database music2 and specify JIM as its owner:
CREATE DATABASE music2 OWNER jim;

-- Create database music3 using template template0 and specify jim as its owner:
CREATE DATABASE music3 OWNER jim TEMPLATE template0;

-- Set the maximum number of connections to database music to 10:
ALTER DATABASE music CONNECTION LIMIT= 10;

-- Rename database music to music4:
ALTER DATABASE music RENAME TO music4;

-- Change the owner of database music2 to tom:
ALTER DATABASE music2 OWNER TO tom;

-- Set the tablespace of database music3 to PG_DEFAULT:
ALTER DATABASE music3 SET TABLESPACE PG_DEFAULT;

-- Close the default index scan on database music3:
ALTER DATABASE music3 SET enable_indexscan TO off;

-- Reset parameter enable_indexscan:
ALTER DATABASE music3 RESET enable_indexscan;

Delete the databases:
DROP DATABASE music2;
```



```
DROP DATABASE music3;
DROP DATABASE music4;

-- Delete users jim and tom:
DROP USER jim;
DROP USER tom;

-- Create a database compatible with Teradata:
CREATE DATABASE td_compatible_db DBCOMPATIBILITY 'TD';

-- Create a database compatible with Oracle:
CREATE DATABASE ora_compatible_db DBCOMPATIBILITY 'ORA';

-- Delete the databases compatible with Teradata or Oracle:
DROP DATABASE td_compatible_db;
DROP DATABASE ora_compatible_db;
```

## Helpful Links

[ALTER DATABASE, DROP DATABASE](#)

## 12.34 CREATE FOREIGN TABLE (for GDS Import and Export)

**CREATE FOREIGN TABLE** creates a GDS foreign table.

### Function

**CREATE FOREIGN TABLE** creates a GDS foreign table in the current database for concurrent data import and export. The GDS foreign table can be read-only or write-only, used for concurrent data import and export, respectively. The OBS foreign table is read-only by default.

### Precautions

- The foreign table is owned by the user who runs the command.
- The distribution mode of a GDS foreign table does not need to be explicitly specified. The default is **ROUNDROBIN**.
- All constraints (including column and row constraints) are invalid to the GDS foreign table.

### Syntax

```
CREATE FOREIGN TABLE [ IF NOT EXISTS ] table_name
( [ { column_name type_name POSITION(offset,length) | LIKE source_table } [, ...] ] )
SERVER gsmpp_server
OPTIONS ( { option_name 'value' } [, ...] )
[ { WRITE ONLY | READ ONLY } ]
[ WITH error_table_name | LOG INTO error_table_name ]
[ REMOTE LOG 'name' ]
[ PER NODE REJECT LIMIT 'value' ]
[ TO { GROUP groupname | NODE ( nodename [, ...] ) } ];
```

### Parameter Overview

**CREATE FOREIGN TABLE** provides multiple parameters, which are classified as follows:

- Mandatory parameters
  - **table\_name**
  - **column\_name**
  - **type\_name**
  - **SERVER gsmpp\_server**
  - **OPTIONS**
- Optional parameters
  - Data source location parameter for foreign tables: **location**
  - Data format parameters
    - **format**
    - **header** (only for CSV and FIXED source data files)
    - **fileheader** (only for CSV and FIXED source data files)
    - **out\_filename\_prefix**
    - **delimiter**
    - **quote** (only for CSV source data files)
    - **escape** (only for CSV source data files)
    - **null**
    - **noescaping** (only for TEXT source data files)
    - **encoding**
    - **eol**
    - **conflict\_delimiter**
  - Error-tolerance parameters
    - **fill\_missing\_fields**
    - **ignore\_extra\_data**
    - **reject\_limit**
    - **compatible\_illegal\_chars**
    - **WITH error\_table\_name**
    - **LOG INTO error\_table\_name**
    - **REMOTE LOG 'name'**
    - **PER NODE REJECT LIMIT 'value'**

## Parameter Description

- **IF NOT EXISTS**  
Does not throw an error if a table with the same name already exists. A notice is issued in this case.
- **table\_name**  
Specifies the name of the foreign table to be created.  
Value range: a string. It must comply with the naming convention rule.
- **column\_name**  
Specifies the name of a column in the foreign table.  
Value range: a string. It must comply with the naming convention rule.
- **type\_name**  
Specifies the data type of the column.
- **POSITION(offset,length)**  
Defining the location of each column in the data file in fixed length mode.

### NOTE

**offset** is the start of the column in the source file, and **length** is the length of the column.

Value range: **offset** must be greater than 0 bytes, and its unit is byte.

The length of each record must be less than 1 GB. By default, columns not in the file are replaced with null.

- **SERVER gsmpp\_server**  
Specifies the server name of the foreign table. For the GDS foreign table, its server is created by initial database, which is **gsmpp\_server**.
- **OPTIONS ( { option\_name ' value ' } [, ...] )**  
Specifies all types of parameters of foreign table data.
  - location  
Specifies the data source location of the foreign table, which can be expressed through URLs. Separate URLs with vertical bars (|).  
Currently, GDS can automatically create a directory defined by a foreign table during data export. For example, when the foreign table **location** defines that **gsfs:// 192.168.0.91:5000/2019/09** executes an export task, if the **2019/09** subdirectory in the GDS data directory does not exist, the subdirectory is automatically created. You do not need to manually create the directory specified in the foreign table.

 NOTE

- For a read-only foreign table imported by GDS from a remote server in parallel, its URL must end with its corresponding schema or file name. (Read-only is the default file attribute.)  
For example: `gsfs://192.168.0.90:5000/*` or `file:///data/data.txt` or `gsfs://192.168.0.90:5000/* | gsfs:// 192.168.0.91:5000/*`.
- For a writable foreign table exported by GDS to a remote server in parallel, its URL does not need to contain a file name. If the data source location is a remote URL, for example, `gsfs:// 192.168.0.90:5000/`, multiple data sources can be specified. If the number of exported data file locations is less than or equal to the number of DNs, when you use the foreign table for export, data is evenly distributed to each data source location. If the number of exported data file locations is greater than the number of DNs, when you export data, the data is evenly distributed to data source locations corresponding to the DNs. Blank data files are created on the excess data source locations.
- For a read-only foreign table imported by GDS from a remote server in parallel, the number of URLs must be less than the number of DNs, and URLs in the same location cannot be used.
- If the URL begins with `gsfss://`, data is imported and exported in encryption mode, and DOP cannot exceed 10.
- During GDS export, the `2019/09` subdirectory in THE `gsfs:// 127.0.0.1:7789/2019/09/` directory specified by the `location` table is automatically created when the export task is executed.

## – format

Specifies the format of the data source file in a foreign table.

Value range: CSV, TEXT, or FIXED. The default value is TEXT.

GaussDB(DWS) only supports CSV and TEXT formats.

- In CSV files, escape sequences are processed as common strings. Therefore, linefeeds are processed as data.
- In TEXT files, escape sequences are processed as they are. Therefore, linefeeds are not processed as data.
- The FIXED file can process newline characters in data columns efficiently, but cannot process special characters very well.

 NOTE

- An escape sequence is a string starting with a backslash (\), including `\b` (backspace), `\f` (formfeed page break), `\n` (new line), `\r` (carriage return), `\t` (horizontal tab), `\v` (vertical tab), `\number` (octal number), and `\xnumber` (hexadecimal number). In TEXT files, strings are processed as they are. In files of other formats, strings are processed as data.
- **FIXED** is defined as follows: (**POSITION** must be specified for each column when **FIXED** is used.)
  1. The column length of each record is the same.
  2. Spaces are used for column padding. Left padding is used for the numeric type and right padding is used for the char type.
  3. No delimiters are used between columns.

## – header

Specifies whether a data file contains a table header. header is available only for CSV and FIXED files.

When data is imported, if **header** is **on**, the first row of the data file will be identified as title row and ignored. If header is **off**, the first row is identified as data.

When data is exported, if **header** is **on**, **fileheader** must be specified. **fileheader** is used to specify the export header file format. If header is **off**, the exported file does not include a title row.

Value range: **true**, **on**, **false**, and **off**. The default value is **false** or **off**.

– fileheader

Specifies a file that defines the content in the header for exported data. The file contains one row of data description of each column.

For example, to add a header in a file containing product information, define the file as follows:

The information of products.\n

---

**NOTICE**

- This parameter is available only when **header** is **on** or **true**. The file must be prepared in advance.
- In Remote mode, the definition file must be put to the working directory of GDS (the **-d** directory specified when starting the GDS).
- The definition file can contain only one row of title information, and end with a newline character. Excess rows will be discarded. (Title information cannot contain newline character).
- The length of the definition file including the newline character cannot exceed 1 MB.

---

– out\_filename\_prefix

Specifies the name prefix of the exported data file exported using GDS from a write-only foreign table.

**NOTICE**

- The prefix of the specified file name must be valid and compliant with the restrictions of the file system in the physical environment where the GDS is deployed. Otherwise, the file will fail to be created.
  - The file name prefix can contain only lowercase letters, uppercase letters, digits, and underscores (\_).
  - The prefix of the specified export file name cannot contain feature fields reserved for the Windows and Linux OS, including but not limited to:  
"con","aux","nul","prn","com0","com1","com2","com3","com4","com5","com6","com7","com8","com9","lpt0","lpt1","lpt2","lpt3","lpt4","lpt5","lpt6","lpt7","lpt8","lpt9"
  - The total length of the absolute path consisting of the exported file prefix, the path specified by **gds -d** and **.dat** should be as required by the file system where GDS is deployed.
  - It is required that the prefix can be correctly parsed and identified by the receiver (including but not limited to the original database where it was exported) of the data file. Identify and modify the option that causes the file name resolution problem (if any).
- To concurrently perform export jobs, do not use the same file name prefix for them. Otherwise, the exported files may overwrite each other or be lost in the OS or file system.

## - delimiter

Specifies the column delimiter of data, and uses the default delimiter if it is not set. The default delimiter of TEXT is a tab and that of CSV is a comma (.). No delimiter is used in FIXED format.

 **NOTE**

- A delimiter cannot be \r or \n.
- A delimiter cannot be the same as the **null** value. The delimiter of CSV cannot be same as the **quote** value.
- The delimiter for the TEXT format data cannot contain any of the following characters: \.abcdefghijklmnopqrstuvwxyz0123456789.
- The data length of a single row should be less than 1 GB. If the delimiters are too long and there are too many rows, the length of valid data will be affected.
- You are advised to use a multi-character, such as the combination of the dollar sign (\$), caret (^), the ampersand (&), or invisible characters, such as 0x07, 0x08, and 0x1b as the delimiter.
- For a multi-character delimiter, do not use the same characters, for example, ---.

Valid value:

The value of **delimiter** can be a multi-character delimiter whose length is less than or equal to 10 bytes.

## - quote

Specifies which characters in a CSV source data file will be identified as quotation marks. The default value is a double quotation mark (").

 NOTE

- The quote parameter cannot be the same as the delimiter or null parameter.
- The **quote** parameter must be a single-byte character.
- Invisible characters are recommended as **quote** values, such as 0x07, 0x08, and 0x1b.

- escape

Specifies which characters in a CSV source data file are escape characters. Escape characters can only be single-byte characters.

The default value is a double quotation mark ("). If it is the same as the value of **quote**, it will be replaced with \0.

- null

Specifies the string that represents a null value.

 NOTE

- The null value cannot be \r or \n. The maximum length is 100 characters.
- The **null** value cannot be the same as the delimiter or **quote** parameter.

Valid value:

- The default value is \n for the TEXT format.
- The default value for the CSV format is an empty string without quotation marks.

- noescaping

Specifies in TEXT format, whether to escape the backslash (\) and its following characters.

 NOTE

**noescaping** is available only for the TEXT format.

Value range: **true**, **on**, **false**, and **off**. The default value is **false** or **off**.

- encoding

Specifies the encoding of a data file, that is, the encoding used to parse, check, and generate a data file. Its default value is the default **client\_encoding** value of the current database.

Before you import foreign tables, it is recommended that you set **client\_encoding** to the file encoding format, or a format matching the character set of the file. Otherwise, unnecessary parsing and check errors may occur, leading to import errors, rollback, or even invalid data import. Before you import foreign tables, you are also advised to specify this parameter, because the export result using the default character set may not be what you expected.

If this parameter is not specified when you create a foreign table, a warning message will be displayed on the client.

 NOTE

Currently, GDS cannot parse or write in a file using multiple encoding formats during foreign table import or export.

- fill\_missing\_fields

Specifies whether to generate an error message when the last column in a row in the source file is lost during data import.

Value range: **true**, **on**, **false**, and **off**. The default value is **false** or **off**.

- If this parameter is set to **true** or **on** and the last column of a data row in a data source file is lost, the column will be replaced with **NULL** and no error message will be generated.
- If this parameter is set to **false** or **off** and the last column is missing, the following error information will be displayed:  
missing data for column "tt"

– ignore\_extra\_data

Specifies whether to ignore excessive columns when the number of data source files exceeds the number of foreign table columns. This parameter is available during data import.

Value range: **true**, **on**, **false**, and **off**. The default value is **false** or **off**.

- If this parameter is set to **true** or **on** and the number of data source files exceeds the number of foreign table columns, excessive columns will be ignored.
- If this parameter is set to **false** or **off** and the number of data source files exceeds the number of foreign table columns, the following error information will be displayed:  
extra data after last expected column

---

**NOTICE**

If the newline character at the end of the row is lost, setting the parameter to **true** will ignore data in the next row.

---

– reject\_limit

Specifies the maximum number of data format errors allowed during a data import task. If the number of errors does not reach the maximum number, the data import task can still be executed.

---

**NOTICE**

You are advised to replace this syntax with **PER NODE REJECT LIMIT 'value'**.

Examples of data format errors include the following: a column is lost, an extra column exists, a data type is incorrect, and encoding is incorrect. Once a non-data format error occurs, the whole data import process is stopped.

---

Value range: a positive integer or **unlimited**

The default value is **0**, indicating that error information is returned immediately.



 NOTE

Enclose positive integer values with single quotation marks (").

– mode

Specifies the data import policy during a specific data import process. GaussDB(DWS) supports only the **Normal** mode.

Valid value:

- **Normal** (default): supports all file types (CSV, TEXT, FIXED). Enabling Gauss data service to help data import.

– eol

Specifies the newline character style of the imported or exported data file.

Value range: multi-character newline characters within 10 bytes. Common newline characters include `\r` (0x0D), `\n` (0x0A), and `\r\n` (0x0D0A). Special newline characters include `$` and `#`.

 NOTE

- The **eol** parameter supports only the TEXT format for data import and export and does not support the CSV or FIXED format for data import. For forward compatibility, the **eol** parameter can be set to **0x0D** or **0x0D0A** for data export in the CSV and FIXED formats.
- The value of the **eol** parameter cannot be the same as that of **DELIMITER** or **NULL**.
- The **eol** parameter value cannot contain lowercase letters, digits, or dot (.).

– conflict\_delimiter

This parameter is generally used with the [compatible\\_illegal\\_chars](#) parameter. If a data file contains a truncated Chinese character, the truncated character and a delimiter will be encoded into another Chinese character due to inconsistent encoding between the foreign table and the database. As a result, the delimiter is masked and an error will be reported, indicating that there are missing fields.

This parameter is used to avoid encoding a truncated character and a delimiter into another character.

Value range: **true**, **on**, **false**, and **off**. The default value is **false** or **off**.

- If the parameter is set to **true** or **on**, encoding a truncated character and a delimiter into another character is allowed.
- If the parameter is set to **false** or **off**, encoding a truncated character and a delimiter into another character is not allowed.

---

**NOTICE**

This parameter is disabled by default. It is recommended that you disable this parameter, because encoding a truncated character and a delimiter into another character is rarely required. If the parameter is enabled, the scenario may be incorrectly identified and thereby causing incorrect information imported to the table.


---

- **fix**  
Specifies the length of fixed format data. The unit is byte. This syntax is available only for READ ONLY foreign tables.  
Value range: Less than **1 GB**, and greater than or equal to the total length specified by **POSITION** (The total length is the sum of **offset** and **length** in the last column of the table definition.)
- **out\_fix\_alignment**  
Specifies how the columns of the types BYTEAOID, CHAROID, NAMEOID, TEXTOID, BPCHAROID, VARCHAROID, NVARCHAR2OID, and CSTRINGOID are aligned during fixed-length export.  
Value range: **align\_left**, **align\_right**  
Default value: **align\_right**

---

**NOTICE**

The bytea data type must be in hexadecimal format (for example, \XXXX) or octal format (for example, \XXX\XXX\XXX). The data to be imported must be left-aligned (that is, the column data starts with either of the two formats instead of spaces). Therefore, if the exported file needs to be imported using a GDS foreign table and the file data length is less than that specified by the foreign table formatter, the exported file must be left aligned. Otherwise, an error is reported during the import.

- 
- **date\_format**  
Imports data of the DATE type. This syntax is available only for READ ONLY foreign tables.  
Value range: any valid DATE value. For details, see [Date and Time Processing Functions and Operators](#).
  -  **NOTE**  
If ORACLE is specified as the compatible database, the DATE format is TIMESTAMP. For details, see [timestamp\\_format](#) below.
  - **time\_format**  
Imports data of the TIME type. This syntax is available only for READ ONLY foreign tables.  
Value range: any valid TIME value. Time zones cannot be used. For details, see [Date and Time Processing Functions and Operators](#).
  - **timestamp\_format**  
Imports data of the TIMESTAMP type. This syntax is available only for READ ONLY foreign tables.  
Value range: any valid TIMESTAMP value. Time zones are not supported. For details, see [Date and Time Processing Functions and Operators](#).
  - **smalldatetime\_format**  
Imports data of the SMALLDATETIME type. This syntax is available only for READ ONLY foreign tables.  
Value range: any valid SMALLDATETIME value. For details, see [Date and Time Processing Functions and Operators](#).

- **compatible\_illegal\_chars**  
Enables or disables fault tolerance on invalid characters during data import. This syntax is available only for READ ONLY foreign tables.  
Value range: **true**, **on**, **false**, and **off**. The default value is **false** or **off**.
  - If this parameter is set to **true** or **on**, invalid characters are tolerated and imported to the database after conversion.
  - If this parameter is set to **false** or **off** and an error occurs when there are invalid characters, the import will be interrupted.

 **NOTE**

The rule of error tolerance when you import invalid characters is as follows:

- **\0** is converted to a space.
  - Other invalid characters are converted to question marks.
  - If **compatible\_illegal\_chars** is set to **true** or **on**, invalid characters are tolerated. If **NULL**, **DELIMITER**, **QUOTE**, and **ESCAPE** are set to spaces or question marks, errors like "illegal chars conversion may confuse COPY escape 0x20" will be displayed to prompt users to modify parameter values that cause confusion, preventing import errors.
- **READ ONLY**  
Specifies whether a foreign table is read-only. This parameter is available only for data import.
  - **WRITE ONLY**  
Specifies whether a foreign table is write-only. This parameter is available only for data import.
  - **WITH error\_table\_name**  
Specifies the table where data format errors generated during parallel data import are recorded. You can query the error information table after data is imported to obtain error details. This parameter is available only after **reject\_limit** is set.

 **NOTE**

To be compatible with PostgreSQL open source interfaces, you are advised to replace this syntax with **LOG INTO**.

Value range: a string. It must comply with the naming convention rule.

- **LOG INTO error\_table\_name**  
Specifies the table where data format errors generated during parallel data import are recorded. You can query the error information table after data is imported to obtain error details.

 **NOTE**

This parameter is available only after **PER NODE REJECT LIMIT** is set.

Value range: a string. It must comply with the naming convention rule.

- **REMOTE LOG 'name'**  
The data format error information is saved as files in GDS. **name** is the prefix of the error data file.
- **PER NODE REJECT LIMIT 'value'**

This parameter specifies the allowed number of data format errors on each DN during data import. If the number of errors exceeds the specified value on any DN, data import fails, an error is reported, and the system exits data import.

#### NOTICE

This syntax specifies the error tolerance of a single node.

Examples of data format errors include the following: a column is lost, an extra column exists, a data type is incorrect, and encoding is incorrect. When a non-data format error occurs, the whole data import process stops.

Value range: integer, unlimited. The default value is **0**, indicating that error information is returned immediately.

- **TO { GROUP groupname | NODE ( nodename [, ... ] ) }**

Currently, **TO GROUP** cannot be used. **TO NODE** is used for internal scale-out tools.

## Examples

```
-- Create a foreign table to import data from GDS servers 192.168.0.90 and 192.168.0.91 in TEXT format.
Record errors that occur during data import in err_HR_staffs.
CREATE FOREIGN TABLE foreign_HR_staffs
(
  staff_ID      NUMBER(6) ,
  FIRST_NAME   VARCHAR2(20),
  LAST_NAME    VARCHAR2(25),
  EMAIL        VARCHAR2(25),
  PHONE_NUMBER VARCHAR2(20),
  HIRE_DATE    DATE,
  employment_ID VARCHAR2(10),
  SALARY       NUMBER(8,2),
  COMMISSION_PCT NUMBER(2,2),
  MANAGER_ID   NUMBER(6),
  section_ID   NUMBER(4)
) SERVER gsmpp_server OPTIONS (location 'gsfs://192.168.0.90:5000/* | gsfs://192.168.0.91:5000/*', format
'TEXT', delimiter E'\x08', null '') WITH err_HR_staffs;
-- Create a foreign table to import data from GDS servers 192.168.0.90 and 192.168.0.91 in TEXT format
and record error messages in the import process to the err_HR_staffs table. A maximum of two data
format errors are allowed during the import.
CREATE FOREIGN TABLE foreign_HR_staffs_ft3
(
  staff_ID      NUMBER(6) ,
  FIRST_NAME   VARCHAR2(20),
  LAST_NAME    VARCHAR2(25),
  EMAIL        VARCHAR2(25),
  PHONE_NUMBER VARCHAR2(20),
  HIRE_DATE    DATE,
  employment_ID VARCHAR2(10),
  SALARY       NUMBER(8,2),
  COMMISSION_PCT NUMBER(2,2),
  MANAGER_ID   NUMBER(6),
  section_ID   NUMBER(4)
) SERVER gsmpp_server OPTIONS (location 'gsfs://192.168.0.90:5000/* | gsfs://192.168.0.91:5000/*', format
'TEXT', delimiter E'\x08', null '', reject_limit '2') WITH err_HR_staffs_ft3;

-- Delete the foreign table:
DROP FOREIGN TABLE foreign_HR_staffs;
DROP FOREIGN TABLE foreign_HR_staffs_ft3;
```

## Helpful Links

[ALTER FOREIGN TABLE \(For GDS\), DROP FOREIGN TABLE](#)

# 12.35 CREATE FOREIGN TABLE (for OBS Import and Export)

## Function

**CREATE FOREIGN TABLE** creates an OBS foreign table in the current database for parallel data import and export.

An OBS foreign table can be set to **READ ONLY** or **WRITE ONLY**. The default value is **READ ONLY**. To import data to the cluster, use **READ ONLY** for the foreign table. To export data, use **WRITE ONLY**.

## Precautions

- The foreign table is owned by the user who runs the command.
- The distribution mode of an OBS foreign table does not need to be explicitly specified. The default mode is **ROUNDROBIN**.
- Only **informational constraints** are valid to the OBS foreign table.
- Ensure no Chinese characters are contained in paths used for importing data to or exporting data from OBS.

## Syntax

```
CREATE FOREIGN TABLE [ IF NOT EXISTS ] table_name  
( { column_name type_name [column_constraint ]  
  | LIKE source_table | table_constraint [, ...] } [, ...] )  
SERVER gsmpp_server  
OPTIONS ( { option_name ' value ' } [, ...] )  
[ { WRITE ONLY | READ ONLY } ]  
[ WITH error_table_name | LOG INTO error_table_name ]  
[ PER NODE REJECT LIMIT 'value' ] ;
```

- **column\_constraint** is as follows:  
[CONSTRAINT constraint\_name]  
{PRIMARY KEY | UNIQUE}  
[NOT ENFORCED [ENABLE QUERY OPTIMIZATION | DISABLE QUERY OPTIMIZATION] | ENFORCED]
- **table\_constraint** is as follows:  
[CONSTRAINT constraint\_name]  
{PRIMARY KEY | UNIQUE} (column\_name)  
[NOT ENFORCED [ENABLE QUERY OPTIMIZATION | DISABLE QUERY OPTIMIZATION] | ENFORCED]

## Parameter Overview

**CREATE FOREIGN TABLE** provides multiple parameters, which are classified as follows:

- Mandatory parameters
  - **table\_name**
  - **column\_name**
  - **type\_name**

- **SERVER gsmpp\_server**
- **access\_key**
- **secret\_access\_key**
- **OPTIONS parameters**
  - Data source location parameter in foreign tables: **location**
  - Data format parameters
    - **format**
    - **header** (Only CSV and FIXED formats are supported.)
    - **delimiter**
    - **quote** (Only the CSV format is supported.)
    - **escape** (Only the CSV format is supported.)
    - **null**
    - **noescaping** (Only the TEXT format is supported.)
    - **encoding**
  - Error-tolerance parameters
    - **fill\_missing\_fields**
    - **ignore\_extra\_data**
    - **compatible\_illegal\_chars**
    - **PER NODE REJECT LIMIT 'val...**
    - **LOG INTO error\_table\_name**
    - **WITH error\_table\_name**

## Parameter Description

- **IF NOT EXISTS**

Does not throw an error if a table with the same name exists. A notice is issued in this case.
- **table\_name**

Specifies the name of the foreign table to be created.  
Value range: a string compliant with the naming convention.
- **column\_name**

Specifies the name of a column in the foreign table.  
Value range: a string compliant with the naming convention.
- **type\_name**

Specifies the data type of the column.
- **SERVER gsmpp\_server**

Specifies the server name of the foreign table. In the OBS foreign table, its server **gsmpp\_server** is created by the initial database.

- **OPTIONS ( { option\_name ' value ' } [, ...] )**

Specifies parameters of foreign table data.

- **encrypt**

Specifies whether HTTPS is enabled for data transfer. **on** enables HTTPS and **off** disables it (in this case, HTTP is used). The default value is **off**.

- **access\_key**

Indicates the access key (AK, obtained from the user information on the console) used for the OBS access protocol. When you create a foreign table, its AK value is encrypted and saved to the metadata table of the database.

- **secret\_access\_key:**

Indicates the secret access key (SK, obtained from the user information on the console) used for the OBS access protocol. When you create a foreign table, its SK value is encrypted and saved to the metadata table of the database.

- **chunksize**

Specifies the cache read by each OBS thread on a DN. Its value range is 8 to 512 in the unit of MB. Its default value is **64**.

- **location**

Specifies the data source location of a foreign table. Currently, only URLs are allowed. Multiple URLs are separated using vertical bars (|).

#### NOTE

- The URL of a read-only foreign table (the default permission is read-only) can end with the path prefix or the full path of the target object in the format of **obs://Bucket/Prefix or full path Prefix** indicates the prefix of an object path, for example, **obs://mybucket/tpch/nation/**.
- If the **region** parameter is explicitly specified in **obs://Bucket/Prefix**, the value of **region** will be read. If the **region** parameter is not specified, the value of **defaultRegion** will be read.
- The URL of a writable foreign table does not need to contain a file name. You can specify only one data source location for a foreign table. The directory corresponding to the location must be created before you specify the location.
- URLs specified for a read-only foreign table must be different.

When importing and exporting data, you are advised to use the **location** parameter as follows:

- You are advised to specify a file name for **location** during data import. If you only specify an OBS bucket or directory, all text files in it will be imported. An error message will be reported if the data format is incorrect. If you set fault tolerance, a large amount of data may be imported to the fault-tolerant table.
- Multiple files in an OBS bucket can be imported at the same time. The matched files are imported based on the file name prefix. For example, you can identify and import the following two files after specifying the prefix **mybucket/input\_data/product\_info** in **location**:

```
mybucket/input_data/product_info.0  
mybucket/input_data/product_info.1
```

- If you specify a file name, for example, **1.csv**, then other files (like **1.csv1** or **1.csv22**) starting with **1.csv** in the bucket or directory where **1.csv** resides will be automatically imported. That is, files, such as **1.csv1** and **1.csv22**, are automatically imported.
  - During import, **location** supports multiple OBS paths, which are separated with vertical bars (|).
  - During data export, a directory is generated for **location** by default. If you specify only a file name, the system automatically creates a directory whose name starts with the file name and then generates the file that stores the exported data. The file name is automatically generated by GaussDB(DWS).
  - You can specify one path for **location** only during data export.
- **region**  
(Optional) specifies the value of **regionCode**, region information on the cloud.
- If the **region** parameter is explicitly specified, the value of **region** will be read. If the **region** parameter is not specified, the value of **defaultRegion** will be read.

#### NOTE

Note the following when setting parameters for importing or exporting OBS foreign tables in TEXT or CSV format:

- The **location** parameter is mandatory. The prefixes **gsobs** and **obs** indicate file locations on OBS. The **gsobs** prefix should be followed by *obs url*, *bucket*, and *prefix*. The **obs** prefix should be followed by *bucket* or *prefix*.
  - The data sources of multiple buckets are separated by vertical bars (|), for example, **LOCATION 'obs://bucket1/folder/ | obs://bucket2/'**. The database scans all objects in the specified folders.
- **format**  
Specifies the format of the source data file in a foreign table.  
Valid value: **CSV** and **TEXT**. The default value is **TEXT**. GaussDB(DWS) only supports CSV and TEXT formats.
- **CSV** (comma-separated format):
    - The CSV file can process linefeeds efficiently, but cannot process certain special characters very well.
    - A CSV file is composed of records that are separated as columns by delimiters. Each record shares the same column sequence.
  - **TEXT** (text format):
    - Records are separated as columns by linefeed. The TEXT file can process special characters efficiently, but cannot process linefeeds well.
- **header**  
Specifies whether a file contains a header with the names of each column in the file.



When OBS exports data, this parameter cannot be set to **true**. Use the default value **false**, indicating that the first row of the exported data file is not the header.

When data is imported, if **header** is **on**, the first row of the data file will be identified as the header and ignored. If **header** is **off**, the first row will be identified as a data row.

Valid value: **true**, **on**, **false**, and **off**. The default value is **false** or **off**.

– delimiter

Specifies the column delimiter of data. Use the default delimiter if it is not set. The default delimiter of TEXT is a tab and that of CSV is a comma (,).

 NOTE

- The delimiter of TEXT cannot be `\r` or `\n`.
- A delimiter cannot be the same as the **null** value. The delimiter for the CSV format cannot be same as the **quote** value.
- The delimiter for the TEXT format data cannot contain backslash (\), lowercase letters, digits, or dot (.).
- The data length of a single row should be less than 1 GB. A row that has many columns using long delimiters cannot contain much valid data.
- You are advised to use a multi-character string, such as the combination of the dollar sign (\$), caret (^), and ampersand (&), or invisible characters, such as 0x07, 0x08, and 0x1b as the delimiter.

Value range:

The value of **delimiter** can be a multi-character delimiter whose length is less than or equal to 10 bytes.

– quote

Specifies the quotation mark for the CSV format. The default value is a double quotation mark (").

 NOTE

- The **quote** value cannot be the same as the delimiter or **null** value.
- The **quote** value must be a single-byte character.
- Invisible characters are recommended as **quote** values, such as 0x07, 0x08, and 0x1b.

– escape

Specifies an escape character for a CSV file. The value must be a single-byte character.

The default value is a double quotation mark ("). If the value is the same as the **quote** value, it will be replaced with `\0`.

– null

Specifies the string that represents a null value.

 NOTE

- The **null** value cannot be `\r` or `\n`. The maximum length is 100 characters.
- The **null** value cannot be the same as the delimiter or **quote** value.

Value range:

- The default value is **\N** for the TEXT format.
- The default value for the CSV format is an empty string without quotation marks.
- **noescaping**  
Specifies whether to escape the backslash (\) and its following characters in the TEXT format.

 **NOTE**

**noescaping** is available only for the TEXT format.

Valid value: **true**, **on**, **false**, and **off**. The default value is **false** or **off**.

- **encoding**  
Specifies the encoding of a data file, that is, the encoding used to parse, check, and generate a data file. Its default value is the default **client\_encoding** value of the current database.  
  
Before you import foreign tables, it is recommended that you set **client\_encoding** to the file encoding format, or a format matching the character set of the file. Otherwise, unnecessary parsing and check errors may occur, leading to import errors, rollback, or even invalid data import. Before exporting foreign tables, you are also advised to specify this parameter, because the export result using the default character set may not be what you expect.  
  
If this parameter is not specified when you create a foreign table, a warning message will be displayed on the client.

 **NOTE**

Currently, OBS cannot parse a file using multiple character sets during foreign table import.

Currently, OBS cannot write a file using multiple character sets during foreign table export.

- **fill\_missing\_fields**  
Specifies how to handle the problem that the last column of a row in the source file is lost during data import.  
  
Valid value: **true**, **on**, **false**, and **off**. The default value is **false** or **off**.
  - If this parameter is set to **true** or **on** and the last column of a data row in a source data file is lost, the column will be replaced with **NULL** and no error message will be generated.
  - If this parameter is set to **false** or **off** and the last column of a data row in a source data file is lost, the following error information will be displayed:  

```
missing data for column "tt"
```
- **ignore\_extra\_data**  
Specifies whether to ignore excessive columns when the number of columns in a source data file exceeds that defined in the foreign table. This parameter is available only for data import.  
  
Valid value: **true**, **on**, **false**, and **off**. The default value is **false** or **off**.

- If this parameter is set to **true** or **on** and the number of source data files exceeds the number of foreign table columns, excessive columns will be ignored.
- If this parameter is set to **false** or **off** and the number of source data files exceeds the number of foreign table columns, the following error information will be displayed:  
extra data after last expected column

---

**NOTICE**

If the linefeed at the end of a row is lost and this parameter is set to **true**, data in the next row will be ignored.

---

- **reject\_limit**  
Specifies the maximum number of data format errors allowed during a data import task. If the number of errors does not reach the maximum number, the data import task can still be executed.

---

**NOTICE**

You are advised to replace this syntax with **PER NODE REJECT LIMIT 'value'**.

Examples of data format errors include the following: a column is lost, an extra column exists, a data type is incorrect, and encoding is incorrect. Once a non-data format error occurs, the whole data import process is stopped.

---

Value range: an integer and **unlimited**.

The default value is **0**, indicating that error information is returned immediately.

- **eol**  
Specifies the newline character style of the imported or exported data file.  
Value range: multi-character newline characters within 10 bytes. Common newline characters include **\r** (0x0D), **\n** (0x0A), and **\r\n** (0x0D0A). Special newline characters include **\$** and **#**.

**NOTE**

- The **eol** parameter supports only the TEXT format for data import. For forward compatibility, the **eol** parameter can be set to **0x0D** or **0x0D0A** for data export in the CSV and FIXED formats.
  - The value of the **eol** parameter cannot be the same as that of **DELIMITER** or **NULL**.
  - The **eol** parameter value cannot contain lowercase letters, digits, or dot (.).
- **date\_format**  
Specifies the DATE format for data import. This syntax is available only for READ ONLY foreign tables.

Value range: a valid DATE value. For details, see [Date and Time Processing Functions and Operators](#).

 NOTE

If Oracle is specified as the compatible database, the DATE format is TIMESTAMP. For details, see [timestamp\\_format](#) below.

– time\_format

Specifies the TIME format for data import. This syntax is available only for READ ONLY foreign tables.

Value range: a valid TIME value. Time zones cannot be used. For details, see [Date and Time Processing Functions and Operators](#).

– timestamp\_format

Specifies the TIMESTAMP format for data import. This syntax is available only for READ ONLY foreign tables.

Value range: any valid TIMESTAMP value. Time zones cannot be used. For details, see [Date and Time Processing Functions and Operators](#).

– smalldatetime\_format

Specifies the SMALLDATETIME format for data import. This syntax is available only for READ ONLY foreign tables.

Value range: a valid SMALLDATETIME value. For details, see [Date and Time Processing Functions and Operators](#).

– compatible\_illegal\_chars

Specifies whether to enable fault tolerance on invalid characters during data import. This syntax is available only for READ ONLY foreign tables.

Valid value: **true**, **on**, **false**, and **off**. The default value is **false** or **off**.

- If this parameter is set to **true** or **on**, invalid characters are tolerated and imported to the database after conversion.
- If this parameter is set to **false** or **off** and an error occurs when there are invalid characters, the import will be interrupted.

---

**NOTICE**

On a Windows platform, if OBS reads data files using the TEXT format, 0x1A will be treated as an EOF symbol and a parsing error will occur. It is the implementation constraint of the Windows platform. Since OBS on a Windows platform does not support BINARY read, the data can be read by OBS on a Linux platform.

---

 NOTE

The rule of error tolerance for invalid characters imported is as follows:

(1) \0 is converted to a space.

(2) Other invalid characters are converted to question marks.

(3) If **compatible\_illegal\_chars** is set to **true** or **on**, invalid characters are tolerated. If **NULL**, **DELIMITER**, **QUOTE**, and **ESCAPE** are set to a spaces or question marks, errors like "illegal chars conversion may confuse COPY escape 0x20" will be displayed to prompt users to change parameter values that cause confusion, preventing import errors.

- **READ ONLY**

Specifies whether a foreign table is read-only. This parameter is available only for data import.

- **WRITE ONLY**

Specifies whether a foreign table is write-only. This parameter is available only for data import.

- **WITH error\_table\_name**

Specifies the table where data format errors generated during parallel data import are recorded. You can query the error information table after data is imported to obtain error details. This parameter is available only after **reject\_limit** is set.

 NOTE

To be compatible with postgres open source interfaces, you are advised to replace this syntax with **LOG INTO**. When this parameter is specified, an error table is automatically created.

Value range: a string compliant with the naming convention.

- **LOG INTO error\_table\_name**

Specifies the table where data format errors generated during parallel data import are recorded. You can query the error information table after data is imported to obtain error details.

 NOTE

- This parameter is available only after **PER NODE REJECT LIMIT** is set.
- When this parameter is specified, an error table is automatically created.

Value range: a string compliant with the naming convention.

- **PER NODE REJECT LIMIT 'value'**

Specifies the maximum number of data format errors on each DN during data import. If the number of errors exceeds the specified value on any DN, data import fails, an error is reported, and the system exits data import.

---

**NOTICE**

This syntax specifies the error tolerance of a single node.

Examples of data format errors include the following: a column is lost, an extra column exists, a data type is incorrect, and encoding is incorrect. When a non-data format error occurs, the whole data import process stops.

---

Valid value: an integer and **unlimited**. The default value is **0**, indicating that error information is returned immediately.

- **NOT ENFORCED**

Specifies the constraint to be an informational constraint. This constraint is guaranteed by the user instead of the database.

- **ENFORCED**

The default value is **ENFORCED**. **ENFORCED** is a reserved parameter and is currently not supported.

- **PRIMARY KEY (column\_name)**

Specifies the informational constraint on **column\_name**.

Value range: a string. It must comply with the naming convention, and the value of **column\_name** must exist.

- **ENABLE QUERY OPTIMIZATION**

Optimizes the query plan using an informational constraint.

- **DISABLE QUERY OPTIMIZATION**

Disables the optimization of the query plan using an informational constraint.

## Example

```
-- Create a foreign table to import data in the .txt format from OBS to the row_tbl table.
drop foreign table if exists OBS_ft;
NOTICE: foreign table "obs_ft" does not exist, skipping
DROP FOREIGN TABLE
create foreign table OBS_ft( a int, b int)SERVER gsmpp_server OPTIONS (location 'obs://gaussdbcheck/
obs_ddl/test_case_data/txt_obs_informatonal_test001',format 'text',encoding 'utf8',chunksiz
e '32', encrypt
'off',ACCESS_KEY 'access_key_value_to_be_replaced',SECRET_ACCESS_KEY
'secret_access_key_value_to_be_replaced',delimiter E'\x08') read only;
CREATE FOREIGN TABLE
drop table row_tbl;
DROP TABLE
create table row_tbl( a int, b int);
NOTICE: The 'DISTRIBUTE BY' clause is not specified. Using 'a' as the distribution column by default.
HINT: Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
insert into row_tbl select * from OBS_ft;
INSERT 0 3
```

## Helpful Links

[ALTER FOREIGN TABLE \(For HDFS or OBS\), DROP FOREIGN TABLE](#)

# 12.36 CREATE FOREIGN TABLE (SQL on Hadoop or OBS)

## Function

**CREATE FOREIGN TABLE** (SQL on Hadoop) creates an HDFS foreign table in the current database to access Hadoop structured data stored on HDFS. An HDFS foreign table is read-only. It can only be queried using **SELECT**.

## Syntax

Create an HDFS foreign table.

```
CREATE FOREIGN TABLE [ IF NOT EXISTS ] table_name  
( [ { column_name type_name  
  [ { [CONSTRAINT constraint_name] NULL |  
    [CONSTRAINT constraint_name] NOT NULL |  
    column_constraint [...] } ] |  
  table_constraint [ , ... ] [ , ... ] } )  
SERVER server_name  
OPTIONS ( { option_name ' value ' } [ , ... ] )  
DISTRIBUTE BY { ROUNDROBIN | REPLICATION }  
  
[ PARTITION BY ( column_name ) [ AUTOMAPPED ] ] ;
```

- **column\_constraint** is as follows:  
[CONSTRAINT constraint\_name]  
{PRIMARY KEY | UNIQUE}  
[NOT ENFORCED [ENABLE QUERY OPTIMIZATION | DISABLE QUERY OPTIMIZATION] | ENFORCED]
- **table\_constraint** is as follows:  
[CONSTRAINT constraint\_name]  
{PRIMARY KEY | UNIQUE} (column\_name)  
[NOT ENFORCED [ENABLE QUERY OPTIMIZATION | DISABLE QUERY OPTIMIZATION] | ENFORCED]

## Parameter Description

- **IF NOT EXISTS**  
Does not throw an error if a table with the same name exists. A notice is issued in this case.
- **table\_name**  
Specifies the name of the foreign table to be created.  
Value range: a string. It must comply with the naming convention.
- **column\_name**  
Specifies the name of a column in the foreign table. Columns are separated by commas (,).  
Value range: a string. It must comply with the naming convention.
- **type\_name**  
Specifies the data type of the column.  
Data types supported by tables in ORC format include: SMALLINT, INTEGER, and BIGINT, FLOAT4 (REAL), FLOAT8(DOUBLE PRECISION), DECIMAL[p,(s)] (maximum precision: 38 decimal points), DATE, TIMESTAMP, BOOLEAN, CHAR(n), VARCHAR(n), TEXT(CLOB).  
The data types supported by TXT table are the same as those in row-store tables.
- **constraint\_name**  
Specifies the name of a constraint for the foreign table.
- **{ NULL | NOT NULL }**  
Specifies whether the column allows **NULL**.  
When you create a table, whether the data in HDFS is **NULL** or **NOT NULL** cannot be guaranteed. The consistency of data is guaranteed by users. Users must decide whether the column is **NULL** or **NOT NULL**. (The optimizer optimizes the **NULL/NOT NULL** and generates a better plan.)

- **SERVER server\_name**

Specifies the server name of the foreign table. Users can customize its name.  
Value range: a string indicating an existing server. It must comply with the naming convention.

- **OPTIONS ( { option\_name ' value ' } [, ...] )**

Specifies the following parameters for a foreign table:

- header

Specifies whether a data file contains a table header. **header** is available only for CSV files.

If **header** is **on**, the first row of the data file will be identified as the header and ignored during export. If **header** is **off**, the first row will be identified as a data row.

Value range: **true**, **on**, **false**, and **off**. The default value is **false** or **off**.

- quote

Specifies the quotation mark for the CSV format. The default value is a double quotation mark (").

 **NOTE**

The **quote** value cannot be the same as the **delimiter** or **null** value.

The **quote** value must be a single-byte character.

Invisible characters are recommended as **quote** values, such as 0x07, 0x08, and 0x1b.

- escape

Specifies an escape character for a CSV file. The value must be a single-byte character.

The default value is a double quotation mark ("). If the value is the same as the **quote** value, it will be replaced with **\0**.

- location

Specifies the file path on OBS. This is an OBS foreign table parameter. The data sources of multiple buckets are separated by vertical bars (|), for example, **LOCATION 'obs://bucket1/folder/ | obs://bucket2/'**. The database scans all objects in the specified folders.

- **format**: format of the data source file in the foreign table. ORC, TEXT, and CSV formats are supported.

- **foldername**: directory of the data source file in the foreign table, that is, the corresponding file directory in HDFS.

- **encoding**: encoding of data source files in foreign tables. The default value is **utf8**. This parameter is optional.

- **totalrows**: (Optional) estimated number of rows in a table. This parameter is used only for OBS foreign tables. Because OBS may store many files, it is slow to analyze data. This parameter allows you to set an estimated value so that the optimizer can estimate the table size according to the value. Generally, query efficiency is high when the estimated value is close to the actual value.

- **filenames**: data source files specified in the foreign table. Multiple files are separated by commas (,).



 NOTE

- You are advised to use **foldername** to specify the locations of data sources.
  - An absolute path in **foldername** should be enclosed with slashes (/). Multiple paths are separated by commas (,).
  - When you query a partitioned table, data is pruned based on partition information, and data files that meet the requirement are queried. Pruning involves scanning HDFS directory contents many times. Therefore, do not use columns with low repetition as partition column.
  - An OBS foreign table is not supported.
- delimiter

Specifies the column delimiter of data, and uses the default delimiter if it is not set. The default delimiter of TEXT is a tab.

 NOTE

- A delimiter cannot be \r or \n.
- A delimiter cannot be the same as the null parameter.
- A delimiter cannot contain the following characters:  
`\.abcdefghijklmnopqrstuvwxyz0123456789`
- The data length of a single row should be less than 1 GB. A row that has many columns using long delimiters cannot contain much valid data.
- You are advised to use a multi-character, such as the combination of the dollar sign (\$), caret (^), ampersand (&), or invisible characters, such as 0x07, 0x08, and 0x1b as the delimiter.
- **delimiter** is available only for TEXT and CSV source data files.

Valid value:

The value of **delimiter** can be a multi-character delimiter whose length is less than or equal to 10 bytes.

- null

Specifies the string that represents a null value.

 NOTE

- The null value cannot be \r or \n. The maximum length is 100 characters.
- The **null** parameter cannot be the same as the delimiter.
- **null** is available only for TEXT and CSV source data files.

Valid value:

The default value is \N for the TEXT format.

- noescaping

Specifies in TEXT format, whether to escape the backslash (\) and its following characters.

 NOTE

**noescaping** is available only for TEXT source data files.

Value range: **true**, **on**, **false**, and **off**. The default value is **false** or **off**.

- fill\_missing\_fields

Specifies whether to generate an error message when the last column in a row in the source file is lost during data loading.

Value range: **true**, **on**, **false**, and **off**. The default value is **false** or **off**.

- If this parameter is set to **true** or **on** and the last column of a data row in a data source file is lost, the column is replaced with **NULL** and no error message will be generated.
- If this parameter is set to **false** or **off** and the last column is missing, the following error information will be displayed:  
missing data for column "tt"

**NOTE**

- Because **SELECT COUNT(\*)** does not parse columns in TEXT format, it does not report missing columns.
- **fill\_missing\_fields** is available only for TEXT and CSV source data files.

– **ignore\_extra\_data**

Specifies whether to ignore excessive columns when the number of data source files exceeds the number of foreign table columns. This parameter is available during data import.

Value range: **true**, **on**, **false**, and **off**. The default value is **false** or **off**.

- If this parameter is set to **true** or **on** and the number of data source files exceeds the number of foreign table columns, excessive columns will be ignored.
- If this parameter is set to **false** or **off** and the number of data source files exceeds the number of foreign table columns, the following error information will be displayed:  
extra data after last expected column

**NOTICE**

- If the newline character at the end of the row is lost, setting the parameter to **true** will ignore data in the next row.
- Because **SELECT COUNT(\*)** does not parse columns in TEXT format, it does not report missing columns.
- **ignore\_extra\_data** is available only for TEXT and CSV source data files.

– **date\_format**

Specifies the DATE format for data import. This syntax is available only for READ ONLY foreign tables.

Value range: any valid DATE value. For details, see [Date and Time Processing Functions and Operators](#).

**NOTE**

- If ORACLE is specified as the compatible database, the DATE format is **TIMESTAMP**. For details, see **timestamp\_format** below.
- **date\_format** is available only for TEXT and CSV source data files.

– **time\_format**

Specifies the TIME format for data import. This syntax is available only for READ ONLY foreign tables.

Value range: a valid TIME value. Time zones cannot be used. For details, see [Date and Time Processing Functions and Operators](#).

 NOTE

**time\_format** is available only for TEXT and CSV source data files.

- **timestamp\_format**  
Specifies the TIMESTAMP format for data import. This syntax is available only for READ ONLY foreign tables.  
Value range: any valid TIMESTAMP value. Time zones are not supported. For details, see [Date and Time Processing Functions and Operators](#).

 NOTE

**timestamp\_format** is available only for TEXT and CSV source data files.

- **smalldatetime\_format**  
Specifies the SMALLDATETIME format for data import. This syntax is available only for READ ONLY foreign tables.  
Value range: a valid SMALLDATETIME value. For details, see [Date and Time Processing Functions and Operators](#).

 NOTE

**smalldatetime\_format** is available only for TEXT and CSV source data files.

- **checkencoding**  
Specifies whether to check the character encoding.  
Value range: **low, high** The default value is **low**.

 NOTE

In TEXT format, the rule of error tolerance for invalid characters imported is as follows:

- \0 is converted to a space.
- Other invalid characters are converted to question marks.
- Setting **checkencoding** to **low** enables invalid characters toleration. If **NULL** and **DELIMITER** are set to spaces or question marks (?), errors like "illegal chars conversion may confuse null 0x20" will be displayed, prompting you to modify parameters that may cause confusion and preventing importing errors.

In ORC format, the rule of error tolerance for invalid characters imported is as follows:

- If **checkencoding** is **low**, an imported field containing invalid characters will be replaced with a quotation mark string of the same length.
- If **checkencoding** is **high**, data import stops when an invalid character is detected.

**Table 12-15** Support for TEXT, CSV, and ORC formats

| Parameter Name | OBS  |     |     | HDFS |     |     |
|----------------|------|-----|-----|------|-----|-----|
|                | TEXT | CSV | ORC | TEXT | CSV | ORC |
| -              | TEXT | CSV | ORC | TEXT | CSV | ORC |

| Parameter Name      | OBS           |               |               | HDFS          |               |               |
|---------------------|---------------|---------------|---------------|---------------|---------------|---------------|
|                     |               |               |               |               |               |               |
| location            | Supported     | Supported     | Supported     | Not supported | Not supported | Not supported |
| format              | Supported     | Supported     | Supported     | Supported     | Supported     | Supported     |
| header              | Not supported | Supported     | Not supported | Not supported | Supported     | Not supported |
| delimiter           | Supported     | Supported     | Not supported | Supported     | Supported     | Not supported |
| quote               | Not supported | Supported     | Not supported | Not supported | Supported     | Not supported |
| escape              | Not supported | Supported     | Not supported | Not supported | Supported     | Not supported |
| null                | Supported     | Supported     | Not supported | Supported     | Supported     | Not supported |
| noescaping          | Supported     | Not supported | Not supported | Supported     | Not supported | Not supported |
| encoding            | Supported     | Supported     | Supported     | Supported     | Supported     | Supported     |
| fill_missing_fields | Supported     | Supported     | Not supported | Supported     | Supported     | Not supported |
| ignore_extra_data   | Supported     | Supported     | Not supported | Supported     | Supported     | Not supported |
| date_format         | Supported     | Supported     | Not supported | Supported     | Supported     | Not supported |
| time_format         | Supported     | Supported     | Not supported | Supported     | Supported     | Not supported |
| timestamp_format    | Supported     | Supported     | Not supported | Supported     | Supported     | Not supported |

| Parameter Name       | OBS           |               |               | HDFS          |               |               |
|----------------------|---------------|---------------|---------------|---------------|---------------|---------------|
|                      |               |               |               |               |               |               |
| smalldatetime_format | Supported     | Supported     | Not supported | Supported     | Supported     | Not supported |
| chunksize            | Supported     | Supported     | Not supported | Supported     | Supported     | Not supported |
| filenames            | Not supported | Not supported | Not supported | Supported     | Supported     | Supported     |
| foldername           | Supported     | Supported     | Supported     | Supported     | Supported     | Supported     |
| checkencoding        | Supported     | Supported     | Supported     | Supported     | Supported     | Supported     |
| totalrows            | Supported     | Supported     | Supported     | Not supported | Not supported | Not supported |

- **DISTRIBUTE BY ROUNDROBIN**

Specifies **ROUNDROBIN** as the distribution mode for the HDFS foreign table.

- **DISTRIBUTE BY REPLICATION**

Specifies **REPLICATION** as the distribution mode for the HDFS foreign table.

- **PARTITION BY ( column\_name ) AUTOMAPPED**

**column\_name** specifies the partition column. **AUTOMAPPED** means the partition column specified by the HDFS partitioned foreign table is automatically mapped with the partition directory information in HDFS. The prerequisite is that the sequences of partition columns specified in the HDFS foreign table and in the directory are the same.

 **NOTE**

Partitioned tables can be used as foreign tables for HDFS, but not for OBS.

- **CONSTRAINT constraint\_name**

Specifies the name of informational constraint of the foreign table.

Value range: a string. It must comply with the naming convention.

- **PRIMARY KEY**

The primary key constraint specifies that one or more columns of a table must contain unique (non-duplicate) and non-null values. Only one primary key can be specified for a table.

- **UNIQUE**

Specifies that a group of one or more columns of a table must contain unique values. For the purpose of a unique constraint, **NULL** is not considered equal.

- **NOT ENFORCED**

Specifies the constraint to be an informational constraint. This constraint is guaranteed by the user instead of the database.

- **ENFORCED**  
The default value is **ENFORCED**. **ENFORCED** is a reserved parameter and is currently not supported.
- **PRIMARY KEY (column\_name)**  
Specifies the informational constraint on **column\_name**.  
Value range: a string. It must comply with the naming convention, and the value of **column\_name** must exist.
- **ENABLE QUERY OPTIMIZATION**  
Optimizes an execution plan using an informational constraint.
- **DISABLE QUERY OPTIMIZATION**  
Disables the optimization of an execution plan using an informational constraint.

## Informational Constraint

In GaussDB(DWS), data is stored in HDFS. GaussDB(DWS) does not support writing data to HDFS. It is the user's responsibility to ensure enforcement of constraints. If the source data is compliant with certain informational constraint requirements, the query of such data can achieve higher efficiency. HDFS foreign tables do not support indexes. Informational constraints are used to optimize query plans.

The constraints of creating informational constraints for an HDFS foreign table are as follows:

- You can create an informational constraint only if the values in a NOT NULL column in your table are unique. Otherwise, the query result will be different from expected.
- Currently, the informational constraints of GaussDB(DWS) support only PRIMARY KEY and UNIQUE constraints.
- The informational constraints of GaussDB(DWS) support only the NOT ENFORCED attribute.
- Both an HDFS foreign table and an HDFS partitioned foreign table supports informational constraint, which is also established in a partitioned column).
- UNIQUE informational constraints can be created for multiple columns in a table, but only one PRIMARY KEY constraint can be created in a table.
- Multiple informational constraints can be established in a column of a table (because the function that establishing a column or multiple constraints in a column is the same.) Therefore, you are not advised to set up multiple informational constraints in a column, and only one Primary Key type can be set up.
- Multi-column combination constraints are not supported.

## Examples

Example 1: In HDFS, import the TPC-H benchmark test tables **part** and **region** using Hive. The path of the **part** table is **/user/hive/warehouse/partition.db/**

**part\_4**, and that of the **region** table is **/user/hive/warehouse/mppdb.db/region\_orc11\_64stripe/**.

- Establish HDFS\_Server, with HDFS\_FDWS or DFS\_FDWS as the foreign data wrapper.

```
-- Create HDFS_Server:  
CREATE SERVER hdfs_server FOREIGN DATA WRAPPER HDFS_FDWS OPTIONS (address  
'10.10.0.100:25000,10.10.0.101:25000',hdfscfgpath '/opt/hadoop_client/HDFS/hadoop/etc/  
hadoop',type'HDFS');
```

#### NOTE

The IP addresses and port numbers of HDFS NameNodes are specified in **OPTIONS**. **10.10.0.100:25000,10.10.0.101:25000** indicates the IP addresses and port numbers of the primary and standby HDFS NameNodes. It is the recommended format. Two groups of parameter values are separated by commas (,). Take '10.10.0.100:25000' as an example. In this example, the IP address is 10.10.0.100, and the port number is 25000.

- Create an HDFS foreign table.

```
-- Create an HDFS foreign table that does not contain any partition columns. The HDFS server  
associated with the table is hdfs_server, the corresponding file format of region on the HDFS server is  
'orc', and the file directory in the HDFS file system is /user/hive/warehouse/mppdb.db/  
region_orc11_64stripe/.
```

```
CREATE FOREIGN TABLE region  
(  
  R_REGIONKEY INT4,  
  R_NAME TEXT,  
  R_COMMENT TEXT  
)  
SERVER  
  hdfs_server  
OPTIONS  
(  
  FORMAT 'orc',  
  encoding 'utf8',  
  FOLDERNAME '/user/hive/warehouse/mppdb.db/region_orc11_64stripe/'  
)  
DISTRIBUTE BY  
  roundrobin;
```

```
-- Create an HDFS foreign table that contains partition columns.
```

```
CREATE FOREIGN TABLE part  
(  
  p_partkey int,  
  p_name text,  
  p_mfgr text,  
  p_brand text,  
  p_type text,  
  p_size int,  
  p_container text,  
  p_retailprice float8,  
  p_comment text  
)  
SERVER  
  hdfs_server  
OPTIONS  
(  
  FORMAT 'orc',  
  encoding 'utf8',  
  FOLDERNAME '/user/hive/warehouse/partition.db/part_4'  
)  
DISTRIBUTE BY  
  roundrobin  
PARTITION BY  
  (p_mfgr) AUTOMAPPED;
```

 NOTE

GaussDB(DWS) allows you to specify files using the keyword **filenames** or **foldername**. The latter is recommended. The key word **distribute** specifies the storage distribution mode of the region table.

- View the created foreign table.

```
-- View the foreign table:
SELECT * FROM pg_foreign_table WHERE ftrelid='region'::regclass;
ftrelid | ftserver | ftwriteonly |          ftoptions
-----+-----+-----+-----
16510 | 16509 | f          | {format=orc,foldername=/user/hive/warehouse/mppdb.db/
region_orc11_64stripe/}
(1 row)

select * from pg_foreign_table where ftrelid='part'::regclass;
ftrelid | ftserver | ftwriteonly |          ftoptions
-----+-----+-----+-----
16513 | 16509 | f          | {format=orc,foldername=/user/hive/warehouse/partition.db/part_4}
(1 row)
```

- Modify and delete the foreign table.

```
-- Modify a foreign table:
ALTER FOREIGN TABLE region ALTER r_name TYPE TEXT;
ALTER FOREIGN TABLE
ALTER FOREIGN TABLE region ALTER r_name SET NOT NULL;
ALTER FOREIGN TABLE
-- Delete the foreign table:
DROP FOREIGN TABLE region;
DROP FOREIGN TABLE
```

Example 2: Operations on an HDFS foreign table that includes informational constraints

```
-- Create an HDFS foreign table with informational constraints
CREATE FOREIGN TABLE region (
R_REGIONKEY int,
R_NAME TEXT,
R_COMMENT TEXT
, primary key (R_REGIONKEY) not enforced)
SERVER hdfs_server
OPTIONS(format 'orc',
encoding 'utf8',
foldername '/user/hive/warehouse/mppdb.db/region_orc11_64stripe')
DISTRIBUTE BY roundrobin;

-- Check whether the region table has an informational constraint index:
SELECT relname,relhasindex FROM pg_class WHERE oid='region'::regclass;
relname | relhasindex
-----+-----
region | f
(1 row)

SELECT conname, contype, consoft, conopt, conindid, conkey FROM pg_constraint WHERE conname
='region_pkey';
conname | contype | consoft | conopt | conindid | conkey
-----+-----+-----+-----+-----+-----
region_pkey | p | t | t | 0 | {}
(1 row)

-- Delete the informational constraint:
ALTER FOREIGN TABLE region DROP CONSTRAINT region_pkey RESTRICT;

SELECT conname, contype, consoft, conindid, conkey FROM pg_constraint WHERE conname ='region_pkey';
conname | contype | consoft | conindid | conkey
-----+-----+-----+-----+-----
(0 rows)
```



```
-- Add a unique informational constraint for the foreign table:
ALTER FOREIGN TABLE region ADD CONSTRAINT constr_unique UNIQUE(R_REGIONKEY) NOT ENFORCED;

-- Delete the informational constraint:
ALTER FOREIGN TABLE region DROP CONSTRAINT constr_unique RESTRICT;

SELECT conname, contype, consoft, conindid, conkey FROM pg_constraint WHERE conname
='constr_unique';
conname | contype | consoft | conindid | conkey
-----+-----+-----+-----+-----
(0 rows)

-- Add a unique informational constraint for the foreign table:
ALTER FOREIGN TABLE region ADD CONSTRAINT constr_unique UNIQUE(R_REGIONKEY) NOT ENFORCED
disable query optimization;

SELECT relname,relhasindex FROM pg_class WHERE oid='region'::regclass;
relname      | relhasindex
-----+-----
region       | f
(1 row)

-- Delete the informational constraint:
ALTER FOREIGN TABLE region DROP CONSTRAINT constr_unique CASCADE;

-- Delete the region table:
DROP FOREIGN TABLE region;

-- Delete the hdfs_server server:
DROP SERVER hdfs_server;
```

Example 3: Read data in OBS through a foreign table.

1. Create **obs\_server**, with **DFS\_FDW** as the foreign data wrapper.

```
-- Create obs_server:
CREATE SERVER obs_server FOREIGN DATA WRAPPER DFS_FDW OPTIONS (
ADDRESS 'obs.abc.com',
ACCESS_KEY 'xxxxxxxxx',
SECRET_ACCESS_KEY 'yyyyyyyyyyyyyy',
TYPE 'OBS'
);
```

#### NOTE

- **ADDRESS** indicates the IP address or domain name of OBS. Replace the value as needed. You can find the domain name by searching for the value of **regionCode** in the **region\_map** file.
  - **ACCESS\_KEY** and **SECRET\_ACCESS\_KEY** are access keys for the cloud account system. Replace the values as needed.
  - **TYPE** indicates the server type. Retain the value **OBS**.
2. Create an OBS foreign table.

```
--- Create the customer_address foreign table that does not contain partition columns. obs_server is the associated OBS server. Files on this server are in .orc format and stored in the user/hive/warehouse/mppdb.db/region_orc11_64stripe1/ directory.
```

```
CREATE FOREIGN TABLE customer_address
(
ca_address_sk      integer      not null,
ca_address_id     char(16)      not null,
ca_street_number  char(10)
ca_street_name    varchar(60)
ca_street_type    char(15)
ca_suite_number   char(10)
ca_city           varchar(60)
ca_county         varchar(30)
ca_state          char(2)
ca_zip           char(10)
ca_country        varchar(20)
```

```
ca_gmt_offset      decimal(36,33)      ,
ca_location_type   char(20)
)
SERVER obs_server OPTIONS (
  FOLDERNAME '/user/hive/warehouse/mppdb.db/region_orc11_64stripe1/',
  FORMAT 'ORC',
  ENCODING 'utf8',
  TOTALROWS '20'
)
DISTRIBUTE BY roundrobin;
```

3. Query data stored in OBS using a foreign table.

```
-- View the foreign table:
SELECT COUNT(*) FROM customer_address;
count
-----
      20
(1 row)
```

4. Delete the foreign table.

```
-- Delete the foreign table:
DROP FOREIGN TABLE customer_address;
DROP FOREIGN TABLE
```

## Helpful Links

[ALTER FOREIGN TABLE \(For HDFS or OBS\), DROP FOREIGN TABLE](#)

# 12.37 CREATE FUNCTION

## Function

**CREATE FUNCTION** creates a function.

## Precautions

- The precision values (if any) of the parameters or return values of a function are not checked.
- When creating a function, you are advised to explicitly specify the schemas of tables in the function definition. Otherwise, the function may fail to be executed.
- **current\_schema** and **search\_path** specified by **SET** during function creation are invalid. **search\_path** and **current\_schema** before and after function execution should be the same.
- If a function has output parameters, the **SELECT** statement uses the default values of the output parameters when calling the function. When the **CALL** statement calls the function, it requires that the output parameter values are adapted to Oracle. When the **CALL** statement calls an overloaded **PACKAGE** function, it can use the default values of the output parameters. For details, see examples in [CALL](#).
- Only the functions compatible with PostgreSQL or those with the **PACKAGE** attribute can be overloaded. After **REPLACE** is specified, a new function is created instead of replacing a function if the number of parameters, parameter type, or return value is different.
- You can use the **SELECT** statement to specify different parameters using identical functions, but cannot use the **CALL** statement to call identical

functions without the **PACKAGE** attribute because **CALL** aligns with Oracle syntax.

- When you create a function, you cannot insert other agg functions out of the avg function or other functions.
- In non-logical cluster mode, return values, parameters, and variables cannot be set to the tables of the Node Groups that are not installed in the system by default. The internal statements of SQL functions cannot be executed on such tables.
- In logical cluster mode, if return values and parameters of the function are user tables, all the tables must be in the same logical cluster. If the function body involves operations on multiple logical cluster tables, the function cannot be set to **IMMUTABLE** or **SHIPPABLE**, preventing the function from being pushed down to a DN.
- In logical cluster mode, the parameters and return values of the function cannot use the **%type** to reference a table column type. Otherwise, the function will fail to be created.
- By default, the permissions to execute new functions are granted to **PUBLIC**. For details, see **GRANT**. You can revoke the default execution permissions from **PUBLIC** and grant them to other users as needed. To avoid the time window during which new functions can be accessed by all users, create functions in transactions and set function execution permissions.

## Syntax

- Syntax (compatible with PostgreSQL) for creating a user-defined function:

```
CREATE [ OR REPLACE ] FUNCTION function_name
( [ { argname [ argmode ] argtype [ { DEFAULT | := | = } expression ] } [, ...] ] )
[ RETURNS rettype [ DETERMINISTIC ] | RETURNS TABLE ( { column_name column_type }
[, ...] ) ]
LANGUAGE lang_name
[
  {IMMUTABLE | STABLE | VOLATILE }
  | {SHIPPABLE | NOT SHIPPABLE}
  | WINDOW
  | [ NOT ] LEAKPROOF
  | {CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT }
  | {[ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER | AUTHID DEFINER |
AUTHID CURRENT_USER}
  | {fenced | not fenced}
  | {PACKAGE}


  | COST execution_cost
  | ROWS result_rows
  | SET configuration_parameter { {TO | =} value | FROM CURRENT } }
][...]
{
  AS 'definition'
  | AS 'obj_file', 'link_symbol'
}
```

- Oracle syntax of creating a customized function:

```
CREATE [ OR REPLACE ] FUNCTION function_name
( [ { argname [ argmode ] argtype [ { DEFAULT | := | = } expression ] } [, ...] ] )
RETURN rettype [ DETERMINISTIC ]
[
  {IMMUTABLE | STABLE | VOLATILE }
  | {SHIPPABLE | NOT SHIPPABLE}
  | {PACKAGE}
  | {FENCED | NOT FENCED}
  | [ NOT ] LEAKPROOF
  | {CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT }
```

```
    | {[ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER |  
AUTHID DEFINER | AUTHID CURRENT_USER  
}  
    | COST execution_cost  
    | ROWS result_rows  
    | SET configuration_parameter { {TO | =} value | FROM CURRENT  
  
][...]  
  
{  
  IS | AS  
} plsql_body  
/
```

## Parameter Description

- **function\_name**  
Indicates the name of the function to create (optionally schema-qualified).  
Value range: a string. It must comply with the naming convention.
- **argname**  
Indicates the name of a function parameter.  
Value range: a string. It must comply with the naming convention.
- **argmode**  
Indicates the mode of a parameter.  
Value range: **IN**, **OUT**, **IN OUT**, **INOUT**, and **VARIADIC**. The default value is **IN**. Only the parameter of **OUT** mode can be followed by **VARIADIC**. The parameters of **OUT** and **INOUT** cannot be used in function definition of **RETURNS TABLE**.  
 **NOTE**  
**VARIADIC** specifies parameters of array types.
- **argtype**  
Indicates the data types of the function's parameters.
- **expression**  
Indicates the default expression of a parameter.
- **rettype**  
Indicates the return data type.  
When there is **OUT** or **IN OUT** parameter, the **RETURNS** clause can be omitted. If the clause exists, it must be the same as the result type indicated by the output parameter. If there are multiple output parameters, the value is **RECORD**. Otherwise, the value is the same as the type of a single output parameter.  
The **SETOF** modifier indicates that the function will return a set of items, rather than a single item.
- **DETERMINISTIC**  
The adaptation oracle SQL syntax. You are not advised to use it.
- **column\_name**  
Specifies the column name.
- **column\_type**  
Specifies the column type.

- **definition**  
Specifies a string constant defining the function; the meaning depends on the language. It can be an internal function name, a path pointing to a target file, a SQL query, or text in a procedural language.
- **LANGUAGE lang\_name**  
Indicates the name of the language that is used to implement the function. It can be **SQL**, **internal**, or the name of user-defined process language. To ensure downward compatibility, the name can use single quotation marks. Contents in single quotation marks must be capitalized.
- **WINDOW**  
Indicates that the function is a window function. The **WINDOW** attribute cannot be changed when the function definition is replaced.

---

**NOTICE**

For a user-defined window function, the value of **LANGUAGE** can only be **internal**, and the referenced internal function must be a window function.

- 
- **IMMUTABLE**  
Indicates that the function always returns the same result if the parameter values are the same.
  - **STABLE**  
Indicates that the function cannot modify the database, and that within a single table scan it will consistently return the same result for the same parameter values, but that its result varies by SQL statements.
  - **VOLATILE**  
Indicates that the function value can change even within a single table scan, so no optimizations can be made.
  - **SHIPPABLE**  
**NOT SHIPPABLE**  
Indicates whether the function can be pushed down to DN for execution.
    - Functions of the **IMMUTABLE** type can always be pushed down to the DNs.
    - Functions of the **STABLE** or **VOLATILE** type can be pushed down to DNs only if their attribute is **SHIPPABLE**.
  - **PACKAGE**  
Indicates whether the function can be overloaded. PostgreSQL-style functions can be overloaded, and this parameter is designed for Oracle-style functions.
    - All **PACKAGE** and non-**PACKAGE** functions cannot be overloaded or replaced.
    - **PACKAGE** functions do not support parameters of the **VARIADIC** type.
    - The **PACKAGE** attribute of functions cannot be modified.
  - **LEAKPROOF**  
Indicates that the function has no side effects. **LEAKPROOF** can be set only by the system administrator.

- **CALLED ON NULL INPUT**  
Declares that some parameters of the function can be invoked in normal mode if the parameter values are **NULL**. This parameter can be omitted.
- **RETURNS NULL ON NULL INPUT**  
**STRICT**  
Indicates that the function always returns **NULL** whenever any of its parameters are **NULL**. If this parameter is specified, the function is not executed when there are **NULL** parameters; instead a **NULL** result is returned automatically.  
The usage of **RETURNS NULL ON NULL INPUT** is the same as that of **STRICT**.
- **EXTERNAL**  
The keyword **EXTERNAL** is allowed for SQL conformance, but it is optional since, unlike in SQL, this feature applies to all functions not only external ones.
- **SECURITY INVOKER**  
**AUTHID CURRENT\_USER**  
Indicates that the function is to be executed with the permissions of the user that calls it. This parameter can be omitted.  
**SECURITY INVOKER** and **AUTHID CURRENT\_USER** have the same functions.
- **SECURITY DEFINER**  
**AUTHID DEFINER**  
Specifies that the function is to be executed with the permissions of the user that created it.  
The usage of **AUTHID DEFINER** is the same as that of **SECURITY DEFINER**.
- **FENCED**  
**NOT FENCED**  
(Effective only for C functions) Specifies whether functions are executed in fenced mode. In **NOT FENCED** mode, a function is executed in a CN or DN process. In **FENCED** mode, a function is executed in a new fork process, which does not affect CN or DN processes.  
Application scenarios:
  - Develop or debug a function in **FENCED** mode and execute it in **NOT FENCED** mode. This reduces the cost of the fork process and communication.
  - Perform complex OS operations, such as open a file, process signals and threads, in **FENCED** mode so that GaussDB(DWS) running is not affected.
  - The default value is **FENCED**.
- **COST execution\_cost**  
A positive number giving the estimated execution cost for the function.  
The unit of **execution\_cost** is **cpu\_operator\_cost**.  
Value range: A positive number.
- **ROWS result\_rows**  
Estimates the number of rows returned by the function. This is only allowed when the function is declared to return a set.

Value range: A positive number. The default is 1000 rows.

- **configuration\_parameter**
  - **value**

Sets a specified database session parameter to a specified value. If the value is **DEFAULT** or **RESET**, the default setting is used in the new session. **OFF** closes the setting.

Value range: a string

    - **DEFAULT**
    - **OFF**
    - **RESET**

Specifies the default value.
  - **from current**

Uses the value of **configuration\_parameter** of the current session.
- **obj\_file, link\_symbol**

(Used for C functions) Specifies the absolute path of the dynamic library using **obj\_file** and the link symbol (function name in C programming language) of the function using **link\_symbol**.
- **plsql\_body**

Indicates the PL/SQL stored procedure body.

---

**NOTICE**

When the function is creating users, the log will record unencrypted passwords. You are not advised to do it.

---

## Examples

```
-- Define the function as SQL query:
CREATE FUNCTION func_add_sql(integer, integer) RETURNS integer
AS 'select $1 + $2;'
LANGUAGE SQL
IMMUTABLE
RETURNS NULL ON NULL INPUT;

-- Add an integer by parameter name using PL/pgSQL:
CREATE OR REPLACE FUNCTION func_increment_plsql(i integer) RETURNS integer AS $$
BEGIN
    RETURN i + 1;
END;
$$ LANGUAGE plpgsql;

--Return the RECORD type.
CREATE OR REPLACE FUNCTION compute(i int, out result_1 bigint, out result_2 bigint)
returns SETOF RECORD
as $$
begin
    result_1 = i + 1;
    result_2 = i * 10;
return next;
end;
$$language plpgsql;
```

```
-- Return a record containing multiple output parameters:
CREATE FUNCTION func_dup_sql(in int, out f1 int, out f2 text)
AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
LANGUAGE SQL;

SELECT * FROM func_dup_sql(42);

-- Compute the sum of two integers with returning the result (if the input is null, the returned result is null):
CREATE FUNCTION func_add_sql2(num1 integer, num2 integer) RETURN integer
AS
BEGIN
RETURN num1 + num2;
END;
/

-- Create an overloaded function with the PACKAGE attribute:
create or replace function package_func_overload(col int, col2 int)
return integer package
as
declare
col_type text;
begin
col := 122;
dbms_output.put_line('two int parameters ' || col2);
return 0;
end;
/

create or replace function package_func_overload(col int, col2 smallint)
return integer package
as
declare
col_type text;
begin
col := 122;
dbms_output.put_line('two smallint parameters ' || col2);
return 0;
end;
/

-- Alter the execution rule of function add to IMMUTABLE (that is, the same result is returned if the
parameter remains unchanged):
ALTER FUNCTION func_add_sql2(INTEGER, INTEGER) IMMUTABLE;

-- Alter the name of function add to add_two_number:
ALTER FUNCTION func_add_sql2(INTEGER, INTEGER) RENAME TO add_two_number;

-- Change the owner of the function add to dbadmin:
ALTER FUNCTION add_two_number(INTEGER, INTEGER) OWNER TO dbadmin;

-- Delete the function:
DROP FUNCTION add_two_number;
DROP FUNCTION func_increment_sql;
DROP FUNCTION func_dup_sql;
DROP FUNCTION func_increment_plsql;
DROP FUNCTION func_add_sql;
```

## Helpful Links

[ALTER FUNCTION, DROP FUNCTION](#)

# 12.38 CREATE GROUP

## Function

**CREATE GROUP** creates a user group.



## Precautions

**CREATE GROUP** is an alias for **CREATE ROLE**, and it is not a standard SQL command and not recommended. Users can use **CREATE ROLE** directly.

## Syntax

```
CREATE GROUP group_name [ [ WITH ] option [ ... ] ]  
    [ ENCRYPTED | UNENCRYPTED ] { PASSWORD | IDENTIFIED BY } { 'password' | DISABLE };
```

The syntax of optional **action** clause is as follows:

```
where option can be:  
{SYSADMIN | NOSYSADMIN}  
| {AUDITADMIN | NOAUDITADMIN}  
| {CREATEDB | NOCREATEDB}  
| {USEFT | NOUSEFT}  
| {CREATEROLE | NOCREATEROLE}  
| {INHERIT | NOINHERIT}  
| {LOGIN | NOLOGIN}  
| {REPLICATION | NOREPLICATION}  
| {INDEPENDENT | NOINDEPENDENT}  
| {VCADMIN | NOVCADMIN}  
| CONNECTION LIMIT connlimit  
| VALID BEGIN 'timestamp'  
| VALID UNTIL 'timestamp'  
| RESOURCE POOL 'respool'  
| USER GROUP 'groupuser'  
| PERM SPACE 'spacelimit'  
| NODE GROUP logic_group_name  
| IN ROLE role_name [, ...]  
| IN GROUP role_name [, ...]  
| ROLE role_name [, ...]  
| ADMIN role_name [, ...]  
| USER role_name [, ...]  
| SYSID uid  
| DEFAULT TABLESPACE tablespace_name  
| PROFILE DEFAULT  
| PROFILE profile_name  
| PGUSER
```

## Parameter Description

See [Parameter Description](#) in **CREATE ROLE**.

## Helpful Links

[ALTER GROUP](#), [DROP GROUP](#), [CREATE ROLE](#)

# 12.39 CREATE INDEX

## Function

**CREATE INDEX-bak** defines a new index.

Indexes are primarily used to enhance database performance (though inappropriate use can result in slower database performance). You are advised to create indexes on:

- Columns that are often queried

- Join conditions. For a query on joined columns, you are advised to create a composite index on the columns, for example, **select \* from t1 join t2 on t1.a=t2.a and t1.b=t2.b**. You can create a composite index on the **a** and **b** columns of table **t1**.
- Columns having filter criteria (especially scope criteria) of a **where** clause
- Columns that appear after **order by**, **group by**, and **distinct**.

The partitioned table does not support concurrent index creation, partial index creation, and **NULL FIRST**.

## Precautions

- Indexes consume storage and computing resources. Creating too many indexes has negative impact on database performance (especially the performance of data import. Therefore, you are advised to import the data before creating indexes). Create indexes only when they are necessary.
- All functions and operators used in an index definition must be immutable, that is, their results must depend only on their arguments and never on any outside influence (such as the contents of another table or the current time). This restriction ensures that the behavior of the index is well-defined. To use a user-defined function in an index expression or **WHERE** clause, remember to mark the function **immutable** when you create it.
- A unique index created on a partitioned table must include a partition column and all the partition keys.
- Column-store and HDFS tables support B-tree and psort indexes. If the two indexes are used, you cannot create expression, partial, and unique indexes.
- Column-store tables support GIN indexes, rather than partial indexes and unique indexes. If GIN indexes are used, you can create expression indexes. However, an expression in this situation cannot contain empty splitters, empty columns, or multiple columns.
- Multiple indexes can be created in an HDFS table. The total number of columns involved in an index cannot exceed 16.

## Syntax

- Create an index on a table.

```
CREATE [ UNIQUE ] INDEX [ [ schema_name. ] index_name ] ON table_name [ USING method ]
( ( { column_name | ( expression ) } [ COLLATE collation ] [ opclass ] [ ASC | DESC ] [ NULLS
{ FIRST | LAST } ] }, ... )
[ WITH ( { storage_parameter = value } [, ... ] ) ]
[ WHERE predicate ];
```
- Create an index for a partitioned table.

```
CREATE [ UNIQUE ] INDEX [ [ schema_name. ] index_name ] ON table_name [ USING method ]
( ( { column_name | ( expression ) } [ COLLATE collation ] [ opclass ] [ ASC | DESC ] [ NULLS
LAST } ] }, ... )
LOCAL [ ( { PARTITION index_partition_name } [, ... ] ) ]
[ WITH ( { storage_parameter = value } [, ... ] ) ]
;
```

## Parameter Description

- **UNIQUE**

Causes the system to check for duplicate values in the table when the index is created (if data exists) and each time data is added. Attempts to insert or update data which would result in duplicate entries will generate an error.

Currently, only B-tree in a row-store table supports **UNIQUE**.

- **schema\_name**

Name of the schema where the index to be created is located. The specified schema name must be the same as the schema of the table.

- **index\_name**

Specifies the name of the index to be created. The schema of the index is the same as that of the table.

Value range: A string. It must comply with the naming convention rule.

- **table\_name**

Specifies the name of the table to be indexed (optionally schema-qualified).

Value range: An existing table name.

- **USING method**

Specifies the name of the index method to be used.

Valid value:

- **btree**: The B-tree index uses a structure that is similar to the B+ tree structure to store data key values, facilitating index search. **btree** supports comparison queries with ranges specified.
- **gin**: GIN indexes are reverse indexes and can process values that contain multiple keys (for example, arrays).
- **gist**: GiST indexes are suitable for the set data type and multidimensional data types, such as geometric and geographic data types.
- **Psort**: psort index. It is used to perform partial sort on column-store tables.

Row-column tables support **btree** (default), **gin**, and **gist**. Column-store tables support **Psort** (default), **btree**, and **gin**.

- **column\_name**

Specifies the name of a column of the table.

Multiple columns can be specified if the index method supports multi-column indexes. A maximum of 32 columns can be specified.

- **expression**

Specifies an expression based on one or more columns of the table. The expression usually must be written with surrounding parentheses, as shown in the syntax. However, the parentheses can be omitted if the expression has the form of a function call.

Expression can be used to obtain fast access to data based on some transformation of the basic data. For example, an index computed on `upper(col)` would allow the clause `WHERE upper(col) = 'JIM'` to use an index.

If an expression contains **IS NULL**, the index for this expression is invalid. In this case, you are advised to create a partial index.

- **COLLATE collation**

Assigns a collation to the column (which must be of a collatable data type). If no collation is specified, the default collation is used.

- **opclass**

Specifies the name of an operator class. Specifies an operator class for each column of an index. The operator class identifies the operators to be used by the index for that column. For example, a B-tree index on the type `int4` would use the **int4\_ops** class; this operator class includes comparison functions for values of type `int4`. In practice, the default operator class for the column's data type is sufficient. The operator class applies to data with multiple sorts. For example, we might want to sort a complex-number data type either by absolute value or by real part. We could do this by defining two operator classes for the data type and then selecting the proper class when making an index.
- **ASC**

Indicates ascending sort order (default). This option is supported only by row storage.
- **DESC**

Indicates descending sort order. This option is supported only by row storage.
- **NULLS FIRST**

Specifies that nulls sort before not-null values. This is the default when **DESC** is specified.
- **NULLS LAST**

Specifies that nulls sort after not-null values. This is the default when **DESC** is not specified.
- **WITH ( {storage\_parameter = value} [, ... ] )**

Specifies the name of an index-method-specific storage parameter.  
Valid value:  
Only the GIN index supports the **FASTUPDATE** and **GIN\_PENDING\_LIST\_LIMIT** parameters. The indexes other than GIN and psort support the **FILLFACTOR** parameter.

  - **FILLFACTOR**

The fillfactor for an index is a percentage between 10 and 100.  
Value range: 10–100
  - **FASTUPDATE**

Specifies whether fast update is enabled for the GIN index.  
Valid value: **ON** and **OFF**  
Default: **ON**
  - **GIN\_PENDING\_LIST\_LIMIT**

Specifies the maximum capacity of the pending list of the GIN index when fast update is enabled for the GIN index.  
Value range: 64–INT\_MAX. The unit is KB.  
Default value: The default value of **gin\_pending\_list\_limit** depends on **gin\_pending\_list\_limit** specified in GUC parameters. By default, the value is 4 MB.
- **WHERE predicate**

Creates a partial index. A partial index is an index that contains entries for only a portion of a table, usually a portion that is more useful for indexing

than the rest of the table. For example, if you have a table that contains both billed and unbilled orders where the unbilled orders take up a small fraction of the total table and yet that is an often used section, you can improve performance by creating an index on just that portion. Another possible application is to use **WHERE** with **UNIQUE** to enforce uniqueness over a subset of a table.

Value range: predicate expression can refer only to columns of the underlying table, but it can use all columns, not just the ones being indexed. Presently, subquery and aggregate expressions are also forbidden in **WHERE**.

- **PARTITION index\_partition\_name**

Specifies the name of the index partition.

Value range: A string. It must comply with the naming convention rule.

## Examples

```
-- Create the tpcds.ship_mode_t1 table:
CREATE TABLE tpcds.ship_mode_t1
(
  SM_SHIP_MODE_SK      INTEGER      NOT NULL,
  SM_SHIP_MODE_ID     CHAR(16)     NOT NULL,
  SM_TYPE              CHAR(30)     ,
  SM_CODE              CHAR(10)     ,
  SM_CARRIER          CHAR(20)     ,
  SM_CONTRACT         CHAR(20)
)
DISTRIBUTE BY HASH(SM_SHIP_MODE_SK);

-- Create a common index on the SM_SHIP_MODE_SK column in the tpcds.ship_mode_t1 table:
CREATE UNIQUE INDEX ds_ship_mode_t1_index1 ON tpcds.ship_mode_t1(SM_SHIP_MODE_SK);

-- Create a B-tree index on the SM_SHIP_MODE_SK column in the tpcds.ship_mode_t1 table:
CREATE INDEX ds_ship_mode_t1_index4 ON tpcds.ship_mode_t1 USING btree(SM_SHIP_MODE_SK);

-- Create an expression index on the SM_CODE column in the tpcds.ship_mode_t1 table:
CREATE INDEX ds_ship_mode_t1_index2 ON tpcds.ship_mode_t1(SUBSTR(SM_CODE,1,4));

-- Create a partial index on the SM_SHIP_MODE_SK column where SM_SHIP_MODE_SK is greater than 10
in the tpcds.ship_mode_t1 table:
CREATE UNIQUE INDEX ds_ship_mode_t1_index3 ON tpcds.ship_mode_t1(SM_SHIP_MODE_SK) WHERE
SM_SHIP_MODE_SK>10;

-- Rename an existing index:
ALTER INDEX tpcds.ds_ship_mode_t1_index1 RENAME TO ds_ship_mode_t1_index5;

-- Set the index as unusable:
ALTER INDEX tpcds.ds_ship_mode_t1_index2 UNUSABLE;

-- Recreate an index:
ALTER INDEX tpcds.ds_ship_mode_t1_index2 REBUILD;

-- Delete an existing index:
DROP INDEX tpcds.ds_ship_mode_t1_index2;

-- Delete the tables:
DROP TABLE tpcds.ship_mode_t1;

--Create the tpcds.customer_address_p1 table:
CREATE TABLE tpcds.customer_address_p1
(
  CA_ADDRESS_SK      INTEGER      NOT NULL,
  CA_ADDRESS_ID     CHAR(16)     NOT NULL,
  CA_STREET_NUMBER  CHAR(10)     ,
  CA_STREET_NAME    VARCHAR(60) ,
)
```

```
CA_STREET_TYPE      CHAR(15)
CA_SUITE_NUMBER     CHAR(10)
CA_CITY             VARCHAR(60)
CA_COUNTY           VARCHAR(30)
CA_STATE            CHAR(2)
CA_ZIP              CHAR(10)
CA_COUNTRY           VARCHAR(20)
CA_GMT_OFFSET       DECIMAL(5,2)
CA_LOCATION_TYPE    CHAR(20)
)
DISTRIBUTE BY HASH(CA_ADDRESS_SK)
PARTITION BY RANGE(CA_ADDRESS_SK)
(
  PARTITION p1 VALUES LESS THAN (3000),
  PARTITION p2 VALUES LESS THAN (5000) ,
  PARTITION p3 VALUES LESS THAN (MAXVALUE)
)
ENABLE ROW MOVEMENT;
-- Create the partitioned table index ds_customer_address_p1_index1 with the name of the index partition
not specified:
CREATE INDEX ds_customer_address_p1_index1 ON tpcds.customer_address_p1(CA_ADDRESS_SK) LOCAL;
-- Create the partitioned table index ds_customer_address_p1_index2 with the name of the index partition
specified:
CREATE INDEX ds_customer_address_p1_index2 ON tpcds.customer_address_p1(CA_ADDRESS_SK) LOCAL
(
  PARTITION CA_ADDRESS_SK_index1,
  PARTITION CA_ADDRESS_SK_index2 TABLESPACE example3,
  PARTITION CA_ADDRESS_SK_index3 TABLESPACE example4
)
;

-- Rename a partitioned table index:
ALTER INDEX tpcds.ds_customer_address_p1_index2 RENAME PARTITION CA_ADDRESS_SK_index1 TO
CA_ADDRESS_SK_index4;

-- Delete the indexes and partitioned table:
DROP INDEX tpcds.ds_customer_address_p1_index1;
DROP INDEX tpcds.ds_customer_address_p1_index2;
DROP TABLE tpcds.customer_address_p1;
```

## Helpful Links

[ALTER INDEX](#), [DROP INDEX](#)

# 12.40 CREATE NODE

## Function

**CREATE NODE** creates a cluster node.

## Precautions

**CREATE NODE** is the internal interface encapsulated in **gs\_om**. You are not advised to use this interface, because doing so affects the cluster.

## Syntax

```
CREATE NODE nodename WITH
(
  [ TYPE = nodetype,]
  [ HOST = hostname,]
  [ PORT = portnum,]
  [ HOST1 = 'hostname',]
```

```
[ PORT1 = portnum,]  
[ HOSTPRIMARY [= boolean ],]  
[ PRIMARY [= boolean ],]  
[ PREFERRED [= boolean ],]  
[ Sctp_PORT = portnum,]  
[ CONTROL_PORT = portnum,]  
[ Sctp_PORT1 = portnum,]  
[ CONTROL_PORT1 = portnum ]  
);
```

## Parameter description

- **nodename**  
Specifies the node name.  
Value range: A string. It must comply with the naming convention rule.
- **TYPE = nodetype**  
Indicates the type of a node.  
Valid value:
  - 'coordinator'
  - 'datanode'
- **HOST = hostname**  
Indicates the primary server name or IP address of a node.
- **PORT = portnum**  
Indicates the primary server port to which a node is bound.
- **HOST1 = hostname**  
Indicates the name or IP address of the standby server of a node.
- **PORT1 = portnum**  
Indicates the port number of the standby server to which a node is bound.
- **HOSTPRIMARY**
- **PRIMARY = boolean**  
Specifies whether the node is a primary node or not. A primary node allows read/write operations. A non-primary node allows only read operations.  
Valid value:
  - true
  - **false** (default value)
- **PREFERRED = boolean**  
Specifies whether the node is a preferred node for read operations.  
Valid value:
  - true
  - **false** (default value)
- **Sctp\_PORT = portnum**  
Specifies the port used by the TCP proxy communication library or Sctp communication library of the primary server to listen to the service data transmission channel. It may be a TCP or Sctp port.
- **CONTROL\_PORT = portnum**

Specifies the port used by the TCP proxy communication library or SCTP communication library of the primary server to listen to the control data transmission channel. It is a TCP port.

- **SCTP\_PORT1 = portnum**

Specifies the port used by the TCP proxy communication library or SCTP communication library of the standby server to listen to the service data transmission channel. It may be a TCP or SCTP port.

- **CONTROL\_PORT 1= portnum**

Specifies the port used by the TCP proxy communication library or SCTP communication library of the standby server to listen to the control data transmission channel. It is a TCP port.

## Helpful Links

[ALTER NODE](#), [DROP NODE](#)

# 12.41 CREATE NODE GROUP

## Function

**CREATE NODE GROUP** creates a cluster node group.

## Precautions

- **CREATE NODE GROUP** is the internal interface encapsulated in **gs\_om**.
- This interface is available only to administrators.

## Syntax

```
CREATE NODE GROUP groupname  
WITH ( nodename [, ... ] )  
[ BUCKETS [ ( bucketnumber [, ... ] ) ] ] [VCGROUP] [DISTRIBUTE FROM src_group_name];
```

## Parameter Description

- **groupname**

Specifies the name of a node group.

Value range: a string. It must comply with the naming convention. A value can contain a maximum of 63 characters.

 **NOTE**

A node group name supports all ASCII characters, but you are advised to name a node group according to the naming convention.

- **nodename**

Specifies the node name.

Value range: a string compliant with the naming convention rules. A value can contain a maximum of 63 characters.

- **BUCKETS [ ( bucketnumber [, ... ] ) ]**

Is designed for internal use of the cluster management tools. You are not advised to use it directly. Otherwise, the cluster may be affected.



- **VCGROUP**  
Creates a Node Group used as a logical cluster.
- **DISTRIBUTE FROM src\_group\_name**  
Creates a Node Group to redistribute data in the Node Group specified by **src\_group\_name**. You are not advised to use this clause, because it may lead to data distribution errors or Node Group unavailability.

## Helpful Links

[DROP NODE GROUP](#)

# 12.42 CREATE ROW LEVEL SECURITY POLICY

## Function

**CREATE ROW LEVEL SECURITY POLICY** creates a row-level access control policy for a table.

The policy takes effect only after row-level access control is enabled (by running **ALTER TABLE... ENABLE ROW LEVEL SECURITY**).

Currently, row-level access control affects the read (**SELECT**, **UPDATE**, **DELETE**) of data tables and does not affect the write (**INSERT** and **MERGE INTO**) of data tables. The table owner or system administrators can create an expression in the **USING** clause. When the client reads the data table, the database server combines the expressions that meet the condition and applies it to the execution plan in the statement rewriting phase of a query. For each tuple in a data table, if the expression returns **TRUE**, the tuple is visible to the current user; if the expression returns **FALSE** or **NULL**, the tuple is invisible to the current user.

A row-level access control policy name is specific to a table. A data table cannot have row-level access control policies with the same name. Different data tables can have the same row-level access control policy.

Row-level access control policies can be applied to specified operations (**SELECT**, **UPDATE**, **DELETE**, and **ALL**). **ALL** indicates that **SELECT**, **UPDATE**, and **DELETE** will be affected. For a new row-level access control policy, the default value **ALL** will be used if you do not specify the operations that will be affected.

Row-level access control policies can be applied to a specified user (role) or to all users (**PUBLIC**). For a new row-level access control policy, the default value **PUBLIC** will be used if you do not specify the user that will be affected.

## Precautions

- Row-level access control policies can be defined for row-store tables, row-store partitioned tables, column-store tables, column-store partitioned tables, replication tables, unlogged tables, and hash tables.
- Row-level access control policies cannot be defined for HDFS tables, foreign tables, and temporary tables.
- Row-level access control policies cannot be defined for views.
- A maximum of 100 row-level access control policies cannot be defined for a table.

- System administrators are not affected by row-level access control policies and can view all data in a table.
- Tables queried by using SQL statements, views, functions, and stored procedures are affected by row-level access control policies.

## Syntax

```
CREATE [ ROW LEVEL SECURITY ] POLICY policy_name ON table_name
[ AS { PERMISSIVE | RESTRICTIVE } ]
[ FOR { ALL | SELECT | INSERT | UPDATE | DELETE } ]
[ TO { role_name | PUBLIC | CURRENT_USER | SESSION_USER } [, ...] ]
USING ( using_expression )
```

## Parameter Description

- *policy\_name*  
Specifies the name of a row-level access control policy to be created. The names of row-level access control policies for a table must be unique.
- *table\_name*  
Specifies the name of a table to which a row-level access control policy is applied.
- *command*  
Specifies the SQL operations affected by a row-level access control policy, including **ALL**, **SELECT**, **UPDATE**, and **DELETE**. If this parameter is not specified, the default value **ALL** will be used, covering **SELECT**, **UPDATE**, and **DELETE**.  
  
If *command* is set to **SELECT**, only tuple data that meets the condition (the return value of *using\_expression* is **TRUE**) can be queried. The operations that are affected include **SELECT**, **UPDATE... RETURNING**, and **DELETE... RETURNING**.  
  
If *command* is set to **UPDATE**, only tuple data that meets the condition (the return value of *using\_expression* is **TRUE**) can be updated. The operations that are affected include **UPDATE**, **UPDATE ... RETURNING**, and **SELECT ... FOR UPDATE/SHARE**.  
  
If *command* is set to **DELETE**, only tuple data that meets the condition (the return value of *using\_expression* is **TRUE**) can be deleted. The operations that are affected include **DELETE** and **DELETE ... RETURNING**.  
  
The following table describes the relationship between row-level access control policies and SQL statements.

**Table 12-16** Relationship between row-level access control policies and SQL statements

| Command                        | SELECT/ALL Policy | UPDATE/ALL Policy | DELETE/ALL Policy |
|--------------------------------|-------------------|-------------------|-------------------|
| <b>SELECT</b>                  | Existing row      | No                | No                |
| <b>SELECT FOR UPDATE/SHARE</b> | Existing row      | Existing row      | No                |
| <b>UPDATE</b>                  | No                | Existing row      | No                |

| Command                 | SELECT/ALL Policy | UPDATE/ALL Policy | DELETE/ALL Policy |
|-------------------------|-------------------|-------------------|-------------------|
| <b>UPDATE RETURNING</b> | Existing row      | Existing row      | No                |
| <b>DELETE</b>           | No                | No                | Existing row      |
| <b>DELETE RETURNING</b> | Existing row      | No                | Existing row      |

- role\_name*

Specifies database users affected by a row-level access control policy. If this parameter is not specified, the default value **PUBLIC** will be used, indicating that all database users will be affected. You can specify multiple affected database users.

---

**NOTICE**

System administrators are not affected by row access control.

---

- using\_expression*

Specifies an expression defined for a row-level access control policy (return type: boolean).

The expression cannot contain aggregate functions and window functions. In the statement rewriting phase of a query, if row-level access control for a data table is enabled, the expressions that meet the specified conditions will be added to the plan tree. The expression is calculated for each tuple in the data table. For **SELECT**, **UPDATE**, and **DELETE**, row data is visible to the current user only when the return value of the expression is **TRUE**. If the expression returns **FALSE**, the tuple is invisible to the current user. In this case, the user cannot view the tuple through the **SELECT** statement, update the tuple through the **UPDATE** statement, or delete the tuple through the **DELETE** statement.

## Examples

```
-- Create user alice.
CREATE ROLE alice PASSWORD 'Gauss@123';

-- Create user bob.
CREATE ROLE bob PASSWORD 'Gauss@123';

-- Create the data table all_data.
CREATE TABLE all_data(id int, role varchar(100), data varchar(100));

-- Insert data into the data table.
INSERT INTO all_data VALUES(1, 'alice', 'alice data');
INSERT INTO all_data VALUES(2, 'bob', 'bob data');
INSERT INTO all_data VALUES(3, 'peter', 'peter data');

-- Grant the read permission for the all_data table to users alice and bob.
GRANT SELECT ON all_data TO alice, bob;

-- Enable row-level access control.
```

```
ALTER TABLE all_data ENABLE ROW LEVEL SECURITY;

-- Create a row-level access control policy to specify that the current user can view only their own data.
CREATE ROW LEVEL SECURITY POLICY all_data_rls ON all_data USING(role = CURRENT_USER);

-- View information about the all_data table.
\d+ all_data
      Table "public.all_data"
  Column |          Type          | Modifiers | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----
  id     | integer                |           |         |              |
  role   | character varying(100) |           | extended |              |
  data   | character varying(100) |           | extended |              |
Row Level Security Policies:
  POLICY "all_data_rls"
  USING (((role)::name = "current_user"()))
Has OIDs: no
Distribute By: HASH(id)
Location Nodes: ALL DATANODES
Options: orientation=row, compression=no, enable_rowsecurity=true

-- Run SELECT.
SELECT * FROM all_data;
 id | role | data
-----+-----+-----
  1 | alice | alice data
  2 | bob   | bob data
  3 | peter | peter data
(3 rows)

EXPLAIN(COSTS OFF) SELECT * FROM all_data;
      QUERY PLAN
-----
Streaming (type: GATHER)
  Node/s: All datanodes
  -> Seq Scan on all_data
(3 rows)

-- Switch to user alice and run SELECT.
SELECT * FROM all_data;
 id | role | data
-----+-----+-----
  1 | alice | alice data
(1 row)

EXPLAIN(COSTS OFF) SELECT * FROM all_data;
      QUERY PLAN
-----
Streaming (type: GATHER)
  Node/s: All datanodes
  -> Seq Scan on all_data
      Filter: ((role)::name = 'alice'::name)
Notice: This query is influenced by row level security feature
(5 rows)
```

## Helpful Links

[DROP ROW LEVEL SECURITY POLICY](#)

# 12.43 CREATE PROCEDURE

## Function

**CREATE PROCEDURE** creates a stored procedure.

## Precautions

- The precision values (if any) of the parameters or return values of a stored procedure are not checked.
- When creating a stored procedure, you are advised to display the specified schema for the operations on the table objects in the stored procedure definition. Otherwise, the stored procedure may fail to be executed.
- **current\_schema** and **search\_path** specified by **SET** during stored procedure creation are invalid. **search\_path** and **current\_schema** before and after function execution should be the same.
- If a stored procedure has output parameters, the **SELECT** statement uses the default values of the output parameters when calling the procedure. When the **CALL** statement calls the stored procedure, it requires that the output parameter values are adapted to Oracle. When the **CALL** statement calls a non-overloaded function, output parameters must be specified. When the **CALL** statement calls an overloaded PACKAGE function, it can use the default values of the output parameters. For details, see examples in [CALL](#).
- A stored procedure with the PACKAGE attribute can use overloaded functions.
- When you create a procedure, you cannot insert aggregate functions or other functions out of the average function.

## Syntax

```
CREATE [ OR REPLACE ] PROCEDURE procedure_name
[ ( ( [ argmode ] [ argname ] argtype [ { DEFAULT | := | = } expression ] ) [ ... ] ) ]
[
  { IMMUTABLE | STABLE | VOLATILE }
  | { SHIPPABLE | NOT SHIPPABLE }
  | { PACKAGE }
  | [ NOT ] LEAKPROOF
  | { CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT }
  | [ [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER | AUTHID DEFINER | AUTHID
CURRENT_USER }
  | COST execution_cost
  | ROWS result_rows
  | SET configuration_parameter { [ TO | = ] value | FROM CURRENT }
] [ ... ]
{ IS | AS }
plsql_body
/
```

## Parameter Description

- **OR REPLACE**  
Replaces the original definition when two stored procedures are with the same name.
- **procedure\_name**  
Specifies the name of the stored procedure that is created (optionally with schema names).  
Value range: a string. It must comply with the naming convention.
- **argmode**  
Specifies the mode of an argument.

**NOTICE**

**VARIADIC** specifies arguments of array types.

Value range: **IN**, **OUT**, **IN OUT**, **INOUT**, and **VARIADIC**. The default value is **IN**. Only the argument of **OUT** mode can be followed by **VARIADIC**. The parameters of **OUT** and **INOUT** cannot be used in procedure definition of **RETURNS TABLE**.

- **argname**  
Specifies the name of an argument.  
Value range: a string. It must comply with the naming convention.
- **argtype**  
Specifies the type of a parameter.  
Value range: A valid data type.
- **IMMUTABLE, STABLE, ...**  
Specifies a constraint. Parameters here are similar to those of **CREATE FUNCTION**. For details, see [5.18.17.13-CREATE FUNCTION](#).
- **plsql\_body**  
Indicates the PL/SQL stored procedure body.

**NOTICE**

When you create a user, or perform other operations requiring password input in a stored procedure, the system catalog and csv log records the unencrypted password. Therefore, you are advised not to perform such operations in the stored procedure.

 **NOTE**

No specific order is applied to **argument\_name** and **argmode**. The following order is advised: **argument\_name**, **argmode**, and **argument\_type**.

## Examples

```
-- Create a stored procedure:
CREATE OR REPLACE PROCEDURE prc_add
(
    param1 IN INTEGER,
    param2 IN OUT INTEGER
)
AS
BEGIN
    param2:= param1 + param2;
    dbms_output.put_line('result is: '||to_char(param2));
END;
/

-- Call the stored procedure:
SELECT prc_add(2,3);

-- Create a stored procedure whose parameter type is VARIADIC:
CREATE OR REPLACE PROCEDURE pro_variadic (var1 VARCHAR2(10) DEFAULT 'hello!',var4 VARIADIC int4[])
AS
```

```
BEGIN
  dbms_output.put_line(var1);
END;
/

-- Execute the stored procedure:
SELECT pro_variadic(var1=>'hello', VARIADIC var4=> array[1,2,3,4]);

-- Create a stored procedure with the permission of the user who calls it:
CREATE PROCEDURE insert_data(v integer)
SECURITY INVOKER
AS
BEGIN
  INSERT INTO tb1 VALUES(v);
END;
/

-- Call the stored procedure:
CALL insert_data1(1);

-- Create a stored procedure with the PACKAGE attribute:
create or replace procedure package_func_overload(col int, col2 out varchar)
package
as
declare
  col_type text;
begin
  col2 := '122';
  dbms_output.put_line('two varchar parameters ' || col2);
end;
/

-- Delete the stored procedure:
DROP PROCEDURE prc_add;
DROP PROCEDURE pro_variadic;
DROP PROCEDURE insert_data;
DROP PROCEDURE package_func_overload;
```

## Helpful Links

[DROP PROCEDURE](#)

# 12.44 CREATE RESOURCE POOL

## Function

**CREATE RESOURCE POOL** creates a resource pool and specifies the Cgroup for the resource pool.

## Precautions

As long as the current user has **CREATE** permission, it can create a resource pool.

## Syntax

```
CREATE RESOURCE POOL pool_name
  [WITH ({MEM_PERCENT=pct | CONTROL_GROUP="group_name" | ACTIVE_STATEMENTS=stmt |
  MAX_DOP = dop | MEMORY_LIMIT='memory_size' | io_limits=io_limits | io_priority='io_priority' |
  nodegroup="nodegroupname" | is_foreign=boolean }[, ... ])];
```

## Parameter Description

- **pool\_name**

Specifies the name of a resource pool.

The name of a resource pool cannot be same as that of an existing resource pool.

Value range: a string. It must comply with the naming convention.

- **group\_name**

Specifies the name of a Cgroup.

 **NOTE**

- You can use either double quotation marks ("" ) or single quotation mark (') in the syntax when setting the name of a Cgroup.
- The value of **group\_name** is case-sensitive.
- If **group\_name** is not specified, the string "Medium" will be used by default in the syntax, indicating the **Medium** Timeshare Cgroup under **DefaultClass**.
- If an administrator specifies a Workload Cgroup under Class, for example, **control\_group** set to **class1:workload1**, the resource pool will be associated with the **workload1** Cgroup under **class1**. The level of Workload can also be specified. For example, **control\_group** is set to **class1:workload1:1**.
- If a database user specifies the Timeshare string (**Rush**, **High**, **Medium**, or **Low**) in the syntax, for example, if **control\_group** is set to **High**, the resource pool will be associated with the **High** Timeshare Cgroup under **DefaultClass**.
- In multi-tenant scenarios, the Cgroup associated with a group resource pool is a Class Cgroup, and that associated with a service resource pool is a Workload Cgroup. Additionally, switching Cgroups between different resource pools is not allowed.

Value range: a string. It must comply with the rule in the description, specifying an existing Cgroup.

- **stmt**

Specifies the maximum number of statements that can be concurrently executed in a resource pool.

Value range: Numeric data ranging from **-1** to **INT\_MAX**.

- **dop**

This is a reserved parameter.

Value range: Numeric data ranging from **1** to **INT\_MAX**.

- **memory\_size**

Specifies the maximum storage for a resource pool.

Value range: a string, from **1KB** to **2047GB**.

- **mem\_percent**

Specifies the proportion of available resource pool memory to the total memory or group user memory.

In multi-tenant scenarios, **mem\_percent** of group users or service users ranges from **1** to **100**. The default value is **20**.

In common scenarios, **mem\_percent** of common users ranges from **0** to **100**. The default value is **0**.



 NOTE

When both of **mem\_percent** and **memory\_limit** are specified, only **mem\_percent** takes effect.

- **io\_limits**

Specifies the upper limit of IOPS in a resource pool.

The IOPS is counted by ones for column storage and by 10 thousands for row storage.

- **io\_priority**

Specifies the I/O priority for jobs that consume many I/O resources. It takes effect when the I/O usage reaches 90%.

There are three priorities: **Low**, **Medium**, and **High**. If you do not want to control I/O resources, use the default value **None**.

 NOTE

The settings of **io\_limits** and **io\_priority** are valid only for complex jobs, such as batch import (using **INSERT INTO SELECT**, **COPY FROM**, or **CREATE TABLE AS**), complex queries involving over 500 MB data on each DN, and **VACUUM FULL**.

## Examples

This example assumes that Cgroups have been created by users in advance.

```
-- Create a default resource pool, and associate it with the Medium Timeshare Cgroup under Workload under DefaultClass:  
CREATE RESOURCE POOL pool1;  
  
-- Create a resource pool, and associate it with the High Timeshare Cgroup under Workload under DefaultClass:  
CREATE RESOURCE POOL pool2 WITH (CONTROL_GROUP="High");  
  
-- Create a resource pool, and associate it with the Low Timeshare Cgroup under Workload under class1:  
CREATE RESOURCE POOL pool3 WITH (CONTROL_GROUP="class1:Low");  
  
-- Create a resource pool, and associate it with the wg1 Workload Cgroup under class1:  
CREATE RESOURCE POOL pool4 WITH (CONTROL_GROUP="class1:wg1");  
  
-- Create a resource pool, and associate it with the wg2 Workload Cgroup under class1:  
CREATE RESOURCE POOL pool5 WITH (CONTROL_GROUP="class1:wg2:3");  
  
-- Delete the resource pools:  
DROP RESOURCE POOL pool1;  
DROP RESOURCE POOL pool2;  
DROP RESOURCE POOL pool3;  
DROP RESOURCE POOL pool4;  
DROP RESOURCE POOL pool5;
```

## Helpful Links

[ALTER RESOURCE POOL, DROP RESOURCE POOL](#)

# 12.45 CREATE ROLE

## Function

**CREATE ROLE** creates a role.

A role is an entity that has own database objects and permissions. In different environments, a role can be considered a user, a group, or both.

## Precautions

- **CREATE ROLE** adds a role to a database. The role does not have the login permission.
- Only the user who has the **CREATE ROLE** permission or a system administrator is allowed to create roles.

## Syntax

```
CREATE ROLE role_name [ [ WITH ] option [ ... ] ] [ ENCRYPTED | UNENCRYPTED ] { PASSWORD | IDENTIFIED BY } { 'password' | DISABLE };
```

The syntax of role information configuration clause **option** is as follows:

```
{SYSADMIN | NOSYSADMIN}
| {AUDITADMIN | NOAUDITADMIN}
| {CREATEDB | NOCREATEDB}
| {USEFT | NOUSEFT}
| {CREATEROLE | NOCREATEROLE}
| {INHERIT | NOINHERIT}
| {LOGIN | NOLOGIN}
| {REPLICATION | NOREPLICATION}
| {INDEPENDENT | NOINDEPENDENT}
| {VCADMIN | NOVCADMIN}
| CONNECTION LIMIT connlimit
| VALID BEGIN 'timestamp'
| VALID UNTIL 'timestamp'
| RESOURCE POOL 'respool'
| USER GROUP 'groupuser'
| PERM SPACE 'spacelimit'
| TEMP SPACE 'tmpspacelimit'
| SPILL SPACE 'spillspacelimit'
| NODE GROUP logic_cluster_name
| IN ROLE role_name [, ...]
| IN GROUP role_name [, ...]
| ROLE role_name [, ...]
| ADMIN rol e_name [, ...]
| USER role_name [, ...]
| SYSID uid
| DEFAULT TABLESPACE tablespace_name
| PROFILE DEFAULT
| PROFILE profile_name
| PGUSER
| LDAP
```

## Parameter Description

- **role\_name**  
Indicates the name of a role.  
Value range: a string. It must comply with the naming convention rule. and can contain a maximum of 63 characters.
- **password**  
Specifies the login password.  
The password must:
  - Contain at least eight characters. This is the default length.
  - Differ from the user name or the user name spelled backward.
  - Contain at least three of the following four character types: uppercase letters, lowercase letters, digits, and special characters, including: ~!@#&

%^&\*()-\_+=\|[]{};,:<.>/?'. If you use characters other than the four types, a warning is displayed, but you can still create the password.

Value range: a string

- **DISABLE**  
By default, you can change your password unless it is disabled. To disable the password of a user, use this parameter. After the password of a user is disabled, the password will be deleted from the system. The user can connect to the database only through external authentication, for example, IAM authentication, Kerberos authentication, or LDAP authentication. Only administrators can enable or disable a password. Common users cannot disable the password of an initial user. To enable a password, run **ALTER USER** and specify the password.
- **ENCRYPTED | UNENCRYPTED**  
These key words control whether the password is stored encrypted in the system catalogs. (If neither is specified, the default behavior is determined by the configuration parameter **password\_encryption\_type**.) According to product security requirements, the password must be stored encrypted. Therefore, **UNENCRYPTED** is forbidden in GaussDB(DWS). If the presented password string is already in SHA256-encrypted format, then it is stored encrypted as-is, regardless of whether **ENCRYPTED** or **UNENCRYPTED** is specified (since the system cannot decrypt the specified encrypted password string). This allows reloading of encrypted passwords during dump/restore.
- **SYSADMIN | NOSYSADMIN**  
Determines whether a new role is a system administrator. Roles having the **SYSADMIN** attribute have the highest permission.  
Value range: If not specified, **NOSYSADMIN** is the default.
- **AUDITADMIN | NOAUDITADMIN**  
Determines whether a role has the audit and management attributes.  
If not specified, **NOAUDITADMIN** is the default.
- **CREATEDB | NOCREATEDB**  
Defines a role's ability to create databases.  
A new role does not have the permission to create databases.  
Value range: If not specified, **NOCREATEDB** is the default.
- **USEFT | NOUSEFT**  
Determines whether a new role can perform operations on foreign tables, such as creating, deleting, modifying, and reading/writing foreign tables.  
A new role does not have permissions for these operations.  
The default value is **NOUSEFT**.
- **CREATEROLE | NOCREATEROLE**  
Determines whether a role will be permitted to create new roles (that is, execute **CREATE ROLE** and **CREATE USER**). A role with the **CREATEROLE** permission can also modify and delete other roles.  
Value range: If not specified, **NOCREATEROLE** is the default.
- **INHERIT | NOINHERIT**  
Determines whether a role "inherits" the permissions of roles it is a member of. You are not advised to execute them.

- **LOGIN | NOLOGIN**

Determines whether a role is allowed to log in to a database. A role having the **LOGIN** attribute can be thought of as a user.

Value range: If not specified, **NOLOGIN** is the default.

- **REPLICATION | NOREPLICATION**

Determines whether a role is allowed to initiate streaming replication or put the system in and out of backup mode. A role having the **REPLICATION** attribute is a highly privileged role, and should only be used on roles used for replication.

If not specified, **NOREPLICATION** is the default.

- **INDEPENDENT | NOINDEPENDENT**

Defines private, independent roles. For a role with the **INDEPENDENT** attribute, administrators' rights to control and access this role are separated. Specific rules are as follows:

- Administrators have no rights to add, delete, query, modify, copy, or authorize the corresponding table objects without the authorization from the **INDEPENDENT** role.
- Administrators have no rights to modify the inheritance relationship of the **INDEPENDENT** role without the authorization from this role.
- Administrators have no rights to modify the owner of the table objects for the **INDEPENDENT** role.
- Administrators have no rights to delete the **INDEPENDENT** attribute of the **INDEPENDENT** role.
- Administrators have no rights to change the database password of the **INDEPENDENT** role. The **INDEPENDENT** role must manage its own password, which cannot be reset if lost.
- The **SYSADMIN** attribute of a user cannot be changed to the **INDEPENDENT** attribute.

- **VCADMIN | NOVCADMIN**

Defines the role of a logical cluster administrator. A logical cluster administrator has the following more permissions than common users:

- Create, modify, and delete resource pools in the associated logical cluster.
- Grant the access permission for the associated logical cluster to other users or roles, or reclaim the access permission from those users or roles.

- **CONNECTION LIMIT**

Indicates how many concurrent connections the role can make.

Value range: Integer,  $\geq -1$ . The default value is **-1**, which means unlimited.

---

**NOTICE**

To ensure the proper running of a cluster, the minimum value of **CONNECTION LIMIT** is the number of CNs in the cluster, because when a cluster runs **ANALYZE** on a CN, other CNs will connect the running CN for metadata synchronization. For example, if there are three CNs in the cluster, set **CONNECTION LIMIT** to **3** or a greater value.

---

- **VALID BEGIN**  
Sets a date and time when the role's password becomes valid. If this clause is omitted, the password will be valid for all time.
- **VALID UNTIL**  
Sets a date and time after which the role's password is no longer valid. If this clause is omitted, the password will be valid for all time.
- **RESOURCE POOL**  
Sets the name of resource pool used by the role, and the name belongs to the system catalog: **pg\_resource\_pool**.
- **USER GROUP 'groupuser'**  
Creates a sub-user.
- **PERM SPACE**  
Sets the storage space of the user permanent table.
- **TEMP SPACE**  
Sets the storage space of the user temporary table.
- **SPILL SPACE**  
Sets the operator disk flushing space of the user.
- **NODE GROUP**  
Specifies the name of the logical cluster associated with a user. If the name contains uppercase characters or special characters, enclose the name with double quotation marks.
- **IN ROLE**  
Lists one or more existing roles whose permissions will be inherited by a new role. You are not advised to execute them.
- **IN GROUP**  
Indicates an obsolete spelling of **IN ROLE**. You are not advised to execute them.
- **ROLE**  
Lists one or more existing roles which are automatically added as members of the new role.
- **ADMIN**  
Is similar to **ROLE**. However, the roles after **ADMIN** can grant rights of new roles to other roles.
- **USER**  
Indicates an obsolete spelling of the **ROLE** clause.
- **SYSID**  
The **SYSID** clause is ignored.
- **DEFAULT TABLESPACE**  
The **DEFAULT TABLESPACE** clause is ignored.
- **PROFILE**  
The **PROFILE** clause is ignored.
- **PGUSER**  
This attribute is used to be compatible with open-source Postgres communication. An open-source Postgres client interface (Postgres 9.2.19 is

recommended) can use a database user having this attribute to connect to the database.

---

**NOTICE**

This attribute only ensures compatibility with the connection process. Incompatibility caused by kernel differences between this product and Postgres cannot be solved using this attribute.

Users having the **PGUSER** attribute are authenticated in a way different from other users. Error information reported by the open-source client may cause the attribute to be enumerated. Therefore, you are advised to use a client of this product. For example:

```
# normaluser is a user that does not have the PGUSER attribute. psql is the Postgres client tool.
pg@MPPDB04:~> psql -d postgres -p 8000 -h 10.11.12.13 -U normaluser
psql: authentication method 10 not supported
```

```
# pguser is a user having the PGUSER attribute.
pg@MPPDB04:~> psql -d postgres -p 8000 -h 10.11.12.13 -U pguser
Password for user pguser:
```

---

- LDAP

This attribute is used to specify the role authentication type as **LDAP**. LDAP authentication is an external authentication mode. Therefore, **PASSWORD DISABLE** must be specified.

---

**NOTICE**

- Additional information about LDAP authentication can be added to the LDAP field, for example, **fulluser** in LDAP authentication, which is equivalent to **ldapprefix+username+ldapsuffix**. The LDAP field must be enclosed in double quotation marks. Only lowercase letters are allowed. If the LDAP field indicates only the role authentication type, the value is case insensitive. In this case, **ldapprefix** and **ldapsuffix** are provided by the matched record in **pg\_hba.conf**.
  - When executing the **ALTER ROLE** command, users are not allowed to change the authentication type. Only LDAP users are allowed to modify LDAP attributes.
- 

## Examples

```
-- Create a role manager whose password is Bigdata123@:
CREATE ROLE manager IDENTIFIED BY 'Bigdata123@';

-- Create a role with a validity from January 1, 2015 to January 1, 2026:
CREATE ROLE miriam WITH LOGIN PASSWORD 'Bigdata123@' VALID BEGIN '2015-01-01' VALID UNTIL '2026-01-01';

-- Create a role. The authentication type is LDAP. Other LDAP authentication information is provided by pg_hba.conf:
CREATE ROLE role1 WITH LOGIN LDAP PASSWORD DISABLE;

-- Create a role. The authentication type is LDAP. The fulluser information for LDAP authentication is specified during the role creation. In this case, LDAP is case sensitive and must be enclosed in double quotation marks:
CREATE ROLE role2 WITH LOGIN "ldapcn=role2,cn=user,dc=lwork,dc=com" PASSWORD DISABLE;
```

```
-- Alter the password of role manager to abcd@123:  
ALTER ROLE manager IDENTIFIED BY 'abcd@123' REPLACE 'Bigdata123@';  
  
-- Alter role manager to the system administrator:  
ALTER ROLE manager SYSADMIN;  
  
-- Modify the fulluser information of the LDAP authentication role:  
ALTER ROLE role2 WITH LOGIN "ldapcn=role2,cn=user2,dc=func,dc=com" PASSWORD DISABLE;  
  
-- Delete the manager role:  
DROP ROLE manager;  
  
-- Delete the role miriam:  
DROP ROLE miriam;  
  
-- Delete the role1 role:  
DROP ROLE role1;  
  
-- Delete the role2 role:  
DROP ROLE role2;
```

## Helpful Links

[SET ROLE](#), [ALTER ROLE](#), [DROP ROLE](#), [GRANT](#), [REVOKE](#)

# 12.46 CREATE SCHEMA

## Function

**CREATE SCHEMA** creates a schema.

Named objects are accessed either by "qualifying" their names with the schema name as a prefix, or by setting a search path that includes the desired schema(s). When creating named objects, you can also use the schema name as a prefix.

Optionally, **CREATE SCHEMA** can include sub-commands to create objects within the new schema. The sub-commands are treated essentially the same as separate commands issued after creating the schema, If the **AUTHORIZATION** clause is used, all the created objects are owned by this user.

## Precautions

- As long as the current user has the CREATE permission, the user can create a schema.
- The owner of an object created by a system administrator in a schema with the same name as a common user is the common user, not the system administrator.

## Syntax

- Create a schema based on a specified name.  
CREATE SCHEMA schema\_name  
[ AUTHORIZATION user\_name ] [ schema\_element [ ... ] ];
- Create a schema based on a user name.  
CREATE SCHEMA AUTHORIZATION user\_name [ schema\_element [ ... ] ];

## Parameter Description

- **schema\_name**  
Indicates the name of the schema to be created.

---

### NOTICE

The name must be unique,  
and cannot start with **pg\_**.

---

Value range: a string. It must comply with the naming convention rule.

- **AUTHORIZATION user\_name**  
Indicates the name of the user who will own this schema. If **schema\_name** is not specified, **user\_name** will be used as the schema name. In this case, **user\_name** can only be a role name.

Value range: An existing user name/role.

- **schema\_element**  
Indicates an SQL statement defining an object to be created within the schema. Currently, only **CREATE TABLE**, **CREATE VIEW**, **CREATE INDEX**, **CREATE PARTITION**, and **GRANT** are accepted as clauses within **CREATE SCHEMA**.

Objects created by sub-commands are owned by the user specified by **AUTHORIZATION**.

### NOTE

If objects in the schema on the current search path are with the same name, specify the schemas different objects are in. You can run the **SHOW SEARCH\_PATH** command to check the schemas on the current search path.

## Examples

```
-- Create the role1 role:  
CREATE ROLE role1 IDENTIFIED BY 'Bigdata123@';  
  
-- Create a schema named role1 for the role1 role. The owner of the films and winners tables created by  
the clause is role1.  
CREATE SCHEMA AUTHORIZATION role1  
  CREATE TABLE films (title text, release date, awards text[])  
  CREATE VIEW winners AS  
  SELECT title, release FROM films WHERE awards IS NOT NULL;  
  
-- Delete the schema:  
DROP SCHEMA role1 CASCADE;  
-- Delete the user:  
DROP USER role1 CASCADE;
```

## Helpful Links

[ALTER SCHEMA, DROP SCHEMA](#)



## 12.47 CREATE SEQUENCE

### Function

**CREATE SEQUENCE** adds a sequence to the current database. The owner of a sequence is the user who creates the sequence.

### Precautions

- A sequence is a special table that stores arithmetic sequence. Such a table is controlled by DBMS. It has no actual meaning and is usually used to generate unique identifiers for rows or tables.
- If a schema name is given, the sequence is created in the specified schema; otherwise, it is created in the current schema. The sequence name must be different from the names of other sequences, tables, indexes, views in the same schema.
- After the sequence is created, functions `nextval()` and `generate_series(1,N)` insert data to the table. Make sure that the number of times for invoking `nextval` is greater than or equal to  $N+1$ . Otherwise, errors will be reported because the number of times for invoking function `generate_series()` is  $N+1$ .
- A sequence cannot be created in the **template1** database.

### Syntax

```
CREATE SEQUENCE name [ INCREMENT [ BY ] increment ]  
  [ MINVALUE minvalue | NO MINVALUE | NOMINVALUE ] [ MAXVALUE maxvalue | NO MAXVALUE |  
  NOMAXVALUE ]  
  [ START [ WITH ] start ] [ CACHE cache ] [ [ NO ] CYCLE | NOCYCLE ]  
  [ OWNED BY { table_name.column_name | NONE } ];
```

### Parameter Description

- **name**  
Specifies the name of the sequence to be created.  
Value range: The value can contain only lowercase letters, uppercase letters, special characters `#_`, and digits.
- **increment**  
Specifies the step for a sequence. A positive generates an ascending sequence, and a negative generates a decreasing sequence.  
The default value is **1**.
- **MINVALUE minvalue | NO MINVALUE| NOMINVALUE**  
Specifies the minimum value of the sequence. If **MINVALUE** is not declared, or **NO MINVALUE** is declared, the default value of the ascending sequence is **1**, and that of the descending sequence is  $-2^{63}-1$ . **NOMINVALUE** is equivalent to **NO MINVALUE**.
- **MAXVALUE maxvalue | NO MAXVALUE| NOMAXVALUE**  
Specifies the maximum value in a sequence. If **MAXVALUE** is not declared or **NO MAXVALUE** is declared, the default value of the ascending sequence is

$2^{63}-1$ , and that of the descending sequence is **-1**. **NOMAXVALUE** is equivalent to **NO MAXVALUE**.

- **start**

Specifies the start value of the sequence. The default value for ascending sequences is **minvalue** and for descending sequences **maxvalue**.

- **cache**

Specifies the number sequences stored in the memory for quick access purposes.

Default value **1** indicates that one value can be generated each time.

 **NOTE**

It is not recommended that you define **cache**, and **maxvalue**, and **minvalue** at the same time. The continuity of sequences cannot be ensured after **cache** is defined because valid characters may be generated, causing waste of sequences.

- **CYCLE**

Used to ensure that sequences can recycle after the number of sequences reaches **maxvalue** or **minvalue**.

If you declare **NO CYCLE**, any invocation of **nextval** would return an error after the sequence reaches its maximum value.

**NOCYCLE** is equivalent to **NO CYCLE**.

The default value is **NO CYCLE**.

If the sequence is defined as **CYCLE**, the sequence uniqueness cannot be ensured.

- **OWNED BY-**

Associates a sequence with a specified column included in a table. In this way, the sequence will be deleted when you delete its associated field or the table where the field belongs. The associated table and sequence must be owned by the same user and in the same schema. **OWNED BY** only establishes the association between a table column and the sequence. The sequence is not created for this column.

If the default value is **OWNED BY NONE**, indicating that such association does not exist.

---

**NOTICE**

You are not advised to use the sequence created using **OWNED BY** in other tables. If multiple tables need to share a sequence, the sequence must not belong to a specific table.

---

## Examples

Create an ascending sequence named **serial**, which starts from 101:

```
CREATE SEQUENCE serial
START 101
CACHE 20;
```

Select the next number from the sequence:

```
SELECT nextval('serial');
nextval
-----
101
```

Select the next number from the sequence:

```
SELECT nextval('serial');
nextval
-----
102
```

Create a sequence associated with the table:

```
CREATE TABLE customer_address
(
  ca_address_sk      integer      not null,
  ca_address_id     char(16)     not null,
  ca_street_number  char(10)
  ca_street_name    varchar(60)
  ca_street_type    char(15)
  ca_suite_number   char(10)
  ca_city           varchar(60)
  ca_county         varchar(30)
  ca_state          char(2)
  ca_zip           char(10)
  ca_country        varchar(20)
  ca_gmt_offset     decimal(5,2)
  ca_location_type  char(20)
);

CREATE SEQUENCE serial1
START 101
CACHE 20
OWNED BY customer_address.ca_address_sk;
-- Delete the sequence.
DROP TABLE customer_address;
DROP SEQUENCE serial cascade;
```

## Helpful Links

[DROP SEQUENCE ALTER SEQUENCE](#)

# 12.48 CREATE SERVER

## Function

**CREATE SERVER** creates an external server.

An external server stores HDFS cluster information, OBS server information, or other homogeneous cluster information.

## Precautions

By default, only the system administrator can create a foreign server. Otherwise, creating a server requires USAGE permission on the foreign-data wrapper being used. The syntax is as follows:


```
GRANT USAGE ON FOREIGN DATA WRAPPER fdw_name TO username
```

**fdw\_name** is the name of **FOREIGN DATA WRAPPER**, and **username** is the user name of creating **SERVER**.

## Syntax

```
CREATE SERVER server_name  
  FOREIGN DATA WRAPPER fdw_name  
  OPTIONS ( { option_name ' value ' } [, ...] );
```

## Parameter Description

- **server\_name**  
Specifies the name of the foreign server to be created.  
Value range: The length must be less than or equal to 63.
- **FOREIGN DATA WRAPPER fdw\_name**  
Specifies the name of the foreign data wrapper.  
Value range: **fdw\_name** indicates the data wrapper created by the system in the initial phase of the database. Currently, **fdw\_name** can be **hdfs\_fdw** or **dfs\_fdw** for the HDFS cluster, and can be **gc\_fdw** for other homogeneous clusters.
- **OPTIONS ( { option\_name ' value ' } [, ...] )**  
Specifies the parameters for the server. The detailed parameter description is as follows:
  - address  
Specifies the IP address and port number of a NameNode (metadata node) in the HDFS cluster, or the IP address and port number of a CN in other homogeneous clusters.  
HDFS NameNodes are deployed in primary/secondary mode for HA. Add the addresses of the primary and secondary NameNodes to **ADDRESS**. When accessing HDFS, GaussDB(DWS) dynamically checks for the primary NameNode.  
 **NOTE**
    - **address option** must exist.
    - **address option** only supports IPv4 addresses in dot-decimal notation, and an address string cannot contain spaces. Multiple IP address and port pairs are separated by commas (,). The IP addresses and ports are separated by colons (:). In an HDFS cluster, you are advised to set two groups of *IP.Port* to represent the addresses of the primary and standby NameNodes, respectively.
  - hdfsfcgpath  
This parameter is available only when **type** is **HDFS**.  
You can set the **hdfsfcgpath** parameter to specify the HDFS configuration file path. GaussDB(DWS) accesses the HDFS cluster based on the connection configuration mode and security mode specified in the HDFS configuration file stored in that path.  
If the HDFS cluster is connected in non-secure mode, data transmission encryption is not supported. If **address** has been specified, the HDFS configuration file does not need to be specified. If it has not been specified, the configuration file needs to be specified by **hdfsfcgpath**.
  - type  
Specifies whether **dfs\_fdw** is connected to OBS or HDFS.  
Valid value:

- **OBS** indicates that OBS is connected.
- **HDFS** indicates that HDFS is connected.

## Examples

Create the **hdfs\_server** server, and **hdfs\_fdw** is the built-in foreign-data wrapper.

```
-- Create the hdfs_server server:
CREATE SERVER hdfs_server FOREIGN DATA WRAPPER HDFS_FDW OPTIONS
  (address '10.10.0.100:25000,10.10.0.101:25000',
   hdfsconfigpath '/opt/hadoop_client/HDFS/hadoop/etc/hadoop',
   type 'HDFS'
  );
-- Delete the hdfs_server server:
DROP SERVER hdfs_server;
```

You are advised to create another server in the homogeneous cluster, where **gc\_fdw** is the foreign data wrapper in the database.

```
-- Create a server:
CREATE SERVER server_remote FOREIGN DATA WRAPPER GC_FDW OPTIONS
  (address '10.10.0.100:25000,10.10.0.101:25000',
   dbname 'test',
   username 'test',
   password 'xxxxxxx'
  );
-- Delete the server:
DROP SERVER server_remote;
```

Helpful Links

[ALTER SERVER DROP SERVER](#)

## 12.49 CREATE SYNONYM

### Function

**CREATE SYNONYM** is used to create a synonym object. A synonym is an alias of a database object and is used to record the mapping between database object names. You can use synonyms to access associated database objects.

### Precautions

- The user of a synonym should be its owner.
- If the schema name is specified, create a synonym in the specified schema. Otherwise create a synonym in the current schema.
- Database objects that can be accessed using synonyms include tables, views, functions, and stored procedures.
- To use synonyms, you must have the required permissions on associated objects.
- The following DML statements support synonyms: **SELECT**, **INSERT**, **UPDATE**, **DELETE**, **EXPLAIN**, and **CALL**.

- The **CREATE SYNONYM** statement of an associated function or stored procedure cannot be used in a stored procedure. You are advised to use synonyms existing in the **pg\_synonym** system catalog in the stored procedure.

## Syntax

```
CREATE [ OR REPLACE ] SYNONYM synonym_name  
FOR object_name;
```

## Parameter Description

- **synonym**  
Name of a synonym which is created (optionally with schema names)  
Value range: a string. It must comply with the identifier naming rules.
- **object\_name**  
Name of an object that is associated (optionally with schema names)  
Value range: a string. It must comply with the identifier naming rules.

### NOTE

**object\_name** can be the name of an object that does not exist.

## Examples

```
-- Create schema ot.  
CREATE SCHEMA ot;  
  
-- Create table ot.t1 and its synonym t1.  
CREATE TABLE ot.t1(id int, name varchar2(10)) DISTRIBUTE BY hash(id);  
CREATE OR REPLACE SYNONYM t1 FOR ot.t1;  
  
-- Use synonym t1.  
SELECT * FROM t1;  
INSERT INTO t1 VALUES (1, 'ada'), (2, 'bob');  
UPDATE t1 SET t1.name = 'cici' WHERE t1.id = 2;  
  
-- Create synonym v1 and its associated view ot.v_t1.  
CREATE SYNONYM v1 FOR ot.v_t1;  
CREATE VIEW ot.v_t1 AS SELECT * FROM ot.t1;  
  
-- Use synonym v1.  
SELECT * FROM v1;  
  
-- Create overloaded function ot.add and its synonym add.  
CREATE OR REPLACE FUNCTION ot.add(a integer, b integer) RETURNS integer AS  
$$  
SELECT $1 + $2  
$$  
LANGUAGE sql;  
  
CREATE OR REPLACE FUNCTION ot.add(a decimal(5,2), b decimal(5,2)) RETURNS decimal(5,2) AS  
$$  
SELECT $1 + $2  
$$  
LANGUAGE sql;  
  
CREATE OR REPLACE SYNONYM add FOR ot.add;  
  
-- Use synonym add.  
SELECT add(1,2);  
SELECT add(1.2,2.3);  
  
-- Create stored procedure ot.register and its synonym register.
```

```
CREATE PROCEDURE ot.register(n_id integer, n_name varchar2(10))
SECURITY INVOKER
AS
BEGIN
    INSERT INTO ot.t1 VALUES(n_id, n_name);
END;
/

CREATE OR REPLACE SYNONYM register FOR ot.register;

-- Use synonym register to invoke the stored procedure.
CALL register(3,'mia');

-- Delete the synonym.
DROP SYNONYM t1;
DROP SYNONYM IF EXISTS v1;
DROP SYNONYM IF EXISTS v1;
DROP SYNONYM IF EXISTS add;
DROP SYNONYM register;
DROP SCHEMA ot CASCADE;
```

## Helpful Links

[ALTER SYNONYM DROP SYNONYM](#)

# 12.50 CREATE TABLE

## Function

**CREATE TABLE** creates a table in the current database. The table will be owned by the user who created it.

## Precautions

- For details about the data types supported by column-store tables, see [Data Types Supported by Column-Store Tables](#).
- It is recommended that the number of column-store and HDFS partitioned tables do not exceed 1000.
- The primary key constraint and unique constraint in the table must contain a distribution column.
- If an error occurs during table creation, after it is fixed, the system may fail to delete the empty disk files created before the last automatic clearance. This problem seldom occurs.
- Only a **PARTIAL CLUSTER KEY** constraint can be used as the table-level constraint of column-store tables. Table-level PRIMARY KEY and FOREIGN KEY constraints are not supported.
- Only the NULL, NOT NULL, and DEFAULT constant values can be used as column-store table column constraints.
- Whether column-store tables support a delta table is specified by the **enable\_delta\_store** parameter. The threshold for storing data into a delta table is specified by the **deltarow\_threshold** parameter.

## Syntax

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT EXISTS ] table_name
({ column_name data_type [ compress_mode ] [ COLLATE collation ] [ column_constraint [ ... ] ]
```

```

| table_constraint
| LIKE source_table [ like_option [...] ] }
[ ... ]
[ WITH ( {storage_parameter = value} [, ... ] ) ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ COMPRESS | NOCOMPRESS ]

[ DISTRIBUTE BY { REPLICATION | { HASH ( column_name [,...] ) } } ]
[ TO { GROUP groupname | NODE ( nodename [, ... ] ) } ];

```

- **column\_constraint** is as follows:

```

[ CONSTRAINT constraint_name ]
{ NOT NULL |
NULL |
CHECK ( expression ) |
DEFAULT default_expr |
UNIQUE index_parameters |
PRIMARY KEY index_parameters }
[ DEFERRABLE | NOT DEFERRABLE | INITIALLY DEFERRED | INITIALLY IMMEDIATE ]

```

- **compress\_mode** of a column is as follows:

```

{ DELTA | PREFIX | DICTIONARY | NUMSTR | NOCOMPRESS }

```

- **table\_constraint** is as follows:

```

[ CONSTRAINT constraint_name ]
{ CHECK ( expression ) |
UNIQUE ( column_name [, ... ] ) index_parameters |
PRIMARY KEY ( column_name [, ... ] ) index_parameters |
PARTIAL CLUSTER KEY ( column_name [, ... ] ) }
[ DEFERRABLE | NOT DEFERRABLE | INITIALLY DEFERRED | INITIALLY IMMEDIATE ]

```

- **like\_option** is as follows:

```

{ INCLUDING | EXCLUDING } { DEFAULTS | CONSTRAINTS | INDEXES | STORAGE | COMMENTS |
PARTITION | REOPTIONS | DISTRIBUTION | ALL }

```

- **index\_parameters** is as follows:

```

[ WITH ( {storage_parameter = value} [, ... ] ) ]

```

## Parameter Description

- **UNLOGGED**

If this key word is specified, the created table is not a log table. Data written to unlogged tables is not written to the write-ahead log, which makes them considerably faster than ordinary tables. However, an unlogged table is automatically truncated after a crash or unclean shutdown, incurring data loss risks. The contents of an unlogged table are also not replicated to standby servers. Any indexes created on an unlogged table are not automatically logged as well.

Usage scenario: Unlogged tables do not ensure safe data. Users can back up data before using unlogged tables; for example, users should back up the data before a system upgrade.

Troubleshooting: If data is missing in the indexes of unlogged tables due to some unexpected operations such as an unclean shutdown, users should re-create the indexes with errors.

- **GLOBAL | LOCAL**

When creating a temporary table, you can specify the **GLOBAL** or **LOCAL** keyword before **TEMP** or **TEMPORARY**. Currently, the two keywords are used to be compatible with the SQL standard. In fact, a local temporary table will be created regardless of whether **GLOBAL** or **LOCAL** is specified.

- **TEMPORARY | TEMP**

If **TEMP** or **TEMPORARY** is specified, the created table is a temporary table. Temporary tables are automatically dropped at the end of a session, or



optionally at the end of the current transaction. Therefore, apart from CN and other CN errors connected by the current session, you can still create and use temporary table in the current session. Temporary tables are created only in the current session. If a DDL statement involves operations on temporary tables, a DDL error will be generated. Therefore, you are not advised to perform operations on temporary tables in DDL statements. **TEMP** is equivalent to **TEMPORARY**.

---

**NOTICE**

- Temporary tables are visible to the current session through schema of the **pg\_temp** start. Users should not delete schema started with **pg\_temp**, **pg\_toast\_temp**.
- If **TEMPORARY** or **TEMP** is not specified when you create a table and the schema of the specified table starts with **pg\_temp\_**, the table is created as a temporary table.

- 
- **IF NOT EXISTS**  
Does not throw an error if a table with the same name exists. A notice is issued in this case.
  - **table\_name**  
Specifies the name of the table to be created.
  - **column\_name**  
Specifies the name of a column to be created in the new table.
  - **data\_type**  
Specifies the data type of the column.
  - **compress\_mode**  
Specifies the compress option of the table, only available for row-store table. The option specifies the algorithm preferentially used by table columns.  
Value range: DELTA, PREFIX, DICTIONARY, NUMSTR, NOCOMPRESS
  - **COLLATE collation**  
Assigns a collation to the column (which must be of a collatable data type). If no collation is specified, the default collation is used.
  - **LIKE source\_table [ like\_option ... ]**  
Specifies a table from which the new table automatically copies all column names, their data types, and their not-null constraints.  
The new table and the source table are decoupled after creation is complete. Changes to the source table will not be applied to the new table, and it is not possible to include data of the new table in scans of the source table.  
Columns and constraints copied by **LIKE** are not merged with the same name. If the same name is specified explicitly or in another **LIKE** clause, an error is reported.
    - The default expressions are copied from the source table to the new table only if **INCLUDING DEFAULTS** is specified. The default behavior is to exclude default expressions, resulting in the copied columns in the new table having default values **NULL**.

- The **CHECK** constraints are copied from the source table to the new table only when **INCLUDING CONSTRAINTS** is specified. Other types of constraints are never copied to the new table. **NOT NULL** constraints are always copied to the new table. These rules also apply to column constraints and table constraints.
- Any indexes on the source table will not be created on the new table, unless the **INCLUDING INDEXES** clause is specified.
- **STORAGE** settings for the copied column definitions are copied only if **INCLUDING STORAGE** is specified. The default behavior is to exclude **STORAGE** settings.
- If **INCLUDING COMMENTS** is specified, comments for the copied columns, constraints, and indexes are copied. The default behavior is to exclude comments.
- If **INCLUDING PARTITION** is specified, the partition definitions of the source table are copied to the new table, and the new table no longer use the **PARTITION BY** clause. The default behavior is to exclude partition definition of the source table.
- If **INCLUDING REOPTIONS** is specified, the storage parameter (**WITH** clause of the source table) of the source table is copied to the new table. The default behavior is to exclude partition definition of the storage parameter of the source table.
- If **INCLUDING DISTRIBUTION** is specified, the distribution information of the source table is copied to the new table, including distribution type and column, and the new table no longer use the **DISTRIBUTE BY** clause. The default behavior is to exclude distribution information of the source table.
- **INCLUDING ALL** contains the meaning of **INCLUDING DEFAULTS**, **INCLUDING CONSTRAINTS**, **INCLUDING INDEXES**, **INCLUDING STORAGE**, **INCLUDING COMMENTS**, **INCLUDING PARTITION**, **INCLUDING REOPTIONS**, and **INCLUDING DISTRIBUTION**.
- If **EXCLUDING** is specified, the specified parameters are not included.

### NOTICE

- If the source table contains a sequence with the SERIAL, BIGSERIAL, or SMALLSERIAL data type, or a column in the source table is a sequence by default and the sequence is created for this table by using **CREATE SEQUENCE... OWNED BY**, these sequences will not be copied to the new table, and another sequence specific to the new table will be created. This is different from earlier versions. To share a sequence between the source table and new table, create a shared sequence (do not use **OWNED BY**) and set a column in the source table to this sequence.
- You are not advised to set a column in the source table to the sequence specific to another table especially when the table is distributed in specific Node Groups, because doing so may result in **CREATE TABLE ... LIKE** execution failures. In addition, doing so may cause the sequence to become invalid in the source sequence because the sequence will also be deleted from the source table when it is deleted from the table that the sequence is specific to. To share a sequence among multiple tables, you are advised to create a shared sequence for them.

- **WITH ( { storage\_parameter = value } [, ... ] )**

Specifies an optional storage parameter for a table or an index.

#### NOTE

Using Numeric of any precision to define column, specifies precision p and scale s. When precision and scale are not specified, the input will be displayed.

The description of parameters is as follows:

#### – FILLFACTOR

The fillfactor of a table is a percentage between 10 and 100. 100 (complete packing) is the default value. When a smaller fillfactor is specified, **INSERT** operations pack table pages only to the indicated percentage. The remaining space on each page is reserved for updating rows on that page. This gives **UPDATE** a chance to place the updated copy of a row on the same page, which is more efficient than placing it on a different page. For a table whose records are never updated, setting the fillfactor to 100 (complete packing) is the appropriate choice, but in heavily updated tables smaller fillfactors are appropriate. The parameter has no meaning for column-store tables.

Value range: 10–100

#### – ORIENTATION

Specifies the storage mode (row-store, column-store, or ) for table data. This parameter cannot be modified once it is set.

Valid value:

- **ROW** indicates that table data is stored in rows.  
**ROW** applies to OLTP service, which has many interactive transactions. An interaction involves many columns in the table. Using ROW can improve the efficiency.
- **COLUMN**, means the data will be stored in columns.

CLOUMN applies to data warehouse service, which has a large amount of aggregation computing, and involves a few column operations.

Default value:

If an ordinary tablespace is specified, the default is **ROW**.

– **COMPRESSION**

Specifies the compression level of the table data. It determines the compression ratio and time. Generally, the higher the level of compression, the higher the ratio, the longer the time, and the lower the level of compression, the lower the ratio, the shorter the time. The actual compression ratio depends on the distribution characteristics of loading table data.

Valid value:

- The valid values for column-store tables are **YES/NO** and **LOW/MIDDLE/HIGH**, and the default is **LOW**.
- The valid values for row-store tables are **YES** and **NO**, and the default is **NO**.

GaussDB(DWS) provides the following compression algorithms:

**Table 12-17** Compression algorithms for column-based storage

| COMPRESSION | NUMERIC                                                | STRING                               | INT                                                     |
|-------------|--------------------------------------------------------|--------------------------------------|---------------------------------------------------------|
| LOW         | Delta compression + RLE compression                    | LZ4 compression                      | Delta compression (RLE is optional.)                    |
| MIDDLE      | Delta compression + RLE compression + LZ4 compression  | dict compression or LZ4 compression  | Delta compression or LZ4 compression (RLE is optional)  |
| HIGH        | Delta compression + RLE compression + zlib compression | dict compression or zlib compression | Delta compression or zlib compression (RLE is optional) |

– **COMPRESSLEVEL**

Specifies the compression level of the table data. It determines the compression ratio and time. This divides a compression level into sublevels, providing you with more choices for compression rate and duration. As the value becomes greater, the compression rate becomes higher and duration longer at the same compression level. The parameter is only valid for column-store table.

Value range: 0 to 3. The default value is **0**.

- **MAX\_BATCHROW**  
Specifies the maximum of a storage unit during data loading process. The parameter is only valid for column-store table.  
Value range: 10000 to 60000
- **PARTIAL\_CLUSTER\_ROWS**  
Specifies the number of records to be partial cluster stored during data loading process. The parameter is only valid for column-store table.  
Value range: 600000 to 2147483647
- **DELTAROW\_THRESHOLD**  
Specifies the upper limit of to-be-imported rows for triggering the data import to a delta table when data is to be imported to a column-store table. This parameter takes effect only if `enable_delta_store` is set to **on**. The parameter is only valid for column-store table.  
The value ranges from **0** to **9999**. The default value is **100**.
- **VERSION**  
Specifies the version of ORC storage format.  
Value range: 0.12. ORC 0.12 format is supported currently. More formats will be supported as the development of ORC format.  
Default value: 0.12
- **ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP }**  
**ON COMMIT** determines what to do when you commit a temporary table creation operation. The three options are as follows. Currently, only **PRESERVE ROWS** and **DELETE ROWS** can be used.
  - **PRESERVE ROWS** (Default): No special action is taken at the ends of transactions. The temporary table and its table data are unchanged.
  - **DELETE ROWS**: All rows in the temporary table will be deleted at the end of each transaction block.
  - **DROP**: The temporary table will be dropped at the end of the current transaction block.
- **COMPRESS | NOCOMPRESS**  
If you specify **COMPRESS** in the **CREATE TABLE** statement, the compression feature is triggered in the case of a bulk **INSERT** operation. If this feature is enabled, a scan is performed for all tuple data within the page to generate a dictionary and then the tuple data is compressed and stored. If **NOCOMPRESS** is specified, the table is not compressed.  
Default value: **NOCOMPRESS**, tuple data is not compressed before storage.
- **DISTRIBUTE BY**  
Specifies how the table is distributed or replicated between DNs.  
Valid value:
  - **REPLICATION**: Each row in the table exists on all DNs, that is, each DN has complete table data.
  - **HASH (column\_name)**: Each row of the table will be placed into all the DNs based on the hash value of the specified column.

 NOTE

- When **DISTRIBUTE BY HASH (column\_name)** is specified, the primary key and its unique index must contain the **column\_name** column.
- When **DISTRIBUTE BY HASH (column\_name)** in a referenced table is specified, the foreign key of the reference table must contain the **column\_name** column.

Default value: **HASH(column\_name)**, the key column of **column\_name** (if any) or the column of distribution column supported by first data type.

**column\_name** supports the following data types:

- Integer types: TINYINT, SMALLINT, INT, BIGINT, and NUMERIC/DECIMAL
- Character types: CHAR, BPCHAR, VARCHAR, VARCHAR2, NVARCHAR2, and TEXT
- Date/time types: DATE, TIME, TIMETZ, TIMESTAMP, TIMESTAMPTZ, INTERVAL, and SMALLDATETIME

 **NOTE**

When you create a table, the choices of distribution keys and partition keys have major impact on SQL query performance. Therefore, choosing proper distribution column and partition key with strategies.

- **Selecting an Appropriate Distribution Column**

In the data distributed table using Hash, an appropriate distributed array should be used to distribute and store data on multiple DNs evenly, preventing data skew (uneven data distribution across several DNs). Determine the proper distribution column based on the following principles:

1. Determine whether data is skewed.

Connect to the database and run the following statements to check the number of tuples on each DN: Replace *tablename* with the actual name of the table to be analyzed.

```
SELECT a.count,b.node_name FROM (SELECT count(*) AS count,xc_node_id FROM
tablename GROUP BY xc_node_id) a, pgxc_node b WHERE a.xc_node_id=b.node_id
ORDER BY a.count DESC;
```

If tuple numbers vary greatly (several times or tenfold) in each DN, a data skew occurs. Change the data distribution key based on the following principles:

2. Recreate a table to change its distribution keys. **ALTER TABLE** cannot change distribution keys. Therefore, you need to recreate a table when changing its distribution keys.

Principles for selecting distribution keys are as follows:

The column value of the distribution column should be discrete so that data can be evenly distributed on each DN. For example, you are advised to select the primary key of a table as the distribution column, and the ID card number as the distribution column in a personnel information table.

With the above principles met, you can select join conditions as distribution keys so that join tasks can be pushed down to DNs, reducing the amount of data transferred between the DNs.

- **Selecting appropriate partition keys**

In range partitioning, the table is partitioned into ranges defined by a key column or set of columns, with no overlap between the ranges of values assigned to different partitions. Each range has a dedicated partition for data storage.

Modify partition keys to make the query result stored in the same or least partitions (partition pruning). Obtaining consecutive I/O to improve the query performance.

In actual services, time is used to filter query objects. Therefore, you can use time as a partition key, and change the key value based on the total data volume and single data query volume.

- **TO { GROUP groupname | NODE ( nodename [, ... ] ) }**

**TO GROUP** specifies the Node Group in which the table is created. Currently, it cannot be used for HDFS tables. **TO NODE** is used for internal scale-out tools.

- **CONSTRAINT constraint\_name**

Specifies a name for a column or table constraint. The optional constraint clauses specify constraints that new or updated rows must satisfy for an insert or update operation to succeed.

There are two ways to define constraints:

- A column constraint is defined as part of a column definition, and it is bound to a particular column.

- A table constraint is not bound to any particular columns but can apply to more than one column.
- **NOT NULL**  
Indicates that the column is not allowed to contain **NULL** values.
- **NULL**  
The column is allowed to contain **NULL** values. This is the default setting. This clause is only provided for compatibility with non-standard SQL databases. You are advised not to use this clause.
- **CHECK ( expression )**  
Specifies an expression producing a Boolean result which new or updated rows must satisfy for an insert or update operation to succeed. Expressions evaluating to **TRUE** or **UNKNOWN** succeed. If any row of an insert or update operation produces a **FALSE** result, an error exception is raised and the insert or update does not alter the database.  
A check constraint specified as a column constraint should reference only the column's values, while an expression appearing in a table constraint can reference multiple columns.

 **NOTE**

<>**NULL** and **!=NULL** are invalid in an expression. Change them to **IS NOT NULL**.

- **DEFAULT default\_expr**  
Assigns a default data value for a column. The value can be any variable-free expressions (Subqueries and cross-references to other columns in the current table are not allowed). The data type of the default expression must match the data type of the column.  
The default expression will be used in any insert operation that does not specify a value for the column. If there is no default value for a column, then the default value is **NULL**.
- **UNIQUE index\_parameters**  
**UNIQUE ( column\_name [, ... ] ) index\_parameters**  
Specifies that a group of one or more columns of a table can contain only unique values.  
For the purpose of a unique constraint, **NULL** is not considered equal.

 **NOTE**

If **DISTRIBUTE BY REPLICATION** is not specified, the column table that contains only unique values must contain distribution columns.

- **PRIMARY KEY index\_parameters**  
**PRIMARY KEY ( column\_name [, ... ] ) index\_parameters**  
Specifies the primary key constraint specifies that a column or columns of a table can contain only unique (non-duplicate) and non-null values.  
Only one primary key can be specified for a table.

 **NOTE**

If **DISTRIBUTE BY REPLICATION** is not specified, the column set with a primary key constraint must contain distributed columns.



- **DEFERRABLE | NOT DEFERRABLE**

Controls whether the constraint can be deferred. A constraint that is not deferrable will be checked immediately after every command. Checking of constraints that are deferrable can be postponed until the end of the transaction using the **SET CONSTRAINTS** command. **NOT DEFERRABLE** is the default value. Currently, only **UNIQUE** and **PRIMARY KEY** constraints accept this clause. All the other constraints are not deferrable.

- **PARTIAL CLUSTER KEY**

Specifies a partial cluster key for storage. When importing data to a column-store table, you can perform local data sorting by specified columns (single or multiple).

- **INITIALLY IMMEDIATE | INITIALLY DEFERRED**

If a constraint is deferrable, this clause specifies the default time to check the constraint.

- If the constraint is **INITIALLY IMMEDIATE** (default value), it is checked after each statement.
- If the constraint is **INITIALLY DEFERRED**, it is checked only at the end of the transaction.

The constraint check time can be altered using the **SET CONSTRAINTS** command.

## Examples

```
-- Create a simple table:
CREATE TABLE tpcds.warehouse_t1
(
  W_WAREHOUSE_SK      INTEGER      NOT NULL,
  W_WAREHOUSE_ID      CHAR(16)     NOT NULL,
  W_WAREHOUSE_NAME    VARCHAR(20)
  W_WAREHOUSE_SQ_FT   INTEGER
  W_STREET_NUMBER     CHAR(10)
  W_STREET_NAME       VARCHAR(60)
  W_STREET_TYPE       CHAR(15)
  W_SUITE_NUMBER      CHAR(10)
  W_CITY              VARCHAR(60)
  W_COUNTY            VARCHAR(30)
  W_STATE             CHAR(2)
  W_ZIP              CHAR(10)
  W_COUNTRY           VARCHAR(20)
  W_GMT_OFFSET        DECIMAL(5,2)
);

CREATE TABLE tpcds.warehouse_t2
(
  W_WAREHOUSE_SK      INTEGER      NOT NULL,
  W_WAREHOUSE_ID      CHAR(16)     NOT NULL,
  W_WAREHOUSE_NAME    VARCHAR(20)
  W_WAREHOUSE_SQ_FT   INTEGER
  W_STREET_NUMBER     CHAR(10)
  W_STREET_NAME       VARCHAR(60)  DICTIONARY,
  W_STREET_TYPE       CHAR(15)
  W_SUITE_NUMBER      CHAR(10)
  W_CITY              VARCHAR(60)
  W_COUNTY            VARCHAR(30)
  W_STATE             CHAR(2)
  W_ZIP              CHAR(10)
  W_COUNTRY           VARCHAR(20)
  W_GMT_OFFSET        DECIMAL(5,2)
);
```

```
-- Create a table and set the default value of the W_STATE column to GA:
CREATE TABLE tpcds.warehouse_t3
(
  W_WAREHOUSE_SK      INTEGER          NOT NULL,
  W_WAREHOUSE_ID      CHAR(16)         NOT NULL,
  W_WAREHOUSE_NAME    VARCHAR(20)      ,
  W_WAREHOUSE_SQ_FT   INTEGER          ,
  W_STREET_NUMBER     CHAR(10)         ,
  W_STREET_NAME       VARCHAR(60)     ,
  W_STREET_TYPE       CHAR(15)        ,
  W_SUITE_NUMBER      CHAR(10)        ,
  W_CITY              VARCHAR(60)     ,
  W_COUNTY            VARCHAR(30)     ,
  W_STATE             CHAR(2)         DEFAULT 'GA',
  W_ZIP              CHAR(10)        ,
  W_COUNTRY           VARCHAR(20)     ,
  W_GMT_OFFSET        DECIMAL(5,2)
);

-- Create a table and check whether the W_WAREHOUSE_NAME column is unique at the end of its
creation:
CREATE TABLE tpcds.warehouse_t4
(
  W_WAREHOUSE_SK      INTEGER          NOT NULL,
  W_WAREHOUSE_ID      CHAR(16)         NOT NULL,
  W_WAREHOUSE_NAME    VARCHAR(20)     UNIQUE DEFERRABLE,
  W_WAREHOUSE_SQ_FT   INTEGER          ,
  W_STREET_NUMBER     CHAR(10)         ,
  W_STREET_NAME       VARCHAR(60)     ,
  W_STREET_TYPE       CHAR(15)        ,
  W_SUITE_NUMBER      CHAR(10)        ,
  W_CITY              VARCHAR(60)     ,
  W_COUNTY            VARCHAR(30)     ,
  W_STATE             CHAR(2)         ,
  W_ZIP              CHAR(10)        ,
  W_COUNTRY           VARCHAR(20)     ,
  W_GMT_OFFSET        DECIMAL(5,2)
);

-- Create a table with its fillfactor set to 70%:
CREATE TABLE tpcds.warehouse_t5
(
  W_WAREHOUSE_SK      INTEGER          NOT NULL,
  W_WAREHOUSE_ID      CHAR(16)         NOT NULL,
  W_WAREHOUSE_NAME    VARCHAR(20)      ,
  W_WAREHOUSE_SQ_FT   INTEGER          ,
  W_STREET_NUMBER     CHAR(10)         ,
  W_STREET_NAME       VARCHAR(60)     ,
  W_STREET_TYPE       CHAR(15)        ,
  W_SUITE_NUMBER      CHAR(10)        ,
  W_CITY              VARCHAR(60)     ,
  W_COUNTY            VARCHAR(30)     ,
  W_STATE             CHAR(2)         ,
  W_ZIP              CHAR(10)        ,
  W_COUNTRY           VARCHAR(20)     ,
  W_GMT_OFFSET        DECIMAL(5,2),
  UNIQUE(W_WAREHOUSE_NAME) WITH(fillfactor=70)
);

-- An alternative for the preceding syntax is as follows:
CREATE TABLE tpcds.warehouse_t6
(
  W_WAREHOUSE_SK      INTEGER          NOT NULL,
  W_WAREHOUSE_ID      CHAR(16)         NOT NULL,
  W_WAREHOUSE_NAME    VARCHAR(20)     UNIQUE,
  W_WAREHOUSE_SQ_FT   INTEGER          ,
  W_STREET_NUMBER     CHAR(10)         ,
  W_STREET_NAME       VARCHAR(60)     ,
  W_STREET_TYPE       CHAR(15)        ,
```

```

W_SUITE_NUMBER      CHAR(10)
W_CITY              VARCHAR(60)
W_COUNTY            VARCHAR(30)
W_STATE             CHAR(2)
W_ZIP               CHAR(10)
W_COUNTRY            VARCHAR(20)
W_GMT_OFFSET        DECIMAL(5,2)
) WITH(fillfactor=70);

-- Create a table and specify that its data is not written to WALs:
CREATE UNLOGGED TABLE tpcds.warehouse_t7
(
  W_WAREHOUSE_SK      INTEGER          NOT NULL,
  W_WAREHOUSE_ID      CHAR(16)         NOT NULL,
  W_WAREHOUSE_NAME     VARCHAR(20)
  W_WAREHOUSE_SQ_FT    INTEGER
  W_STREET_NUMBER     CHAR(10)
  W_STREET_NAME        VARCHAR(60)
  W_STREET_TYPE        CHAR(15)
  W_SUITE_NUMBER       CHAR(10)
  W_CITY              VARCHAR(60)
  W_COUNTY            VARCHAR(30)
  W_STATE             CHAR(2)
  W_ZIP               CHAR(10)
  W_COUNTRY            VARCHAR(20)
  W_GMT_OFFSET        DECIMAL(5,2)
);

-- Create a temporary table:
CREATE TEMPORARY TABLE warehouse_t24
(
  W_WAREHOUSE_SK      INTEGER          NOT NULL,
  W_WAREHOUSE_ID      CHAR(16)         NOT NULL,
  W_WAREHOUSE_NAME     VARCHAR(20)
  W_WAREHOUSE_SQ_FT    INTEGER
  W_STREET_NUMBER     CHAR(10)
  W_STREET_NAME        VARCHAR(60)
  W_STREET_TYPE        CHAR(15)
  W_SUITE_NUMBER       CHAR(10)
  W_CITY              VARCHAR(60)
  W_COUNTY            VARCHAR(30)
  W_STATE             CHAR(2)
  W_ZIP               CHAR(10)
  W_COUNTRY            VARCHAR(20)
  W_GMT_OFFSET        DECIMAL(5,2)
);

-- Create a temporary table in a transaction and specify that data of this table is deleted when the
transaction is committed:
CREATE TEMPORARY TABLE warehouse_t25
(
  W_WAREHOUSE_SK      INTEGER          NOT NULL,
  W_WAREHOUSE_ID      CHAR(16)         NOT NULL,
  W_WAREHOUSE_NAME     VARCHAR(20)
  W_WAREHOUSE_SQ_FT    INTEGER
  W_STREET_NUMBER     CHAR(10)
  W_STREET_NAME        VARCHAR(60)
  W_STREET_TYPE        CHAR(15)
  W_SUITE_NUMBER       CHAR(10)
  W_CITY              VARCHAR(60)
  W_COUNTY            VARCHAR(30)
  W_STATE             CHAR(2)
  W_ZIP               CHAR(10)
  W_COUNTRY            VARCHAR(20)
  W_GMT_OFFSET        DECIMAL(5,2)
) ON COMMIT DELETE ROWS;

-- Create a table and specify that no error is reported for duplicate tables (if any):
CREATE TABLE IF NOT EXISTS tpcds.warehouse_t8

```

```
(
W_WAREHOUSE_SK      INTEGER      NOT NULL,
W_WAREHOUSE_ID      CHAR(16)      NOT NULL,
W_WAREHOUSE_NAME     VARCHAR(20)
W_WAREHOUSE_SQ_FT    INTEGER
W_STREET_NUMBER      CHAR(10)
W_STREET_NAME        VARCHAR(60)
W_STREET_TYPE        CHAR(15)
W_SUITE_NUMBER       CHAR(10)
W_CITY               VARCHAR(60)
W_COUNTY             VARCHAR(30)
W_STATE              CHAR(2)
W_ZIP                CHAR(10)
W_COUNTRY            VARCHAR(20)
W_GMT_OFFSET         DECIMAL(5,2)
);
```

-- Create a table with a primary key constraint:

CREATE TABLE tpcds.warehouse\_t11

```
(
W_WAREHOUSE_SK      INTEGER      PRIMARY KEY,
W_WAREHOUSE_ID      CHAR(16)      NOT NULL,
W_WAREHOUSE_NAME     VARCHAR(20)
W_WAREHOUSE_SQ_FT    INTEGER
W_STREET_NUMBER      CHAR(10)
W_STREET_NAME        VARCHAR(60)
W_STREET_TYPE        CHAR(15)
W_SUITE_NUMBER       CHAR(10)
W_CITY               VARCHAR(60)
W_COUNTY             VARCHAR(30)
W_STATE              CHAR(2)
W_ZIP                CHAR(10)
W_COUNTRY            VARCHAR(20)
W_GMT_OFFSET         DECIMAL(5,2)
);
```

-- An alternative for the preceding syntax is as follows:

CREATE TABLE tpcds.warehouse\_t12

```
(
W_WAREHOUSE_SK      INTEGER      NOT NULL,
W_WAREHOUSE_ID      CHAR(16)      NOT NULL,
W_WAREHOUSE_NAME     VARCHAR(20)
W_WAREHOUSE_SQ_FT    INTEGER
W_STREET_NUMBER      CHAR(10)
W_STREET_NAME        VARCHAR(60)
W_STREET_TYPE        CHAR(15)
W_SUITE_NUMBER       CHAR(10)
W_CITY               VARCHAR(60)
W_COUNTY             VARCHAR(30)
W_STATE              CHAR(2)
W_ZIP                CHAR(10)
W_COUNTRY            VARCHAR(20)
W_GMT_OFFSET         DECIMAL(5,2),
PRIMARY KEY(W_WAREHOUSE_SK)
);
```

-- Or use the following statement to create table films with a primary key constraint and specify the name of the constraint:

CREATE TABLE tpcds.warehouse\_t13

```
(
W_WAREHOUSE_SK      INTEGER      NOT NULL,
W_WAREHOUSE_ID      CHAR(16)      NOT NULL,
W_WAREHOUSE_NAME     VARCHAR(20)
W_WAREHOUSE_SQ_FT    INTEGER
W_STREET_NUMBER      CHAR(10)
W_STREET_NAME        VARCHAR(60)
W_STREET_TYPE        CHAR(15)
W_SUITE_NUMBER       CHAR(10)
W_CITY               VARCHAR(60)
);
```

```

W_COUNTY          VARCHAR(30)
W_STATE           CHAR(2)
W_ZIP             CHAR(10)
W_COUNTRY         VARCHAR(20)
W_GMT_OFFSET      DECIMAL(5,2),
CONSTRAINT W_CSTR_KEY1 PRIMARY KEY(W_WAREHOUSE_SK)
);

-- Create a table with a compound primary key constraint:
CREATE TABLE tpcds.warehouse_t14
(
  W_WAREHOUSE_SK    INTEGER          NOT NULL,
  W_WAREHOUSE_ID    CHAR(16)         NOT NULL,
  W_WAREHOUSE_NAME  VARCHAR(20)
  W_WAREHOUSE_SQ_FT INTEGER
  W_STREET_NUMBER  CHAR(10)
  W_STREET_NAME     VARCHAR(60)
  W_STREET_TYPE    CHAR(15)
  W_SUITE_NUMBER   CHAR(10)
  W_CITY           VARCHAR(60)
  W_COUNTY         VARCHAR(30)
  W_STATE          CHAR(2)
  W_ZIP            CHAR(10)
  W_COUNTRY        VARCHAR(20)
  W_GMT_OFFSET     DECIMAL(5,2),
  CONSTRAINT W_CSTR_KEY2 PRIMARY KEY(W_WAREHOUSE_SK, W_WAREHOUSE_ID)
);

-- Create a column-store table.
CREATE TABLE tpcds.warehouse_t15
(
  W_WAREHOUSE_SK    INTEGER          NOT NULL,
  W_WAREHOUSE_ID    CHAR(16)         NOT NULL,
  W_WAREHOUSE_NAME  VARCHAR(20)
  W_WAREHOUSE_SQ_FT INTEGER
  W_STREET_NUMBER  CHAR(10)
  W_STREET_NAME     VARCHAR(60)
  W_STREET_TYPE    CHAR(15)
  W_SUITE_NUMBER   CHAR(10)
  W_CITY           VARCHAR(60)
  W_COUNTY         VARCHAR(30)
  W_STATE          CHAR(2)
  W_ZIP            CHAR(10)
  W_COUNTRY        VARCHAR(20)
  W_GMT_OFFSET     DECIMAL(5,2)
) WITH (ORIENTATION = COLUMN);

-- Create a column-store table using partial clustered storage:
CREATE TABLE tpcds.warehouse_t16
(
  W_WAREHOUSE_SK    INTEGER          NOT NULL,
  W_WAREHOUSE_ID    CHAR(16)         NOT NULL,
  W_WAREHOUSE_NAME  VARCHAR(20)
  W_WAREHOUSE_SQ_FT INTEGER
  W_STREET_NUMBER  CHAR(10)
  W_STREET_NAME     VARCHAR(60)
  W_STREET_TYPE    CHAR(15)
  W_SUITE_NUMBER   CHAR(10)
  W_CITY           VARCHAR(60)
  W_COUNTY         VARCHAR(30)
  W_STATE          CHAR(2)
  W_ZIP            CHAR(10)
  W_COUNTRY        VARCHAR(20)
  W_GMT_OFFSET     DECIMAL(5,2),
  PARTIAL CLUSTER KEY(W_WAREHOUSE_SK, W_WAREHOUSE_ID)
) WITH (ORIENTATION = COLUMN);

-- Define a column-store table with compression enabled:
CREATE TABLE tpcds.warehouse_t17

```

```
(
W_WAREHOUSE_SK      INTEGER      NOT NULL,
W_WAREHOUSE_ID      CHAR(16)      NOT NULL,
W_WAREHOUSE_NAME     VARCHAR(20)
W_WAREHOUSE_SQ_FT    INTEGER
W_STREET_NUMBER     CHAR(10)
W_STREET_NAME        VARCHAR(60)
W_STREET_TYPE        CHAR(15)
W_SUITE_NUMBER       CHAR(10)
W_CITY               VARCHAR(60)
W_COUNTY             VARCHAR(30)
W_STATE              CHAR(2)
W_ZIP                CHAR(10)
W_COUNTRY            VARCHAR(20)
W_GMT_OFFSET         DECIMAL(5,2)
) WITH (ORIENTATION = COLUMN, COMPRESSION=HIGH);

-- Define a table with compression enabled:
CREATE TABLE tpcds.warehouse_t18
(
W_WAREHOUSE_SK      INTEGER      NOT NULL,
W_WAREHOUSE_ID      CHAR(16)      NOT NULL,
W_WAREHOUSE_NAME     VARCHAR(20)
W_WAREHOUSE_SQ_FT    INTEGER
W_STREET_NUMBER     CHAR(10)
W_STREET_NAME        VARCHAR(60)
W_STREET_TYPE        CHAR(15)
W_SUITE_NUMBER       CHAR(10)
W_CITY               VARCHAR(60)
W_COUNTY             VARCHAR(30)
W_STATE              CHAR(2)
W_ZIP                CHAR(10)
W_COUNTRY            VARCHAR(20)
W_GMT_OFFSET         DECIMAL(5,2)
) COMPRESS;

-- Define a check column constraint:
CREATE TABLE tpcds.warehouse_t19
(
W_WAREHOUSE_SK      INTEGER      PRIMARY KEY CHECK (W_WAREHOUSE_SK > 0),
W_WAREHOUSE_ID      CHAR(16)      NOT NULL,
W_WAREHOUSE_NAME     VARCHAR(20)  CHECK (W_WAREHOUSE_NAME <> ''),
W_WAREHOUSE_SQ_FT    INTEGER
W_STREET_NUMBER     CHAR(10)
W_STREET_NAME        VARCHAR(60)
W_STREET_TYPE        CHAR(15)
W_SUITE_NUMBER       CHAR(10)
W_CITY               VARCHAR(60)
W_COUNTY             VARCHAR(30)
W_STATE              CHAR(2)
W_ZIP                CHAR(10)
W_COUNTRY            VARCHAR(20)
W_GMT_OFFSET         DECIMAL(5,2)
);

CREATE TABLE tpcds.warehouse_t20
(
W_WAREHOUSE_SK      INTEGER      PRIMARY KEY,
W_WAREHOUSE_ID      CHAR(16)      NOT NULL,
W_WAREHOUSE_NAME     VARCHAR(20)  CHECK (W_WAREHOUSE_NAME <> ''),
W_WAREHOUSE_SQ_FT    INTEGER
W_STREET_NUMBER     CHAR(10)
W_STREET_NAME        VARCHAR(60)
W_STREET_TYPE        CHAR(15)
W_SUITE_NUMBER       CHAR(10)
W_CITY               VARCHAR(60)
W_COUNTY             VARCHAR(30)
W_STATE              CHAR(2)
W_ZIP                CHAR(10)
);
```

```
W_COUNTRY          VARCHAR(20)          ,
W_GMT_OFFSET       DECIMAL(5,2),
CONSTRAINT W_CONSTR_KEY2 CHECK(W_WAREHOUSE_SK > 0 AND W_WAREHOUSE_NAME <> ")
);

-- Define a table with each of its rows stored in all DNs:
CREATE TABLE tpcds.warehouse_t21
(
  W_WAREHOUSE_SK    INTEGER          NOT NULL,
  W_WAREHOUSE_ID    CHAR(16)         NOT NULL,
  W_WAREHOUSE_NAME  VARCHAR(20)      ,
  W_WAREHOUSE_SQ_FT INTEGER          ,
  W_STREET_NUMBER   CHAR(10)         ,
  W_STREET_NAME     VARCHAR(60)      ,
  W_STREET_TYPE     CHAR(15)         ,
  W_SUITE_NUMBER    CHAR(10)         ,
  W_CITY            VARCHAR(60)      ,
  W_COUNTRY         VARCHAR(30)      ,
  W_STATE           CHAR(2)          ,
  W_ZIP             CHAR(10)         ,
  W_COUNTRY         VARCHAR(20)      ,
  W_GMT_OFFSET      DECIMAL(5,2)
)DISTRIBUTE BY REPLICATION;

-- Define a hash table:
CREATE TABLE tpcds.warehouse_t22
(
  W_WAREHOUSE_SK    INTEGER          NOT NULL,
  W_WAREHOUSE_ID    CHAR(16)         NOT NULL,
  W_WAREHOUSE_NAME  VARCHAR(20)      ,
  W_WAREHOUSE_SQ_FT INTEGER          ,
  W_STREET_NUMBER   CHAR(10)         ,
  W_STREET_NAME     VARCHAR(60)      ,
  W_STREET_TYPE     CHAR(15)         ,
  W_SUITE_NUMBER    CHAR(10)         ,
  W_CITY            VARCHAR(60)      ,
  W_COUNTRY         VARCHAR(30)      ,
  W_STATE           CHAR(2)          ,
  W_ZIP             CHAR(10)         ,
  W_COUNTRY         VARCHAR(20)      ,
  W_GMT_OFFSET      DECIMAL(5,2),
  CONSTRAINT W_CONSTR_KEY3 UNIQUE(W_WAREHOUSE_SK)
)DISTRIBUTE BY HASH(W_WAREHOUSE_SK);

-- Add a varchar column to the tpcds.warehouse_t19 table:
ALTER TABLE tpcds.warehouse_t19 ADD W_GOODS_CATEGORY varchar(30);

-- Add a check constraint to the tpcds.warehouse_t19 table:
ALTER TABLE tpcds.warehouse_t19 ADD CONSTRAINT W_CONSTR_KEY4 CHECK (W_STATE <> ");

-- Use one statement to alter the types of two existing columns:
ALTER TABLE tpcds.warehouse_t19
  ALTER COLUMN W_GOODS_CATEGORY TYPE varchar(80),
  ALTER COLUMN W_STREET_NAME TYPE varchar(100);

-- This statement is equivalent to the preceding statement.
ALTER TABLE tpcds.warehouse_t19 MODIFY (W_GOODS_CATEGORY varchar(30), W_STREET_NAME
varchar(60));

-- Add a Not-Null constraint to an existing column:
ALTER TABLE tpcds.warehouse_t19 ALTER COLUMN W_GOODS_CATEGORY SET NOT NULL;

-- Remove Not-Null constraints from an existing column:
ALTER TABLE tpcds.warehouse_t19 ALTER COLUMN W_GOODS_CATEGORY DROP NOT NULL;

-- If no partial cluster is specified in a column-store table, add a partial cluster to the table:
ALTER TABLE tpcds.warehouse_t17 ADD PARTIAL CLUSTER KEY(W_WAREHOUSE_SK);

-- View the constraint name and delete the partial cluster column of a column-store table:
```

```

\d+ tpcds.warehouse_t17
      Table "tpcds.warehouse_t17"
  Column      | Type          | Modifiers | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----
w_warehouse_sk | integer       | not null  | plain   |              |
w_warehouse_id | character(16) | not null  | extended|              |
w_warehouse_name | character varying(20) |          | extended|              |
w_warehouse_sq_ft | integer      |          | plain   |              |
w_street_number | character(10) |          | extended|              |
w_street_name   | character varying(60) |          | extended|              |
w_street_type   | character(15) |          | extended|              |
w_suite_number  | character(10) |          | extended|              |
w_city          | character varying(60) |          | extended|              |
w_county        | character varying(30) |          | extended|              |
w_state         | character(2)  |          | extended|              |
w_zip           | character(10) |          | extended|              |
w_country       | character varying(20) |          | extended|              |
w_gmt_offset    | numeric(5,2) |          | main    |              |
Partial Cluster :
  "warehouse_t17_cluster" PARTIAL CLUSTER KEY (w_warehouse_sk)
Has OIDs: no
Distribute By: HASH(w_warehouse_sk)
Location Nodes: ALL DATANODES
Options: orientation=orc, compression=no, version=0.12
ALTER TABLE tpcds.warehouse_t17 DROP CONSTRAINT warehouse_t17_cluster;

-- Create the joe schema:
CREATE SCHEMA joe;

-- Move a table to another schema:
ALTER TABLE tpcds.warehouse_t19 SET SCHEMA joe;

-- Rename an existing table:
ALTER TABLE joe.warehouse_t19 RENAME TO warehouse_t23;

-- Delete a column from the warehouse_t23 table:
ALTER TABLE joe.warehouse_t23 DROP COLUMN W_STREET_NAME;

-- Delete the tables warehouse.
DROP TABLE tpcds.warehouse_t1;
DROP TABLE tpcds.warehouse_t2;
DROP TABLE tpcds.warehouse_t3;
DROP TABLE tpcds.warehouse_t4;
DROP TABLE tpcds.warehouse_t5;
DROP TABLE tpcds.warehouse_t6;
DROP TABLE tpcds.warehouse_t7;
DROP TABLE tpcds.warehouse_t8;
DROP TABLE tpcds.warehouse_t9;
DROP TABLE tpcds.warehouse_t10;
DROP TABLE tpcds.warehouse_t11;
DROP TABLE tpcds.warehouse_t12;
DROP TABLE tpcds.warehouse_t13;
DROP TABLE tpcds.warehouse_t14;
DROP TABLE tpcds.warehouse_t15;
DROP TABLE tpcds.warehouse_t16;

```

## Helpful Links

[ALTER TABLE, DROP TABLE](#)

# 12.51 CREATE TABLE AS

## Function

**CREATE TABLE AS** creates a table based on the results of a query.



It creates a table and fills it with data obtained using **SELECT**. The table columns have the names and data types associated with the output columns of the **SELECT**. Except that you can override the **SELECT** output column names by giving an explicit list of new column names.

**CREATE TABLE AS** queries once the source table and writes data in the new table. The query result view changes when the source table changes. In contrast, a view re-evaluates its defining **SELECT** statement whenever it is queried.

## Precautions

- This command cannot be used to create a partitioned table.
- If an error occurs when you create a table, after the system is recovered, the system probably cannot automatically clear the created disk file whose size is not 0. This problem seldom occurs.

## Syntax

```
CREATE [ UNLOGGED ] TABLE table_name
  [ ( column_name [, ... ] ) ]
  [ WITH ( {storage_parameter = value} [, ... ] ) ]
  [ COMPRESS | NOCOMPRESS ]

  [ DISTRIBUTE BY { REPLICATION | { [HASH] ( column_name ) } } ]

AS query
[ WITH [ NO ] DATA ];
```

## Parameter Description

- **UNLOGGED**  
Specifies that the table is created as an unlogged table. Data written to unlogged tables is not written to the write-ahead log, which makes them considerably faster than ordinary tables. However, they are not crash-safe: an unlogged table is automatically truncated after a crash or unclean shutdown. The contents of an unlogged table are also not replicated to standby servers. Any indexes created on an unlogged table are automatically unlogged as well.
  - Usage scenario: Unlogged tables do not ensure safe data. Users can back up data before using unlogged tables; for example, users should back up the data before a system upgrade.
  - Troubleshooting: If data is missing in the indexes of unlogged tables due to some unexpected operations such as an unclean shutdown, users should re-create the indexes with errors.
- **table\_name**  
Specifies the name of the table to be created.  
Value range: a string. It must comply with the naming convention.
- **column\_name**  
Specifies the name of a column to be created in the new table.  
Value range: a string. It must comply with the naming convention.
- **WITH ( storage\_parameter [= value] [, ... ] )**  
Specifies an optional storage parameter for a table or an index. See details of parameters below.

- **FILLFACTOR**  
The fillfactor of a table is a percentage between 10 and 100. 100 (complete packing) is the default value. When a smaller fillfactor is specified, **INSERT** operations pack table pages only to the indicated percentage. The remaining space on each page is reserved for updating rows on that page. This gives **UPDATE** a chance to place the updated copy of a row on the same page, which is more efficient than placing it on a different page. For a table whose records are never updated, setting the fillfactor to 100 (complete packing) is the appropriate choice, but in heavily updated tables smaller fillfactors are appropriate. The parameter is only valid for row-store table.  
Value range: 10–100
- **ORIENTATION**  
Valid value:  
**COLUMN**: The data will be stored in columns.  
**ROW** (default value): The data will be stored in rows.
- **COMPRESSION**  
Specifies the compression level of the table data. It determines the compression ratio and time. Generally, the higher the level of compression, the higher the ratio, the longer the time, and the lower the level of compression, the lower the ratio, the shorter the time. The actual compression ratio depends on the distribution characteristics of loading table data.  
Valid value:  
The valid values for column-store tables are **YES/NO** and **LOW/MIDDLE/HIGH**, and the default is **LOW**.  
The valid values for row-store tables are **YES** and **NO**, and the default is **NO**.
- **MAX\_BATCHROW**  
Specifies the maximum of a storage unit during data loading process. The parameter is only valid for column-store table.  
Value range: 10000 to 60000
- **PARTIAL\_CLUSTER\_ROWS**  
Specifies the number of records to be partial cluster stored during data loading process. The parameter is only valid for column-store table.  
Value range: 600000 to 2147483647
- **COMPRESS / NOCOMPRESS**  
Specifies the keyword **COMPRESS** during the creation of a table, so that the compression feature is triggered in the case of a bulk **INSERT** operation. If this feature is enabled, a scan is performed for all tuple data within the page to generate a dictionary and then the tuple data is compressed and stored. If **NOCOMPRESS** is specified, the table is not compressed.  
Default value: **NOCOMPRESS**, tuple data is not compressed before storage.
- **DISTRIBUTE BY**  
Specifies how the table is distributed or replicated between DNs.
  - **REPLICATION**: Each row in the table exists on all DNs, that is, each DN has complete table data.

- **HASH (column\_name)**: Each row of the table will be placed into all the DNs based on the hash value of the specified column.

---

**NOTICE**

- When **DISTRIBUTE BY HASH (column\_name)** is specified, the primary key and its unique index must contain the **column\_name** column.
- When **DISTRIBUTE BY HASH (column\_name)** in a referenced table is specified, the foreign key of the reference table must contain the **column\_name** column.

---

Default value: **HASH(column\_name)**, the key column of **column\_name** (if any) or the column of distribution column supported by first data type.

**column\_name** supports the following data types:

- Integer types: TINYINT, SMALLINT, INT, BIGINT, and NUMERIC/DECIMAL
  - Character types: CHAR, BPCHAR, VARCHAR, VARCHAR2, and NVARCHAR2
  - Date/time types: DATE, TIME, TIMETZ, TIMESTAMP, TIMESTAMPTZ, INTERVAL, and SMALLDATETIME
- **AS query**  
Indicates a **SELECT** or **VALUES** command, or an **EXECUTE** command that runs a prepared **SELECT**, or **VALUES** query.
  - **[ WITH [ NO ] DATA ]**  
Specifies whether the data produced by the query should be copied into the new table. By default, the data is copied. If the **NO** parameter is used, the data is not copied.

## Examples

```
-- Create the tpcds.store_returns_t1 table and insert numbers that are greater than 16 in the sr_item_sk
column of the tpcds.store_returns table:
CREATE TABLE tpcds.store_returns_t1 AS SELECT * FROM tpcds.store_returns WHERE sr_item_sk > '4795';

-- Copy tpcds.store_returns to create the tpcds.store_returns_t2 table:
CREATE TABLE tpcds.store_returns_t2 AS table tpcds.store_returns;

-- Delete the tables:
DROP TABLE tpcds.store_returns_t1 ;
DROP TABLE tpcds.store_returns_t2 ;
```

## Helpful Links

[CREATE TABLE, SELECT](#)

# 12.52 CREATE TABLE PARTITION

## Function

**CREATE TABLE PARTITION** creates a partitioned table. Partitioning refers to splitting what is logically one large table into smaller physical pieces based on specific schemes. The table based on the logic is called a partition cable, and a

physical piece is called a partition. Data is stored on these smaller physical pieces, namely, partitions, instead of the larger logical partitioned table.

The common forms of partitioning include range partitioning, hash partitioning, list partitioning, and value partitioning. Currently, row-store and column-store tables support only range partitioning.

In range partitioning, the table is partitioned into ranges defined by a key column or set of columns, with no overlap between the ranges of values assigned to different partitions. Each range has a dedicated partition for data storage.

The partitioning policy for Range Partitioning refers to how data is inserted into partitions. Currently, range partitioning only allows the use of the range partitioning policy.

Range partitioning policy: Data is mapped to a created partition based on the partition key value. If the data can be mapped to, it is inserted into the specific partition; if it cannot be mapped to, error messages are returned. This is the most commonly used partitioning policy.

Partitioning can provide several benefits:

- Query performance can be improved dramatically in certain situations, particularly when most of the heavily accessed rows of the table are in a single partition or a small number of partitions. Partitioning narrows the range of data search and improves data access efficiency.
- When queries or updates access a large percentage of a single partition, performance can be improved by taking advantage of sequential scan of that partition instead of reads scattered across the whole table.
- Bulk loads and deletion can be performed by adding or removing partitions, if that requirement is planned into the partitioning design. It also entirely avoids the **VACUUM** overhead caused by a bulk **DELETE** (only for range partitioning).

## Precautions

A partitioned table supports unique and primary key constraints. The constraint keys of these constraints contain all partition keys.

## Syntax

```
CREATE TABLE [ IF NOT EXISTS ] partition_table_name
( [
  { column_name data_type [ COLLATE collation ] [ column_constraint [ ... ] ]
  | table_constraint
  | LIKE source_table [ like_option [ ... ] } ], ... ]
)
[ WITH ( { storage_parameter = value } [ , ... ] ) ]
[ COMPRESS | NOCOMPRESS ]
[ TABLESPACE tablespace_name ]
[ DISTRIBUTE BY { REPLICATION | { [ HASH ] ( column_name ) } } ]
[ TO { GROUP groupname | NODE ( nodename [ , ... ] ) } ]
PARTITION BY {
  { VALUES (partition_key) } |
  { RANGE (partition_key) ( partition_less_than_item [ , ... ] ) } |
  { RANGE (partition_key) ( partition_start_end_item [ , ... ] ) }
} [ { ENABLE | DISABLE } ROW MOVEMENT ];
```

- **column\_constraint** is as follows:

```
[ CONSTRAINT constraint_name ]
{ NOT NULL |
  NULL |
  CHECK ( expression ) |
  DEFAULT default_expr |
  UNIQUE index_parameters |
  PRIMARY KEY index_parameters }
[ DEFERRABLE | NOT DEFERRABLE | INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```
- **table\_constraint** is as follows:

```
[ CONSTRAINT constraint_name ]
{ CHECK ( expression ) |
  UNIQUE ( column_name [, ... ] ) index_parameters |
  PRIMARY KEY ( column_name [, ... ] ) index_parameters }
[ DEFERRABLE | NOT DEFERRABLE | INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```
- **like\_option** is as follows:

```
{ INCLUDING | EXCLUDING } { DEFAULTS | CONSTRAINTS | INDEXES | STORAGE | COMMENTS |
  REOPTIONS | DISTRIBUTION | ALL }
```
- **index\_parameters** is as follows:

```
[ WITH ( {storage_parameter = value} [, ... ] ) ]
[ USING INDEX TABLESPACE tablespace_name ]
```
- **partition\_less\_than\_item** is as follows:

```
PARTITION partition_name VALUES LESS THAN ( { partition_value | MAXVALUE } ) [TABLESPACE
  tablespace_name]
```
- **partition\_start\_end\_item** is as follows:

```
PARTITION partition_name {
  {START(partition_value) END (partition_value) EVERY (interval_value)} |
  {START(partition_value) END ({partition_value | MAXVALUE})} |
  {START(partition_value)} |
  {END({partition_value | MAXVALUE})}
} [TABLESPACE tablespace_name]
```

## Parameter Description

- **IF NOT EXISTS**  
Does not throw an error if a table with the same name exists. A notice is issued in this case.
- **partition\_table\_name**  
Name of the partitioned table  
Value range: a string. It must comply with the naming convention.
- **column\_name**  
Specifies the name of a column to be created in the new table.  
Value range: a string. It must comply with the naming convention.
- **data\_type**  
Specifies the data type of the column.
- **COLLATE collation**  
Assigns a collation to the column (which must be of a collatable data type). If no collation is specified, the default collation is used.
- **CONSTRAINT constraint\_name**  
Specifies a name for a column or table constraint. The optional constraint clauses specify constraints that new or updated rows must satisfy for an insert or update operation to succeed.  
There are two ways to define constraints:

- A column constraint is defined as part of a column definition, and it is bound to a particular column.
- A table constraint is not bound to any particular columns but can apply to more than one column.
- **LIKE source\_table [ like\_option ... ]**

Specifies a table from which the new table automatically copies all column names, their data types, and their not-null constraints.

Unlike **INHERITS**, the new table and original table are decoupled after creation is complete. Changes to the original table will not be applied to the new table, and it is not possible to include data of the new table in scans of the original table.

Default expressions for the copied column definitions will only be copied if **INCLUDING DEFAULTS** is specified. The default behavior is to exclude default expressions, resulting in the copied columns in the new table having default values **NULL**.

**NOT NULL** constraints are always copied to the new table. **CHECK** constraints will only be copied if **INCLUDING CONSTRAINTS** is specified; other types of constraints will never be copied. These rules also apply to column constraints and table constraints.

Unlike **INHERITS**, columns and constraints copied by **LIKE** are not merged with similarly named columns and constraints. If the same name is specified explicitly or in another **LIKE** clause, an error is reported.

  - Any indexes on the source table will not be created on the new table, unless the **INCLUDING INDEXES** clause is specified.
  - **STORAGE** settings for the copied column definitions will only be copied if **INCLUDING STORAGE** is specified. The default behavior is to exclude **STORAGE** settings.
  - Comments for the copied columns, constraints, and indexes will only be copied if **INCLUDING COMMENTS** is specified. The default behavior is to exclude comments.
  - If **INCLUDING REOPTIONS** is specified, the new table will copy the storage parameter (**WITH** clause of the source table) of the source table. The default behavior is to exclude partition definition of the storage parameter of the source table.
  - If **INCLUDING DISTRIBUTION** is specified, the new table will copy the distribution information of the source table, including distribution type and column, and the new table cannot use **DISTRIBUTE BY** clause. The default behavior is to exclude distribution information of the source table.
  - **INCLUDING ALL** is an abbreviated form of **INCLUDING DEFAULTS INCLUDING CONSTRAINTS INCLUDING INDEXES INCLUDING STORAGE INCLUDING COMMENTS INCLUDING REOPTIONS INCLUDING DISTRIBUTION**.
- **WITH ( storage\_parameter [= value] [, ... ] )**

Specifies an optional storage parameter for a table or an index. Optional parameters are as follows:

  - **FILLFACTOR**

The fillfactor of a table is a percentage between 10 and 100. 100 (complete packing) is the default value. When a smaller fillfactor is

specified, **INSERT** operations pack table pages only to the indicated percentage. The remaining space on each page is reserved for updating rows on that page. This gives **UPDATE** a chance to place the updated copy of a row on the same page, which is more efficient than placing it on a different page. For a table whose records are never updated, setting the fillfactor to 100 (complete packing) is the appropriate choice, but in heavily updated tables smaller fillfactors are appropriate. The parameter has no meaning for column-store tables.

Value range: 10–100

– **ORIENTATION**

Determines the storage mode of the data in the table.

Valid value:

- **COLUMN**: The data will be stored in columns.
- **ROW** (default value): The data will be stored in rows.
- **ORC**: The data of the table will be stored in ORC format (only HDFS table).

---

**NOTICE**

**orientation** cannot be modified.

---

– **COMPRESSION**

- The valid values for column-store tables are **YES/NO** and **LOW/MIDDLE/HIGH**, and the default is **LOW**.
- The valid values for row-store tables are **YES** and **NO**, and the default is **NO**.

– **MAX\_BATCHROW**

Specifies the maximum of a storage unit during data loading process. The parameter is only valid for column-store table.

Value range: 10000 to 60000

– **PARTIAL\_CLUSTER\_ROWS**

Specifies the number of records to be partial cluster stored during data loading process. The parameter is only valid for column-store table.

Value range: The valid value is no less than 100000. The value is the multiple of **MAX\_BATCHROW**.

– **DELTAROW\_THRESHOLD**

A reserved parameter. The parameter is only valid for column-store table.

Value range: 0 to 9999

• **COMPRESS / NOCOMPRESS**

Specifies the keyword **COMPRESS** during the creation of a table, so that the compression feature is triggered in the case of a bulk **INSERT** operation. If this feature is enabled, a scan is performed for all tuple data within the page to generate a dictionary and then the tuple data is compressed and stored. If **NOCOMPRESS** is specified, the table is not compressed.

Default value: **NOCOMPRESS**, tuple data is not compressed before storage.

- **TABLESPACE tablespace\_name**  
Specifies the new table will be created in **tablespace\_name** tablespace. If not specified, default tablespace is used.
- **DISTRIBUTE BY**  
Specifies how the table is distributed or replicated between DNs.  
Valid value:
  - **REPLICATION**: Each row in the table exists on all DNs, that is, each DN has complete table data.
  - **HASH (column\_name)**: Each row of the table will be placed into all the DNs based on the hash value of the specified column.

---

#### NOTICE

- When **DISTRIBUTE BY HASH (column\_name)** is specified, the primary key and its unique index must contain the **column\_name** column.
- When **DISTRIBUTE BY HASH (column\_name)** in a referenced table is specified, the foreign key of the reference table must contain the **column\_name** column.

---

Default value: **HASH(column\_name)**, the key column of **column\_name** (if any) or the column of distribution column supported by first data type.

**column\_name** supports the following data types:

- INTEGER TYPES: TINYINT, SMALLINT, INT, BIGINT, NUMERIC/DECIMAL
- CHARACTER TYPES: CHAR, BPCHAR, VARCHAR, VARCHAR2, NVARCHAR2
- DATA/TIME TYPES: DATE, TIME, TIMETZ, TIMESTAMP, TIMESTAMPTZ, INTERVAL, SMALLDATETIME
- **TO { GROUP groupname | NODE ( nodename [, ... ] ) }**  
**TO GROUP** specifies the Node Group in which the table is created. Currently, it cannot be used for HDFS tables. **TO NODE** is used for internal scale-out tools.
- **PARTITION BY RANGE(partition\_key)**  
Creates a range partition. **partition\_key** is the name of the partition key.  
(1) Assume that the **VALUES LESS THAN** syntax is used.

---

#### NOTICE

In this case, a maximum of four partition keys are supported.

---

Data types supported by the partition keys are as follows: SMALLINT, INTEGER, BIGINT, DECIMAL, NUMERIC, REAL, DOUBLE PRECISION, CHARACTER VARYING(n), VARCHAR(n), CHARACTER(n), CHAR(n), CHARACTER, CHAR, TEXT, NVARCHAR2, NAME, TIMESTAMP[(p)] [WITHOUT TIME ZONE], TIMESTAMP[(p)] [WITH TIME ZONE], and DATE.

(2) Assume that the **START END** syntax is used.



**NOTICE**

In this case, only one partition key is supported.

Data types supported by the partition key are as follows: SMALLINT, INTEGER, BIGINT, DECIMAL, NUMERIC, REAL, DOUBLE PRECISION, TIMESTAMP[(p)] [WITHOUT TIME ZONE], TIMESTAMP[(p)] [WITH TIME ZONE], and DATE.

- **PARTITION partition\_name VALUES LESS THAN ( { partition\_value | MAXVALUE } )**

Specifies the information of partitions. **partition\_name** is the name of a range partition. **partition\_value** is the upper limit of range partition, and the value depends on the type of **partition\_key**. **MAXVALUE** can specify the upper boundary of a range partition, and it is commonly used to specify the upper boundary of the last range partition.

**NOTICE**

- Upper boundaries must be specified for each partition.
  - The types of upper boundaries must be the same as those of partition keys.
  - In a partition list, partitions are arranged in ascending order of upper boundary values. Therefore, a partition with a certain upper boundary value is placed before another partition with a larger upper boundary value.
  - If a partition key consists of multiple columns, the columns are used for partitioning in sequence. The first column is preferred to be used for partitioning. If the values of the first columns are the same, the second column is used. The subsequent columns are used in the same manner.
- **PARTITION partition\_name {START (partition\_value) END (partition\_value) EVERY (interval\_value)} | {START (partition\_value) END (partition\_value|MAXVALUE)} | {START(partition\_value)} | {END (partition\_value | MAXVALUE)}**

Specifies partition definitions.

    - **partition\_name**: name or name prefix of a range partition. It is the name prefix only in the following cases (assuming that **partition\_name** is **p1**):
      - If START+END+EVERY is used, the names of partitions will be defined as **p1\_1**, **p1\_2**, and the like. For example, if **PARTITION p1 START(1) END(4) EVERY(1)** is defined, the generated partitions are [1, 2), [2, 3), and [3, 4), and their names are **p1\_1**, **p1\_2**, and **p1\_3**. In this case, **p1** is a name prefix.
      - If the defined statement is in the first place and has **START** specified, the range (**MINVALUE**, **START**) will be automatically used as the first actual partition, and its name will be **p1\_0**. The other partitions are then named **p1\_1**, **p1\_2**, and the like. For example, if **PARTITION p1 START(1), PARTITION p2 START(2)** is defined, generated partitions are (MINVALUE, 1), [1, 2), and [2, MAXVALUE), and their names will

be **p1\_0**, **p1\_1**, and **p2**. In this case, **p1** is a name prefix and **p2** is a partition name. **MINVALUE** means the minimum value.

- **partition\_value**: start point value or end point value of a range partition. The value depends on **partition\_key** and cannot be **MAXVALUE**.
- **interval\_value**: width of each partition for dividing the [**START**, **END**] range. It cannot be **MAXVALUE**. If the value of (**END** - **START**) divided by **EVERY** has a remainder, the width of only the last partition is less than the value of **EVERY**.
- **MAXVALUE**: maximum value. It is usually used to set the upper boundary for the last range partition.

---

**NOTICE**

1. If the defined statement is in the first place and has **START** specified, the range (**MINVALUE**, **START**) will be automatically used as the first actual partition.
2. The **START END** syntax must comply with the following rules:
  - The value of **START** (if any, same for the following situations) in each **partition\_start\_end\_item** must be smaller than that of **END**.
  - In two adjacent **partition\_start\_end\_item** statements, the value of the first **END** must be equal to that of the second **START**.
  - The value of **EVERY** in each **partition\_start\_end\_item** must be a positive number (in ascending order) and must be smaller than **END** minus **START**.
  - Each partition includes the start value (unless it is **MINVALUE**) and excludes the end value. The format is as follows: [Start value, end value).
  - Partitions created by the same **partition\_start\_end\_item** belong to the same tablespace.
  - If **partition\_name** is a name prefix of a partition, the length must not exceed 57 bytes. If there are more than 57 bytes, the prefix will be automatically truncated.
  - When creating or modifying a partitioned table, ensure that the total number of partitions in the table does not exceed the maximum value (32767).
3. In statements for creating partitioned tables, **START END** and **LESS THAN** cannot be used together.
4. The **START END** syntax in a partitioned table creation SQL statement will be replaced with the **VALUES LESS THAN** syntax when **gs\_dump** is executed.

---

- **{ ENABLE | DISABLE } ROW MOVEMENT**

Specifies the row movement switch.

If the tuple value is updated on the partition key during the **UPDATE** action, the partition where the tuple is located is altered. Setting of this parameter enables error messages to be reported or movement of the tuple between partitions.

Valid value:

- **ENABLE**: Row movement is enabled.
- **DISABLE** (default value): Disable row movement.

- **NOT NULL**

Indicates that the column is not allowed to contain **NULL** values. **ENABLE** can be omitted.

- **NULL**

Indicates that the column is allowed to contain **NULL** values. This is the default setting.

This clause is only provided for compatibility with non-standard SQL databases. You are advised not to use this clause.

- **CHECK (condition) [ NO INHERIT ]**

Specifies an expression producing a Boolean result which new or updated rows must satisfy for an insert or update operation to succeed. Expressions evaluating to **TRUE** or **UNKNOWN** succeed. If any row of an insert or update operation produces a **FALSE** result, an error exception is raised and the insert or update does not alter the database.

A check constraint specified as a column constraint should reference only the column's values, while an expression appearing in a table constraint can reference multiple columns.

A constraint marked with **NO INHERIT** will not propagate to child tables.

**ENABLE** can be omitted.

- **DEFAULT default\_expr**

Assigns a default data value for a column. The value can be any variable-free expressions (Subqueries and cross-references to other columns in the current table are not allowed). The data type of the default expression must match the data type of the column.

The default expression will be used in any insert operation that does not specify a value for the column. If there is no default value for a column, then the default value is **NULL**.

- **UNIQUE index\_parameters**

**UNIQUE ( column\_name [, ... ] ) index\_parameters**

Specifies that a group of one or more columns of a table can contain only unique values.

For the purpose of a unique constraint, **NULL** is not considered equal.

 **NOTE**

If **DISTRIBUTE BY REPLICATION** is not specified, the column table that contains only unique values must contain distribution columns.

- **PRIMARY KEY index\_parameters**

**PRIMARY KEY ( column\_name [, ... ] ) index\_parameters**

Specifies the primary key constraint specifies that a column or columns of a table can contain only unique (non-duplicate) and non-null values.

Only one primary key can be specified for a table.

 NOTE

If **DISTRIBUTE BY REPLICATION** is not specified, the column set with a primary key constraint must contain distributed columns.

- **DEFERRABLE | NOT DEFERRABLE**

Controls whether the constraint can be deferred. A constraint that is not deferrable will be checked immediately after every command. Checking of constraints that are deferrable can be postponed until the end of the transaction using the **SET CONSTRAINTS** command. **NOT DEFERRABLE** is the default value. Currently, only **UNIQUE** and **PRIMARY KEY** constraints accept this clause. All the other constraints are not deferrable.

- **INITIALLY IMMEDIATE | INITIALLY DEFERRED**

If a constraint is deferrable, this clause specifies the default time to check the constraint.

- If the constraint is **INITIALLY IMMEDIATE** (default value), it is checked after each statement.
- If the constraint is **INITIALLY DEFERRED**, it is checked only at the end of the transaction.

The constraint check time can be altered using the **SET CONSTRAINTS** command.

- **USING INDEX TABLESPACE tablespace\_name**

Allows selection of the tablespace in which the index associated with a **UNIQUE** or **PRIMARY KEY** constraint will be created. If not specified, **default\_tablespace** is consulted, or the default tablespace in the database if **default\_tablespace** is empty.

## Examples

- Example 1: Create a range-partitioned table **tpcds.web\_returns\_p1**. The table has eight partitions and the data type of their partition key is integer. The ranges of the partitions are:  $wr\_returned\_date\_sk < 2450815$ ,  $2450815 \leq wr\_returned\_date\_sk < 2451179$ ,  $2451179 \leq wr\_returned\_date\_sk < 2451544$ ,  $2451544 \leq wr\_returned\_date\_sk < 2451910$ ,  $2451910 \leq wr\_returned\_date\_sk < 2452275$ ,  $2452275 \leq wr\_returned\_date\_sk < 2452640$ ,  $2452640 \leq wr\_returned\_date\_sk < 2453005$ , and  $wr\_returned\_date\_sk \geq 2453005$ .

```
-- Create a range-partitioned table tpcds.web_returns_p1:
```

```
CREATE TABLE tpcds.web_returns_p1
(
  WR_RETURNED_DATE_SK    INTEGER           ,
  WR_RETURNED_TIME_SK    INTEGER           ,
  WR_ITEM_SK             INTEGER           NOT NULL,
  WR_REFUNDED_CUSTOMER_SK INTEGER           ,
  WR_REFUNDED_CDEMO_SK   INTEGER           ,
  WR_REFUNDED_HDEMO_SK   INTEGER           ,
  WR_REFUNDED_ADDR_SK    INTEGER           ,
  WR_RETURNING_CUSTOMER_SK INTEGER           ,
  WR_RETURNING_CDEMO_SK  INTEGER           ,
  WR_RETURNING_HDEMO_SK  INTEGER           ,
  WR_RETURNING_ADDR_SK   INTEGER           ,
  WR_WEB_PAGE_SK         INTEGER           ,
  WR_REASON_SK           INTEGER           ,
  WR_ORDER_NUMBER        BIGINT           NOT NULL,
  WR_RETURN_QUANTITY     INTEGER           ,
  WR_RETURN_AMT          DECIMAL(7,2)     ,
  WR_RETURN_TAX          DECIMAL(7,2)     ,
  WR_RETURN_AMT_INC_TAX  DECIMAL(7,2)     ,
)
```

```
WR_FEE          DECIMAL(7,2)
WR_RETURN_SHIP_COST DECIMAL(7,2)
WR_REFUNDED_CASH DECIMAL(7,2)
WR_REVERSED_CHARGE DECIMAL(7,2)
WR_ACCOUNT_CREDIT DECIMAL(7,2)
WR_NET_LOSS     DECIMAL(7,2)
)
WITH (ORIENTATION = COLUMN,COMPRESSION=MIDDLE)
DISTRIBUTE BY HASH (WR_ITEM_SK)
PARTITION BY RANGE(WR_RETURNED_DATE_SK)
(
    PARTITION P1 VALUES LESS THAN(2450815),
    PARTITION P2 VALUES LESS THAN(2451179),
    PARTITION P3 VALUES LESS THAN(2451544),
    PARTITION P4 VALUES LESS THAN(2451910),
    PARTITION P5 VALUES LESS THAN(2452275),
    PARTITION P6 VALUES LESS THAN(2452640),
    PARTITION P7 VALUES LESS THAN(2453005),
    PARTITION P8 VALUES LESS THAN(MAXVALUE)
);

-- Import data from the example data table:
INSERT INTO tpcds.web_returns_p1 SELECT * FROM tpcds.web_returns;

-- Delete the P8 partition:
ALTER TABLE tpcds.web_returns_p1 DROP PARTITION P8;

-- Add a partition WR_RETURNED_DATE_SK with values ranging from 2453005 to 2453105:
ALTER TABLE tpcds.web_returns_p1 ADD PARTITION P8 VALUES LESS THAN (2453105);

-- Add a partition WR_RETURNED_DATE_SK with values ranging from 2453105 and MAXVALUE:
ALTER TABLE tpcds.web_returns_p1 ADD PARTITION P9 VALUES LESS THAN (MAXVALUE);

-- Delete the P8 partition:
ALTER TABLE tpcds.web_returns_p1 DROP PARTITION FOR (2453005);

-- Rename the P7 partition as P10:
ALTER TABLE tpcds.web_returns_p1 RENAME PARTITION P7 TO P10;

-- Rename the P6 partition as P11:
ALTER TABLE tpcds.web_returns_p1 RENAME PARTITION FOR (2452639) TO P11;

-- Query rows in the P10 partition:
SELECT count(*) FROM tpcds.web_returns_p1 PARTITION (P10);
count
-----
929340
(1 row)

-- Query the number of rows in the P1 partition:
SELECT COUNT(*) FROM tpcds.web_returns_p1 PARTITION FOR (2450815);
count
-----
446854
(1 row)
```

- Example 2: Create a range partitioned table **tpcds.web\_returns\_p2**. The table has eight partitions and the data type of their partition key is integer. The upper limit of the eighth partition is **MAXVALUE**.

The ranges of the partitions are:  $wr\_returned\_date\_sk < 2450815$ ,  $2450815 \leq wr\_returned\_date\_sk < 2451179$ ,  $2451179 \leq wr\_returned\_date\_sk < 2451544$ ,  $2451544 \leq wr\_returned\_date\_sk < 2451910$ ,  $2451910 \leq wr\_returned\_date\_sk < 2452275$ ,  $2452275 \leq wr\_returned\_date\_sk < 2452640$ ,  $2452640 \leq wr\_returned\_date\_sk < 2453005$ , and  $wr\_returned\_date\_sk \geq 2453005$ .

The tablespace of the **tpcds.web\_returns\_p2** partitioned table is **example1**. Partitions **P1** to **P7** have no specified tablespaces, and use the **example1**

tablespace of the **tpcds.web\_returns\_p2** partitioned table. The tablespace of the **P8** partitioned table is **example2**.

Assume that *CN and DN data directory/pg\_location/mount1/path1*, *CN and DN data directory/pg\_location/mount2/path2*, *CN and DN data directory/pg\_location/mount3/path3*, and *CN and DN data directory/pg\_location/mount4/path4* are empty directories for which user **dwsadmin** has read and write permissions.

```
CREATE TABLESPACE example1 RELATIVE LOCATION 'tablespace1/tablespace_1';
CREATE TABLESPACE example2 RELATIVE LOCATION 'tablespace2/tablespace_2';
CREATE TABLESPACE example3 RELATIVE LOCATION 'tablespace3/tablespace_3';
CREATE TABLESPACE example4 RELATIVE LOCATION 'tablespace4/tablespace_4';

CREATE TABLE tpcds.web_returns_p2
(
  WR_RETURNED_DATE_SK    INTEGER          ,
  WR_RETURNED_TIME_SK   INTEGER          ,
  WR_ITEM_SK            INTEGER          NOT NULL,
  WR_REFUNDED_CUSTOMER_SK INTEGER        ,
  WR_REFUNDED_CDEMO_SK  INTEGER        ,
  WR_REFUNDED_HDEMO_SK  INTEGER        ,
  WR_REFUNDED_ADDR_SK   INTEGER        ,
  WR_RETURNING_CUSTOMER_SK INTEGER      ,
  WR_RETURNING_CDEMO_SK  INTEGER      ,
  WR_RETURNING_HDEMO_SK  INTEGER      ,
  WR_RETURNING_ADDR_SK  INTEGER      ,
  WR_WEB_PAGE_SK        INTEGER        ,
  WR_REASON_SK          INTEGER        ,
  WR_ORDER_NUMBER       BIGINT         NOT NULL,
  WR_RETURN_QUANTITY    INTEGER        ,
  WR_RETURN_AMT         DECIMAL(7,2)   ,
  WR_RETURN_TAX         DECIMAL(7,2)   ,
  WR_RETURN_AMT_INC_TAX DECIMAL(7,2)   ,
  WR_FEE                DECIMAL(7,2)   ,
  WR_RETURN_SHIP_COST   DECIMAL(7,2)   ,
  WR_REFUNDED_CASH      DECIMAL(7,2)   ,
  WR_REVERSED_CHARGE    DECIMAL(7,2)   ,
  WR_ACCOUNT_CREDIT     DECIMAL(7,2)   ,
  WR_NET_LOSS           DECIMAL(7,2)
)
TABLESPACE example1
DISTRIBUTE BY HASH (WR_ITEM_SK)
PARTITION BY RANGE(WR_RETURNED_DATE_SK)
(
  PARTITION P1 VALUES LESS THAN(2450815),
  PARTITION P2 VALUES LESS THAN(2451179),
  PARTITION P3 VALUES LESS THAN(2451544),
  PARTITION P4 VALUES LESS THAN(2451910),
  PARTITION P5 VALUES LESS THAN(2452275),
  PARTITION P6 VALUES LESS THAN(2452640),
  PARTITION P7 VALUES LESS THAN(2453005),
  PARTITION P8 VALUES LESS THAN(MAXVALUE) TABLESPACE example2
)
ENABLE ROW MOVEMENT;

-- Create a partitioned table using LIKE:
CREATE TABLE tpcds.web_returns_p3 (LIKE tpcds.web_returns_p2 INCLUDING PARTITION);

-- Change the tablespace of the P1 partition to example2:
ALTER TABLE tpcds.web_returns_p2 MOVE PARTITION P1 TABLESPACE example2;

-- Change the tablespace of the P2 partition to example3:
ALTER TABLE tpcds.web_returns_p2 MOVE PARTITION P2 TABLESPACE example3;

-- Split the P8 partition at 2453010:
ALTER TABLE tpcds.web_returns_p2 SPLIT PARTITION P8 AT (2453010) INTO
(
  PARTITION P9,
```

```
    PARTITION P10
);

-- Merge the P6 and P7 partitions into one:
ALTER TABLE tpceds.web_returns_p2 MERGE PARTITIONS P6, P7 INTO PARTITION P8;

-- Modify the migration attribute of a partitioned table:
ALTER TABLE tpceds.web_returns_p2 DISABLE ROW MOVEMENT;
-- Delete tables and tablespaces.
DROP TABLE tpceds.web_returns_p1;
DROP TABLE tpceds.web_returns_p2;
DROP TABLE tpceds.web_returns_p3;
DROP TABLESPACE example1;
DROP TABLESPACE example2;
DROP TABLESPACE example3;
DROP TABLESPACE example4;
```

- Example 3: Use **START END** to create and modify a range partitioned table.

Assume that `/home/dbadmin/startend_tbs1`, `/home/dbadmin/startend_tbs2`, `/home/dbadmin/startend_tbs3`, and `/home/dbadmin/startend_tbs4` are empty directories for which user `dbadmin` has the read/write permission.

```
-- Create tablespaces.
CREATE TABLESPACE startend_tbs1 LOCATION '/home/dbadmin/startend_tbs1';
CREATE TABLESPACE startend_tbs2 LOCATION '/home/dbadmin/startend_tbs2';
CREATE TABLESPACE startend_tbs3 LOCATION '/home/dbadmin/startend_tbs3';
CREATE TABLESPACE startend_tbs4 LOCATION '/home/dbadmin/startend_tbs4';

-- Create a temporary schema.
CREATE SCHEMA tpceds;
SET CURRENT_SCHEMA TO tpceds;

-- Create a partitioned table with the partition key of type integer.
CREATE TABLE tpceds.startend_pt (c1 INT, c2 INT)
TABLESPACE startend_tbs1
DISTRIBUTE BY HASH (c1)
PARTITION BY RANGE (c2) (
    PARTITION p1 START(1) END(1000) EVERY(200) TABLESPACE startend_tbs2,
    PARTITION p2 END(2000),
    PARTITION p3 START(2000) END(2500) TABLESPACE startend_tbs3,
    PARTITION p4 START(2500),
    PARTITION p5 START(3000) END(5000) EVERY(1000) TABLESPACE startend_tbs4
)
ENABLE ROW MOVEMENT;

-- View the information of the partitioned table.
SELECT relname, boundaries, spcname FROM pg_partition p JOIN pg_tablespace t ON
p.reltablespace=t.oid and p.parentid='tpceds.startend_pt'::regclass ORDER BY 1;
 relname | boundaries | spcname
-----+-----+-----
p1_0    | {1}       | startend_tbs2
p1_1    | {201}     | startend_tbs2
p1_2    | {401}     | startend_tbs2
p1_3    | {601}     | startend_tbs2
p1_4    | {801}     | startend_tbs2
p1_5    | {1000}    | startend_tbs2
p2      | {2000}    | startend_tbs1
p3      | {2500}    | startend_tbs3
p4      | {3000}    | startend_tbs1
p5_1    | {4000}    | startend_tbs4
p5_2    | {5000}    | startend_tbs4
startend_pt |          | startend_tbs1
(12 rows)

-- Import data and check the data volume in the partition.
INSERT INTO tpceds.startend_pt VALUES (GENERATE_SERIES(0, 4999), GENERATE_SERIES(0, 4999));
SELECT COUNT(*) FROM tpceds.startend_pt PARTITION FOR (0);
count
```

```

-----
 1
(1 row)

SELECT COUNT(*) FROM tpcds.startend_pt PARTITION (p3);
count
-----
 500
(1 row)

-- Add partitions [5000, 5300), [5300, 5600), [5600, 5900), and [5900, 6000).
ALTER TABLE tpcds.startend_pt ADD PARTITION p6 START(5000) END(6000) EVERY(300) TABLESPACE
startend_tbs4;

-- Add the partition p7, specified by MAXVALUE.
ALTER TABLE tpcds.startend_pt ADD PARTITION p7 END(MAXVALUE);

-- Rename the partition p7 to p8.
ALTER TABLE tpcds.startend_pt RENAME PARTITION p7 TO p8;

-- Delete the partition p8.
ALTER TABLE tpcds.startend_pt DROP PARTITION p8;

-- Rename the partition where 5950 is located to p71.
ALTER TABLE tpcds.startend_pt RENAME PARTITION FOR(5950) TO p71;

-- Split the partition [4000, 5000) where 4500 is located.
ALTER TABLE tpcds.startend_pt SPLIT PARTITION FOR(4500) INTO(PARTITION q1 START(4000)
END(5000) EVERY(250) TABLESPACE startend_tbs3);

-- Change the tablespace of the partition p2 to startend_tbs4.
ALTER TABLE tpcds.startend_pt MOVE PARTITION p2 TABLESPACE startend_tbs4;

-- View the partition status.
SELECT relname, boundaries, spcname FROM pg_partition p JOIN pg_tablespace t ON
p.reltablespace=t.oid and p.parentid='tpcds.startend_pt'::regclass ORDER BY 1;
  relname | boundaries | spcname
-----+-----+-----
p1_0     | {1}        | startend_tbs2
p1_1     | {201}     | startend_tbs2
p1_2     | {401}     | startend_tbs2
p1_3     | {601}     | startend_tbs2
p1_4     | {801}     | startend_tbs2
p1_5     | {1000}    | startend_tbs2
p2       | {2000}    | startend_tbs4
p3       | {2500}    | startend_tbs3
p4       | {3000}    | startend_tbs1
p5_1     | {4000}    | startend_tbs4
p6_1     | {5300}    | startend_tbs4
p6_2     | {5600}    | startend_tbs4
p6_3     | {5900}    | startend_tbs4
p71      | {6000}    | startend_tbs4
q1_1     | {4250}    | startend_tbs3
q1_2     | {4500}    | startend_tbs3
q1_3     | {4750}    | startend_tbs3
q1_4     | {5000}    | startend_tbs3
startend_pt |          | startend_tbs1
(19 rows)

-- Delete tables and tablespaces.
DROP SCHEMA tpcds CASCADE;
DROP TABLESPACE startend_tbs1;
DROP TABLESPACE startend_tbs2;
DROP TABLESPACE startend_tbs3;
DROP TABLESPACE startend_tbs4;

```

## Helpful Links

[ALTER TABLE PARTITION, DROP TABLE](#)



## 12.53 CREATE TEXT SEARCH CONFIGURATION

### Function

**CREATE TEXT SEARCH CONFIGURATION** creates a text search configuration. A text search configuration specifies a text search parser that can divide a string into tokens, plus dictionaries that can be used to determine which tokens are of interest for searching.

### Precautions

- If only the parser is specified, then the new text search configuration initially has no mappings from token types to dictionaries, and therefore will ignore all words. Subsequent **ALTER TEXT SEARCH CONFIGURATION** commands must be used to create mappings to make the configuration useful. If **COPY** option is specified, the parser, mapping and configuration option of text search configuration is copied automatically.
- If the schema name is given, the text search configuration will be created in the specified schema. Otherwise, the configuration is created in the current schema.
- The user who defines a text search configuration becomes its owner.
- **PARSER** and **COPY** options are mutually exclusive, because when an existing configuration is copied, its parser selection is copied too.
- If only the parser is specified, then the new text search configuration initially has no mappings from token types to dictionaries, and therefore will ignore all words.

### Syntax

```
CREATE TEXT SEARCH CONFIGURATION name  
  ( PARSER = parser_name | COPY = source_config )  
  [ WITH ( {configuration_option = value} [, ...] )];
```

### Parameter Description

- **name**  
Specifies the name of the text search configuration to be created. Specifies the name can be schema-qualified.
- **parser\_name**  
Specifies the name of the text search parser to use for this configuration.
- **source\_config**  
Specifies the name of an existing text search configuration to copy.
- **configuration\_option**  
Specifies the configuration parameter of text search configuration is mainly for the parser executed by **parser\_name** or contained by **source\_config**.  
Value range: The default, ngram, and zhparser parsers are supported. The parser of default type has no corresponding **configuration\_option**. [Table 12-18](#) lists **configuration\_option** for ngram and zhparser parsers.

**Table 12-18** Configuration parameters for ngram and zhparser parsers

| Parser   | Parameters for adding an account | Description                                                                                                                 | Value Range                                                                                                                                                                                                                          |
|----------|----------------------------------|-----------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ngram    | gram_size                        | Length of word segmentation                                                                                                 | Integer, 1 to 4<br>Default value: 2                                                                                                                                                                                                  |
|          | punctuation_ignore               | Whether to ignore punctuations                                                                                              | <ul style="list-style-type: none"> <li>● <b>true</b> (default value): Ignore punctuations.</li> <li>● <b>false</b>: Do not ignore punctuations.</li> </ul>                                                                           |
|          | graphical_ignore                 | Whether to ignore graphical characters                                                                                      | <ul style="list-style-type: none"> <li>● <b>true</b>: Ignore graphical characters.</li> <li>● <b>false</b> (default value): Do not ignore graphical characters.</li> </ul>                                                           |
| zhparser | punctuation_ignore               | Whether to ignore special characters including punctuations (\r and \n will not be ignored) in the word segmentation result | <ul style="list-style-type: none"> <li>● <b>true</b> (default value): Ignore all the special characters including punctuations.</li> <li>● <b>false</b>: Do not ignore all the special characters including punctuations.</li> </ul> |
|          | segment_with_duality             | Whether to aggregate segments with duality                                                                                  | <ul style="list-style-type: none"> <li>● <b>true</b>: Aggregate segments with duality.</li> <li>● <b>false</b> (default value): Do not aggregate segments with duality.</li> </ul>                                                   |
|          | multi_word_short                 | Whether to execute long words composite divide                                                                              | <ul style="list-style-type: none"> <li>● <b>true</b> (default value): Execute long words composite divide.</li> <li>● <b>false</b>: Do not execute long words composite divide.</li> </ul>                                           |
|          | multi_word_duality               | Whether to aggregate segments in long words with duality                                                                    | <ul style="list-style-type: none"> <li>● <b>true</b>: Aggregate segments in long words with duality.</li> <li>● <b>false</b> (default value): Do not aggregate segments in long words with duality.</li> </ul>                       |

| Parser | Parameters for adding an account | Description                                       | Value Range                                                                                                                                                                                    |
|--------|----------------------------------|---------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|        | multi_zmain                      | Whether to display key single words individually  | <ul style="list-style-type: none"> <li>• <b>true</b>: Display key single words individually.</li> <li>• <b>false</b> (default value): Do not display key single words individually.</li> </ul> |
|        | multi_zall                       | Whether to display all single words individually. | <ul style="list-style-type: none"> <li>• <b>true</b>: Display all single words individually.</li> <li>• <b>false</b> (default value): Do not display all single words individually.</li> </ul> |

## Examples

```
-- Create a text search configuration:
CREATE TEXT SEARCH CONFIGURATION ngram2 (parser=ngram) WITH (gram_size = 2, grapsymbol_ignore = false);

-- Create a text search configuration:
CREATE TEXT SEARCH CONFIGURATION ngram3 (copy=ngram2) WITH (gram_size = 2, grapsymbol_ignore = false);

-- Add type mapping:
ALTER TEXT SEARCH CONFIGURATION ngram2 ADD MAPPING FOR multisymbol WITH simple;

-- Create user joe:
CREATE USER joe IDENTIFIED BY 'Bigdata123@';

-- Change the owner of text search configuration:
ALTER TEXT SEARCH CONFIGURATION ngram2 OWNER TO joe;

-- Modify the schema of text search configuration:
ALTER TEXT SEARCH CONFIGURATION ngram2 SET SCHEMA joe;

-- Rename a text search configuration:
ALTER TEXT SEARCH CONFIGURATION joe.ngram2 RENAME TO ngram_2;

-- Delete type mapping:
ALTER TEXT SEARCH CONFIGURATION joe.ngram_2 DROP MAPPING IF EXISTS FOR multisymbol;

-- Delete a text search configuration:
DROP TEXT SEARCH CONFIGURATION joe.ngram_2;
DROP TEXT SEARCH CONFIGURATION ngram3;

-- Delete the schema and user joe:
DROP SCHEMA IF EXISTS joe CASCADE;
DROP ROLE IF EXISTS joe;
```

## Helpful Links

[ALTER TEXT SEARCH CONFIGURATION, DROP TEXT SEARCH CONFIGURATION](#)

## 12.54 CREATE TEXT SEARCH DICTIONARY

### Function

**CREATE TEXT SEARCH DICTIONARY** creates a full-text retrieval dictionary. A dictionary is used to identify and process particular words during full-text retrieval.

Dictionaries are created by using predefined templates (defined in the **PG\_TS\_TEMPLATE** system catalog). Five types of dictionaries can be created, **Simple**, **Ispell**, **Synonym**, **Thesaurus**, and **Snowball**. Each type of dictionaries is used to handle different tasks.

### Precautions

- A user with the **SYSADMIN** permission can create a dictionary. Then, the user automatically becomes the owner of the dictionary.
- A dictionary cannot be created in **pg\_temp** mode.
- After a dictionary is created or modified, any modification to the user-defined dictionary definition file will not affect the dictionary in the database. To make such modifications take effect in the dictionary in the database, run the **ALTER** statement to update the definition file of the dictionary.

### Syntax

```
CREATE TEXT SEARCH DICTIONARY name (  
    TEMPLATE = template  
    [, option = value [, ... ]]  
);
```

### Parameter Description

- *name*  
Specifies the name of a dictionary to be created. (If you do not specify a schema name, the dictionary will be created in the current schema.)  
Value range: a string, which complies with the identifier naming convention. A value can contain a maximum of 63 characters.
- *template*  
Specifies a template name.  
Valid value: templates (**Simple**, **Synonym**, **Thesaurus**, **Ispell**, and **Snowball**) defined in the **PG\_TS\_TEMPLATE** system catalog
- *option*  
Specifies a parameter name. Each type of dictionaries has a template containing their custom parameters. Parameters function in a way irrelevant to their setting sequence.
  - Parameters for a **Simple** dictionary
    - **STOPWORDS**  
Specifies the name of a file listing stop words. The default file name extension is .stop. In the file, each line defines a stop word.

Dictionaries will ignore blank lines and spaces in the file and convert stop-word phrases into lowercase.

- **ACCEPT**

Specifies whether to accept a non-stop word as recognized. The default value is **true**.

If **ACCEPT=true** is set for a **Simple** dictionary, no token will be passed to subsequent dictionaries. In this case, you are advised to place the **Simple** dictionary at the end of the dictionary list. If **ACCEPT=false** is set, you are advised to place the **Simple** dictionary before at least one dictionary in the list.

- **FILEPATH**

Specifies the directory for storing dictionary files. The directory can be an OBS directory. The OBS directory format is **obs://bucket\_name/path accesskey=ak secretkey=sk region=rg**. The default value is the directory where predefined dictionary files are located. If any of the **FILEPATH** and **STOPWORDS** parameters is specified, the other one must also be specified.

- Parameters for a **Synonym** dictionary

- **SYNONYM**

Specifies the name of the definition file for a **Synonym** dictionary. The default file name extension is `.syn`.

The file is a list of synonyms. Each line is in the format of *token synonym*, that is, token and its synonym separated by a space.

- **CASESENSITIVE**

Specifies whether tokens and their synonyms are case sensitive. The default value is **false**, indicating that tokens and synonyms in dictionary files will be converted into lowercase. If this parameter is set to **true**, they will not be converted into lowercase.

- **FILEPATH**

Specifies the directory for storing **Synonym** dictionary files. The directory can be an OBS directory. The OBS directory format is **obs://bucket\_name/path accesskey=ak secretkey=sk region=rg**. The default value is the directory where predefined dictionary files are located.

- Parameters for a **Thesaurus** dictionary

- **DICTFILE**

Specifies the name of a dictionary definition file. The default file name extension is `.ths`.

The file is a list of synonyms. Each line is in the format of *sample words: indexed words*. The colon (:) is used as a separator between a phrase and its substitute word. If multiple sample words are matched, the TZ selects the longest one.

- **DICTIONARY**

Specifies the name of a subdictionary used for word normalization. This parameter is mandatory and only one subdictionary name can

be specified. The specified subdictionary must exist. It is used to identify and normalize input text before phrase matching.

If an input word cannot be recognized by the subdictionary, an error will be reported. In this case, remove the word or update the subdictionary to make the word recognizable. In addition, an asterisk (\*) can be placed at the beginning of an indexed word to skip the application of a subdictionary on it, but all sample words must be recognizable by the subdictionary.

If the sample words defined in the dictionary file contain stop words defined in the subdictionary, use question marks (?) to replace them. Assume that **a** and **the** are stop words defined in the subdictionary.

```
? one ? two : ssws
```

**a one the two** and **the one a two** will be matched and output as **ssws**.

- **FILEPATH**

Specifies the directory for storing dictionary definition files. The directory can be an OBS directory. The OBS directory format is **obs://bucket\_name/path accesskey=ak secretkey=sk region=rg**. The default value is the directory where predefined dictionary files are located.

- Parameters for an **Ispell** dictionary

- **DICTFILE**

Specifies the name of a dictionary definition file. The default file name extension is .dict.

- **AFFFILE**

Specifies the name of an affix file. The default file name extension is .affix.

- **STOPWORDS**

Specifies the name of a file listing stop words. The default file name extension is .stop. The file content format is the same as that of the file for a **Simple** dictionary.

- **FILEPATH**

Specifies the directory for storing dictionary files. The directory can be an OBS directory. The OBS directory format is **obs://bucket\_name/path accesskey=ak secretkey=sk region=rg**. The default value is the directory where predefined dictionary files are located.

- Parameters for a **Snowball** dictionary

- **LANGUAGE**

Specifies the name of a language whose stemming algorithm will be used. According to spelling rules in the language, the algorithm normalizes the variants of an input word into a basic word or a stem.

- **STOPWORDS**

Specifies the name of a file listing stop words. The default file name extension is .stop. The file content format is the same as that of the file for a **Simple** dictionary.

- **FILEPATH**

Specifies the directory for storing dictionary definition files. The directory can be an OBS directory. The OBS directory format is **obs://bucket\_name/path accesskey=ak secretkey=sk region=rg**. The default value is the directory where predefined dictionary files are located. If any of the **FILEPATH** and **STOPWORDS** parameters is specified, the other one must also be specified.

 **NOTE**

The name of a dictionary definition file can contain only lowercase letters, digits, and underscores (\_).

- *value*  
Specifies a parameter value. If the value is not an identifier or a number, enclose it with single quotation marks ("). You can also enclose identifiers and numbers.

## Examples

See examples in [Configuration Examples](#).

## Helpful Links

[ALTER TEXT SEARCH DICTIONARY](#), [CREATE TEXT SEARCH DICTIONARY](#)

# 12.55 CREATE TRIGGER

## Function

**CREATE TRIGGER** creates a trigger. The trigger will be associated with a specified table or view, and will execute a specified function when certain events occur.

## Precautions

- Currently, triggers can be created only on ordinary row-store tables, instead of on column-store tables, temporary tables, or unlogged tables.
- If multiple triggers of the same kind are defined for the same event, they will be fired in alphabetical order by name.
- A trigger works only on one table. There is no limit on the number of triggers that can be created. However, more triggers on a table consume more performance.
- Triggers are usually used for data association and synchronization between multiple tables. SQL execution performance is greatly affected. Therefore, you are advised not to use this statement when a large amount of data needs to be synchronized and performance requirements are high.
- When a trigger meets the following conditions, the trigger statement and trigger itself can be pushed together down to a DN for execution, improving the trigger execution performance:
  - **enable\_trigger\_shipping** and **enable\_fast\_query\_shipping** are both enabled. (This is the default configuration.)

- The trigger function used by the source table is a PL/pgSQL function (recommended).
- The source and target tables have the same type and number of distribution keys, are both row-store tables, and belong to the same Node Group.
- The **INSERT**, **UPDATE**, or **DELETE** statement on the source table contains an expression about equality comparison between all the distribution keys and the *NEW* or *OLD* variable.
- The **INSERT**, **UPDATE**, or **DELETE** statement on the source table can be pushed down without a trigger.
- There are only six types of triggers, specified by **INSERT/UPDATE/DELETE**, **AFTER/BEFORE**, and **FOR EACH ROW**, on the source table, and all the triggers can be pushed down.

## Syntax

```
CREATE [ CONSTRAINT ] TRIGGER trigger_name { BEFORE | AFTER | INSTEAD OF } { event [ OR ... ] }  
ON table_name  
[ FROM referenced_table_name ]  
{ NOT DEFERRABLE | [ DEFERRABLE ] } { INITIALLY IMMEDIATE | INITIALLY DEFERRED } }  
[ FOR [ EACH ] { ROW | STATEMENT } ]  
[ WHEN ( condition ) ]  
EXECUTE PROCEDURE function_name ( arguments );
```

Events include:

```
INSERT  
UPDATE [ OF column_name [, ... ] ]  
DELETE  
TRUNCATE
```

## Parameter Description

- **CONSTRAINT**  
(Optional) Creates a constraint trigger, that is, a trigger is used as a constraint. Such a trigger is similar to a regular trigger except that the timing of the trigger firing can be adjusted using **SET CONSTRAINTS**. Constraint triggers must be **AFTER ROW** triggers.
- **trigger\_name**  
Specifies the name of a new trigger. The name cannot be schema-qualified because the trigger inherits the schema of its table. In addition, triggers on the same table cannot be named the same. For a constraint trigger, this is also the name to use when you modify the trigger's behavior using **SET CONSTRAINTS**.  
Value range: a string that complies with the identifier naming convention. A value can contain a maximum of 63 characters.
- **BEFORE**  
Specifies that a trigger function is called before the trigger event.
- **AFTER**  
Specifies that a trigger function is called after the trigger event. A constraint trigger can only be specified as **AFTER**.
- **INSTEAD OF**  
Specifies that a trigger function directly replaces the trigger event.



- **event**

Specifies the event that will fire a trigger. Values are **INSERT**, **UPDATE**, **DELETE**, and **TRUNCATE**. You can also specify multiple trigger events through **OR**.

For **UPDATE** events, use the following syntax to specify a list of columns:

```
UPDATE OF column_name1 [, column_name2 ... ]
```

The trigger will only fire if at least one of the listed columns is mentioned as a target of the **UPDATE** statement. **INSTEAD OF UPDATE** events do not support lists of columns.
- **table\_name**

Specifies the name of the table where a trigger needs to be created.

Value range: name of an existing table in the database
- **referenced\_table\_name**

Specifies the name of another table referenced by a constraint. This parameter can be specified only for constraint triggers. It does not support foreign key constraints and is not recommended for general use.

Value range: name of an existing table in the database
- **DEFERRABLE | NOT DEFERRABLE**

Controls whether a constraint can be deferred. The two parameters determine the timing for firing a constraint trigger, and can be specified only for constraint triggers.

For details, see [CREATE TABLE](#).
- **INITIALLY IMMEDIATE | INITIALLY DEFERRED**

If a constraint is deferrable, the two clauses specify the default time to check the constraint, and can be specified only for constraint triggers.

For details, see [CREATE TABLE](#).
- **FOR EACH ROW | FOR EACH STATEMENT**

Specifies the frequency of firing a trigger.

  - **FOR EACH ROW** indicates that the trigger should be fired once for every row affected by the trigger event.
  - **FOR EACH STATEMENT** indicates that the trigger should be fired just once per SQL statement.

If this parameter is not specified, the default value **FOR EACH STATEMENT** will be used. Constraint triggers can only be specified as **FOR EACH ROW**.
- **condition**

Specifies a Boolean expression that determines whether a trigger function will actually be executed. If **WHEN** is specified, the function will be called only when **condition** returns **true**.

In **FOR EACH ROW** triggers, the **WHEN** condition can reference the columns of old or new row values by writing **OLD.column\_name** or **NEW.column\_name**, respectively. In addition, **INSERT** triggers cannot reference **OLD** and **DELETE** triggers cannot reference **NEW**.

**INSTEAD OF** triggers do not support **WHEN** conditions.

**WHEN** expressions cannot contain subqueries.

For constraint triggers, evaluation of the **WHEN** condition is not deferred, but occurs immediately after the update operation is performed. If the condition does not return **true**, the trigger will not be queued for deferred execution.

- **function\_name**

Specifies a user-defined function, which must be declared as taking no parameters and returning data of the trigger type. This function is executed when a trigger fires.

- **arguments**

Specifies an optional, comma-separated list of parameters to be provided to a function when a trigger is executed. Parameters are literal string constants. Simple names and numeric constants can also be included, but they will all be converted to strings. Check descriptions of the implementation language of a trigger function to find out how these parameters are accessed within the function.

 **NOTE**

The following details trigger types:

- **INSTEAD OF** triggers must be marked as **FOR EACH ROW** and can be defined only on views.
- **BEFORE** and **AFTER** triggers on a view must be marked as **FOR EACH STATEMENT**.
- **TRUNCATE** triggers must be marked as **FOR EACH STATEMENT**.

**Table 12-19** Types of triggers supported on tables and views

| Trigger Timing | Trigger Event        | Row-level     | Statement-level  |
|----------------|----------------------|---------------|------------------|
| BEFORE         | INSERT/UPDATE/DELETE | Tables        | Tables and views |
|                | TRUNCATE             | Not supported | Tables           |
| AFTER          | INSERT/UPDATE/DELETE | Tables        | Tables and views |
|                | TRUNCATE             | Not supported | Tables           |
| INSTEAD OF     | INSERT/UPDATE/DELETE | Views         | Not supported    |
|                | TRUNCATE             | Not supported | Not supported    |

**Table 12-20** Special variables in the functions PL/pgSQL triggers

| Variable | Description                                                                                               |
|----------|-----------------------------------------------------------------------------------------------------------|
| NEW      | New tuple for <b>INSERT/UPDATE</b> operations. This variable is <b>NULL</b> for <b>DELETE</b> operations. |

| Variable        | Description                                                                                                              |
|-----------------|--------------------------------------------------------------------------------------------------------------------------|
| OLD             | Old tuple for <b>UPDATE/DELETE</b> operations. This variable is <b>NULL</b> for <b>INSERT</b> operations.                |
| TG_NAME         | Trigger name                                                                                                             |
| TG_WHEN         | Trigger timing ( <b>BEFORE/AFTER/INSTEAD OF</b> )                                                                        |
| TG_LEVEL        | Trigger frequency ( <b>ROW/STATEMENT</b> )                                                                               |
| TG_OP           | Trigger event ( <b>INSERT/UPDATE/DELETE/TRUNCATE</b> )                                                                   |
| TG_RELID        | OID of the table where a trigger is located                                                                              |
| TG_RELNAME      | Name of the table where a trigger is located. (This variable is now discarded and is replaced by <b>TG_TABLE_NAME</b> .) |
| TG_TABLE_NAME   | Name of the table where a trigger is located.                                                                            |
| TG_TABLE_SCHEMA | Schema information of the table where a trigger is located                                                               |
| TG_NARGS        | Number of parameters for a trigger function                                                                              |
| TG_ARGV[]       | List of parameters for a trigger function                                                                                |

## Examples

```
-- Create a source table and a target table:
CREATE TABLE test_trigger_src_tbl(id1 INT, id2 INT, id3 INT);
CREATE TABLE test_trigger_des_tbl(id1 INT, id2 INT, id3 INT);

-- Create a trigger function:
CREATE OR REPLACE FUNCTION tri_insert_func() RETURNS TRIGGER AS
$$
DECLARE
BEGIN
    INSERT INTO test_trigger_des_tbl VALUES(NEW.id1, NEW.id2, NEW.id3);
    RETURN NEW;
END
$$ LANGUAGE PLPGSQL;

CREATE OR REPLACE FUNCTION tri_update_func() RETURNS TRIGGER AS
$$
DECLARE
BEGIN
    UPDATE test_trigger_des_tbl SET id3 = NEW.id3 WHERE id1=OLD.id1;
    RETURN OLD;
END
$$
```

```
    $$ LANGUAGE PLPGSQL;

CREATE OR REPLACE FUNCTION TRI_DELETE_FUNC() RETURNS TRIGGER AS
    $$
    DECLARE
    BEGIN
        DELETE FROM test_trigger_des_tbl WHERE id1=OLD.id1;
        RETURN OLD;
    END
    $$ LANGUAGE PLPGSQL;

-- Create an INSERT trigger:
CREATE TRIGGER insert_trigger
    BEFORE INSERT ON test_trigger_src_tbl
    FOR EACH ROW
    EXECUTE PROCEDURE tri_insert_func();

-- Create an UPDATE trigger:
CREATE TRIGGER update_trigger
    AFTER UPDATE ON test_trigger_src_tbl
    FOR EACH ROW
    EXECUTE PROCEDURE tri_update_func();

-- Create a DELETE trigger:
CREATE TRIGGER delete_trigger
    BEFORE DELETE ON test_trigger_src_tbl
    FOR EACH ROW
    EXECUTE PROCEDURE tri_delete_func();

-- Execute the INSERT event and check the trigger results:
INSERT INTO test_trigger_src_tbl VALUES(100,200,300);
SELECT * FROM test_trigger_src_tbl;
SELECT * FROM test_trigger_des_tbl; // Check whether the trigger operation takes effect.

-- Execute the UPDATE event and check the trigger results:
UPDATE test_trigger_src_tbl SET id3=400 WHERE id1=100;
SELECT * FROM test_trigger_src_tbl;
SELECT * FROM test_trigger_des_tbl; // Check whether the trigger operation takes effect.

-- Execute the DELETE event and check the trigger results:
DELETE FROM test_trigger_src_tbl WHERE id1=100;
SELECT * FROM test_trigger_src_tbl;
SELECT * FROM test_trigger_des_tbl; // Check whether the trigger operation takes effect.

-- Modify a trigger:
ALTER TRIGGER delete_trigger ON test_trigger_src_tbl RENAME TO delete_trigger_renamed;

-- Disable insert_trigger:
ALTER TABLE test_trigger_src_tbl DISABLE TRIGGER insert_trigger;

-- Disable all triggers on the current table:
ALTER TABLE test_trigger_src_tbl DISABLE TRIGGER ALL;

-- Delete the triggers:
DROP TRIGGER insert_trigger ON test_trigger_src_tbl;
DROP TRIGGER update_trigger ON test_trigger_src_tbl;
DROP TRIGGER delete_trigger_renamed ON test_trigger_src_tbl;
```

## Helpful Links

[ALTER TRIGGER](#), [DROP TRIGGER](#), [ALTER TABLE](#)

## 12.56 CREATE TYPE

### Function

**CREATE TYPE** defines a new data type in the current database. The user who defines a new data type becomes its owner. Types are designed only for row-store tables.

Four types of data can be created by using **CREATE TYPE**: composite data, base data, a shell data, and enumerated data.

- **Composite types**  
A composite type is specified by a list of attribute names and data types. If the data type of an attribute is collatable, the attribute's collation rule can also be specified. A composite type is essentially the same as the row type of a table. However, using **CREATE TYPE** avoids the need to create an actual table when only a type needs to be defined. In addition, a standalone composite type is useful, for example, as the parameter or return type of a function.  
To create a composite type, you must have the **USAGE** permission for all its attribute types.
- **Base types**  
You can customize a new base type (scalar type). Generally, functions required for base types must be coded in C or another low-level language.
- **Shell types**  
A shell type is simply a placeholder for a type to be defined later. It can be created by delivering **CREATE TYPE** with no parameters except for a type name. Shell types are needed as forward references when base types are created.
- **Enumerated types**  
An enumerated type is a list of enumerated values. Each value is a non-empty string with the maximum length of 64 bytes.

### Precautions

If a schema name is given, the type will be created in the specified schema. Otherwise, it will be created in the current schema. A type name must be different from the name of any existing type or domain in the same schema. (Since tables have associated data types, a type name must also be different from the name of any existing table in the same schema.)

### Syntax

```
CREATE TYPE name AS  
    ( [ attribute_name data_type [ COLLATE collation ] [, ... ] ] )  
  
CREATE TYPE name (  
    INPUT = input_function,  
    OUTPUT = output_function  
    [ , RECEIVE = receive_function ]  
    [ , SEND = send_function ]
```

```
[ , TYPMOD_IN =  
type_modifier_input_function ]  
[ , TYPMOD_OUT =  
type_modifier_output_function ]  
[ , ANALYZE = analyze_function ]  
[ , INTERNALLENGTH = { internallength |  
VARIABLE } ]  
[ , PASSEDBYVALUE ]  
[ , ALIGNMENT = alignment ]  
[ , STORAGE = storage ]  
[ , LIKE = like_type ]  
[ , CATEGORY = category ]  
[ , PREFERRED = preferred ]  
[ , DEFAULT = default ]  
[ , ELEMENT = element ]  
[ , DELIMITER = delimiter ]  
[ , COLLATABLE = collatable ]  
)  
  
CREATE TYPE name  
  
CREATE TYPE name AS ENUM  
( [ 'label' [ , ... ] ] )
```

## Parameter Description

Composite types

- **name**  
Specifies the name of the type to be created. It can be schema-qualified.
- **attribute\_name**  
Specifies the name of an attribute (column) for the composite type.
- **data\_type**  
Specifies the name of an existing data type to become a column of the composite type.
- **collation**  
Specifies the name of an existing collation rule to be associated with a column of the composite type.

Base types

When creating a base type, you can place parameters in any order. The **input\_function** and **output\_function** parameters are mandatory, and other parameters are optional.

- **input\_function**  
Specifies the name of a function that converts data from the external text format of a type to its internal format.  
An input function can be declared as taking one parameter of the cstring type or taking three parameters of the cstring, oid, and integer types.
  - The cstring-type parameter is the input text as a C string.
  - The oid-type parameter is the OID of the type (except for array types, where the parameter is the element type OID of an array type).
  - The integer-type parameter is typmod of the destination column, if known (-1 will be passed if not known).An input function must return a value of the data type itself. Generally, an input function must be declared as **STRICT**. If it is not, it will be called with a

**NULL** parameter coming first when the system reads a **NULL** input value. In this case, the function must still return **NULL** unless an error raises. (This mechanism is designed for supporting domain input functions, which may need to reject **NULL** input values.)

 **NOTE**

Input and output functions can be declared to have the results or parameters of a new type because they have to be created before the new type is created. The new type should first be defined as a shell type, which is a placeholder type that has no attributes except a name and an owner. This can be done by delivering the **CREATE TYPE name** statement, with no additional parameters. Then, the C I/O functions can be defined as referencing the shell type. Finally, **CREATE TYPE** with a full definition replaces the shell type with a complete, valid type definition. After that, the new type can be used normally.

- **output\_function**

Specifies the name of a function that converts data from the internal format of a type to its external text format.

An output function must be declared as taking one parameter of a new data type. It must return data of the cstring type. Output functions are not invoked for **NULL** values.

- **receive\_function**

(Optional) Specifies the name of a function that converts data from the external binary format of a type to its internal format.

If this function is not used, the type cannot participate in binary input. It costs lower to convert the binary format to the internal format, more portable. (For example, the standard integer data types use the network byte order as an external binary representation, whereas the internal representation is in the machine's native byte order.) This function should perform adequate checks to ensure a valid value.

Also, this function can be declared as taking one parameter of the internal type or taking three parameters of the internal, oid, and integer types.

- The internal-type parameter is a pointer to a StringInfo buffer holding received byte strings.
- The oid- and integer-type parameters are the same as those of the text input function.

A receive function must return a value of the data type itself. Generally, a receive function must be declared as **STRICT**. If it is not, it will be called with a **NULL** parameter coming first when the system reads a **NULL** input value. In this case, the function must still return **NULL** unless an error raises. (This mechanism is designed for supporting domain receive functions, which may need to reject **NULL** input values.)

- **send\_function**

(Optional) Specifies the name of a function that converts data from the internal format of a type to its external binary format.

If this function is not used, the type cannot participate in binary output. A send function must be declared as taking one parameter of a new data type. It must return data of the bytea type. Send functions are not invoked for **NULL** values.

- **type\_modifier\_input\_function**

(Optional) Specifies the name of a function that converts an array of modifiers for a type to its internal format.

- **type\_modifier\_output\_function**

(Optional) Specifies the name of a function that converts the internal format of modifiers for a type to its external text format.

 **NOTE**

**type\_modifier\_input\_function** and **type\_modifier\_output\_function** are needed if a type supports modifiers, that is, optional constraints attached to a type declaration, such as `char(5)` or `numeric(30,2)`. GaussDB(DWS) allows user-defined types to take one or more simple constants or identifiers as modifiers. However, this information must be capable of being packed into a single non-negative integer value for storage in system catalogs. Declared modifiers are passed to **type\_modifier\_input\_function** in the `cstring` array format. The parameter must check values for validity, throwing an error if they are wrong. If they are correct, the parameter will return a single non-negative integer value, which will be stored as `typmod` in a column. If the type does not have **type\_modifier\_input\_function**, type modifiers will be rejected.

**type\_modifier\_output\_function** converts the internal integer `typmod` value back to a correct format for user display. It must return a `cstring` value, which is the exact string appending to the type name. For example, a numeric function may return `(30,2)`. If the default display format is enclosing a stored `typmod` integer value in parentheses, you can omit **type\_modifier\_output\_function**.

- **analyze\_function**

(Optional) Specifies the name of a function that performs statistical analysis for a data type.

By default, if there is a default B-tree operator class for a type, **ANALYZE** will attempt to gather statistics by using the "equals" and "less-than" operators of the type. This behavior is inappropriate for non-scalar types, and can be overridden by specifying a custom analysis function. The analysis function must be declared to take one parameter of the internal type and return a boolean result.

- **internallength**

(Optional) Specifies a numeric constant for specifying the length in bytes of the internal representation of a new type. By default, it is variable-length.

Although the details of the new type's internal representation are only known to I/O functions and other functions that you create to work with the type, there are still some attributes of the internal representation that must be declared to GaussDB(DWS). The most important one is **internallength**. Base data types can be fixed-length (when **internallength** is a positive integer) or variable-length (when **internallength** is set to **VARIABLE**; internally, this is represented by setting **typlen** to **-1**). The internal representation of all variable-length types must start with a 4-byte integer. **internallength** defines the total length.

- **PASSEDBYVALUE**

(Optional) Specifies that values of a data type are passed by value, rather than by reference. Types passed by value must be fixed-length, and their internal representation cannot be larger than the size of the Datum type (4 bytes on some machines, and 8 bytes on others).

- **alignment**

(Optional) Specifies the storage alignment required for a data type. It supports values **char**, **int2**, **int4**, and **double**. The default value is **int4**.



The allowed values equate to alignment on 1-, 2-, 4-, or 8-byte boundaries. Note that variable-length types must have an alignment of at least 4 since they must contain an int4 value as their first component.

- **storage**

(Optional) Specifies the storage strategy for a data type.

It supports values **plain**, **external**, **extended**, and **main**. The default value is **plain**.

- **plain** specifies that data of a type will always be stored in-line and not compressed. (Only **plain** is allowed for fixed-length types.)
- **extended** specifies that the system will first try to compress a long data value and will then move the value out of the main table row if it is still too long.
- **external** allows a value to be moved out of the main table, but the system will not try to compress it.
- **main** allows for compression, but discourages moving a value out of the main table. (Data items with this storage strategy might still be moved out of the main table if there is no other way to make a row fit. However, they will be kept in the main table preferentially over **extended** and **external** items.)

All **storage** values except **plain** imply that the functions of the data type can handle values that have been toasted. A given value merely determines the default **TOAST** storage strategy for columns of a toastable data type. Users can choose other strategies for individual columns by using **ALTER TABLE SET STORAGE**.

- **like\_type**

(Optional) Specifies the name of an existing data type that has the same representation as a new type. The values of **internallength**, **passedbyvalue**, **alignment**, and **storage** are copied from this type, unless they are overridden by explicit specifications elsewhere in the **CREATE TYPE** command.

Specifying representation in this way is especially useful when the low-level implementation of a new type references an existing type.

- **category**

(Optional) Specifies the category code (a single ASCII character) for a type. The default value is **U** for a user-defined type. You can also choose other ASCII characters to create custom categories.

- **preferred**

(Optional) Specifies whether a type is preferred within its type category. If it is, the value will be **TRUE**, else **FALSE**. The default value is **FALSE**. Be cautious when creating a new preferred type within an existing type category because this could cause great changes in behavior.

 **NOTE**

The **category** and **preferred** parameters can be used to help determine which implicit cast excels in ambiguous situations. Each data type belongs to a category named by a single ASCII character, and each type is either preferred or not within its category. If this rule is helpful in resolving overloaded functions or operators, the parser will prefer casting to preferred types (but only from other types within the same category). For types that have no implicit casts to or from any other types, it is sufficient to leave these parameters at their default values. However, for a group of types that have implicit casts, mark them all as belonging to a category and select one or two of the most general types as being preferred within the category. The **category** parameter is helpful in adding a user-defined type to an existing built-in category, such as the numeric or string type. However, you can also create new entirely-user-defined type categories. Select any ASCII character other than an uppercase letter to name such a category.

- **default**

(Optional) Specifies the default value for a data type. If this parameter is omitted, the default value will be **NULL**.

A default value can be specified if you expect the columns of a data type to default to something other than the **NULL** value. You can also specify a default value using the **DEFAULT** keyword. (Such a default value can be overridden by an explicit **DEFAULT** clause attached to a particular column.)

- **element**

(Optional) Specifies the type of an array element when an array type is created. For example, to define an array of 4-byte integers (**int4**), set **ELEMENT** to **int4**.

- **delimiter**

(Optional) Specifies the delimiter character to be used between values in arrays made of a type.

**delimiter** can be set to a specific character. The default delimiter is a comma (,). Note that a delimiter is associated with the array element type, instead of the array type itself.

- **collatable**

(Optional) Specifies whether a type's operations can use collation information. If they can, the value will be **TRUE**, else **FALSE** (default).

If **collatable** is **TRUE**, column definitions and expressions of a type may carry collation information by using the **COLLATE** clause. It is the implementations of functions operating on the type that actually use the collation information. This use cannot be achieved merely by marking the type collatable.

- **lable**

(Optional) Specifies a text label associated with an enumerated value. It is a non-empty string of up to 64 characters.

 **NOTE**

Whenever a user-defined type is created, GaussDB(DWS) automatically creates an associated array type whose name consists of the element type name prepended with an underscore (\_).

## Examples

```
-- Create a composite type, create a table, insert data, and make a query:  
CREATE TYPE compfoo AS (f1 int, f2 text);
```

```
CREATE TABLE t1_compfoo(a int, b compfoo);
CREATE TABLE t2_compfoo(a int, b compfoo);
INSERT INTO t1_compfoo values(1,(1,'demo'));
INSERT INTO t2_compfoo select * from t1_typ5;
SELECT (b).f1 FROM t1_compfoo;
SELECT * FROM t1_compfoo t1 join t2_compfoo t2 on (t1.b).f1=(t1.b).f1;

-- Rename the data type:
ALTER TYPE compfoo RENAME TO compfoo1;

-- Change the owner of the user-defined type compfoo1 to usr1:
CREATE USER usr1 PASSWORD 'Bigdata123@';
ALTER TYPE compfoo1 OWNER TO usr1;

-- Change the schema of the user-defined type compfoo1 to usr1:
ALTER TYPE compfoo1 SET SCHEMA usr1;

Add a new attribute to the data type:
ALTER TYPE usr1.compfoo1 ADD ATTRIBUTE f3 int;

Delete the compfoo1 type:
DROP TYPE usr1.compfoo1 cascade;

Delete related tables and users:
DROP TABLE t1_compfoo;
DROP TABLE t2_compfoo;
DROP SCHEMA usr1;
DROP USER usr1;

-- Create an enumerated type.
CREATE TYPE bugstatus AS ENUM ('create', 'modify', 'closed');

-- Add a label.
ALTER TYPE bugstatus ADD VALUE IF NOT EXISTS 'regress' BEFORE 'closed';

-- Rename a label.
ALTER TYPE bugstatus RENAME VALUE 'create' TO 'new';

-- Compile the .so file and create a shell type:
CREATE TYPE complex;
-- This statement creates a placeholder for the type to be defined so that the type can be referenced when
its I/O functions are defined. Then, you can define I/O functions. Note that the functions must be declared
to take the NOT FENCED mode during creation.
CREATE FUNCTION
complex_in(cstring)
  RETURNS complex
  AS 'filename'
  LANGUAGE C IMMUTABLE STRICT not fenced;

CREATE FUNCTION
complex_out(complex)
  RETURNS cstring
  AS 'filename'
  LANGUAGE C IMMUTABLE STRICT not fenced;

CREATE FUNCTION
complex_recv(internal)

RETURNS complex

AS 'filename'

LANGUAGE C IMMUTABLE STRICT not fenced;

CREATE FUNCTION
complex_send(complex)

RETURNS bytea
```

```
AS 'filename'

LANGUAGE C IMMUTABLE STRICT not fenced;
-- Finally, provide a complete definition of the data type:
CREATE TYPE complex (

internallength = 16,

input = complex_in,

output = complex_out,

receive = complex_recv,

send = complex_send,

alignment = double
);
```

The C functions corresponding to the input, output, receive, and send functions are defined as follows:

```
-- Define a structure body Complex:
typedef struct Complex {
    double    x;
    double    y;
} Complex;

-- Define an input function:
PG_FUNCTION_INFO_V1(complex_in);

Datum
complex_in(PG_FUNCTION_ARGS)
{
    char      *str = PG_GETARG_CSTRING(0);
    double    x,
             y;
    Complex   *result;

    if (sscanf(str, " (%lf , %lf )", &x, &y) != 2)
        ereport(ERROR,
                (errmsg("invalid input syntax for complex: \"%s\"",
                       str)));

    result = (Complex *) palloc(sizeof(Complex));
    result->x = x;
    result->y = y;
    PG_RETURN_POINTER(result);
}

-- Define an output function:
PG_FUNCTION_INFO_V1(complex_out);

Datum
complex_out(PG_FUNCTION_ARGS)
{
    Complex   *complex = (Complex *) PG_GETARG_POINTER(0);
    char      *result;

    result = (char *) palloc(100);
    snprintf(result, 100, "(%g,%g)", complex->x, complex->y);
    PG_RETURN_CSTRING(result);
}

-- Define a receive function:
PG_FUNCTION_INFO_V1(complex_recv);

Datum
complex_recv(PG_FUNCTION_ARGS)
```

```
{
    StringInfo buf = (StringInfo) PG_GETARG_POINTER(0);
    Complex *result;

    result = (Complex *) palloc(sizeof(Complex));
    result->x = pq_getmsgfloat8(buf);
    result->y = pq_getmsgfloat8(buf);
    PG_RETURN_POINTER(result);
}

-- Define a send function:
PG_FUNCTION_INFO_V1(complex_send);

Datum
complex_send(PG_FUNCTION_ARGS)
{
    Complex *complex = (Complex *) PG_GETARG_POINTER(0);
    StringInfoData buf;

    pq_begintypsend(&buf);
    pq_sendfloat8(&buf, complex->x);
    pq_sendfloat8(&buf, complex->y);
    PG_RETURN_BYTEA_P(pq_endtypsend(&buf));
}
```

## Helpful Links

[ALTER TYPE](#), [DROP TYPE](#)

# 12.57 CREATE USER

## Function

**CREATE USER** creates a user.

## Precautions

- A user created using the **CREATE USER** statement has the **LOGIN** permission by default.
- A schema named after the user is automatically created in the database where the statement is executed, but not in other databases. You can run the **CREATE SCHEMA** statement to create such a schema for the user in other databases.
- The owner of an object created by a system administrator in a schema with the same name as a common user is the common user, not the system administrator.

## Syntax

```
CREATE USER user_name [ [ WITH ] option [ ... ] ] [ ENCRYPTED | UNENCRYPTED ] { PASSWORD | IDENTIFIED BY } { 'password' | DISABLE };
```

The **option** clause is used for setting information including permissions and attributes.

```
{SYSADMIN | NOSYSADMIN}
| {AUDITADMIN | NOAUDITADMIN}
| {CREATEDB | NOCREATEDB}
| {USEFT | NOUSEFT}
| {CREATEROLE | NOCREATEROLE}
```

```
| {INHERIT | NOINHERIT}  
| {LOGIN | NOLOGIN}  
| {REPLICATION | NOREPLICATION}  
| {INDEPENDENT | NOINDEPENDENT}  
| {VCADMIN | NOVCADMIN}  
| CONNECTION LIMIT connlimit  
| VALID BEGIN 'timestamp'  
| VALID UNTIL 'timestamp'  
| RESOURCE POOL 'respool'  
| USER GROUP 'groupuser'  
| PERM SPACE 'spacelimit'  
| TEMP SPACE 'tmpspacelimit'  
| SPILL SPACE 'spillspacelimit'  
| NODE GROUP logic_cluster_name  
| IN ROLE role_name [, ...]  
| IN GROUP role_name [, ...]  
| ROLE role_name [, ...]  
| ADMIN role_name [, ...]  
| USER role_name [, ...]  
| SYSID uid  
| DEFAULT TABLESPACE tablespace_name  
| PROFILE DEFAULT  
| PROFILE profile_name  
| PGUSER  
| LDAP
```

## Parameter Description

- **user\_name**  
Specifies the user name.  
Value range: a string. It must comply with the naming convention. A value can contain a maximum of 63 characters.
- **password**  
Specifies the login password.  
A password must:
  - Contain at least eight characters. This is the default length.
  - Differ from the user name or the user name spelled backward.
  - Contain at least three of the following four character types: uppercase letters, lowercase letters, digits, and special characters, including: ~!@#\$\$%^&\*()-\_+=+|[{}];,;<.>/? . If you use characters other than the four types, a warning is displayed, but you can still create the password.
  - Be enclosed by single or double quotation marks.Value range: a string.

For details on other parameters, see [Parameter Description](#) in **CREATE ROLE**.

## Examples

```
-- Create user jim whose login password is Bigdata123@:  
CREATE USER jim PASSWORD 'Bigdata123@';  
  
-- The following statements are equivalent to the above.  
CREATE USER kim IDENTIFIED BY 'Bigdata123@';  
  
-- For a user having the Create Database permission, add the CREATEDB keyword:  
CREATE USER dim CREATEDB PASSWORD 'Bigdata123@';  
  
-- Change user jim's login password from Bigdata123@ to Abcd@123:  
ALTER USER jim IDENTIFIED BY 'Abcd@123' REPLACE 'Bigdata123@';
```

```
-- Add the CREATEROLE permission to user jim:  
ALTER USER jim CREATEROLE;  
  
-- Set enable_seqscan to on (the setting will take effect in the next session):  
ALTER USER jim SET enable_seqscan TO on;  
  
-- Reset the enable_seqscan parameter for user jim:  
ALTER USER jim RESET enable_seqscan;  
  
-- Lock the jim account:  
ALTER USER jim ACCOUNT LOCK;  
  
-- Delete the user:  
DROP USER jim CASCADE;  
DROP USER jim CASCADE;  
DROP USER jim CASCADE;
```

## Helpful Links

[ALTER USER](#), [CREATE ROLE](#), [DROP USER](#)

# 12.58 CREATE VIEW

## Function

**CREATE VIEW** creates a view. A view is a virtual table, not a base table. A database only stores the definition of a view and does not store its data. The data is still stored in the original base table. If data in the base table changes, the data in the view changes accordingly. In this sense, a view is like a window through which users can know their interested data and data changes in the database.

## Precautions

None

## Syntax

```
CREATE [ OR REPLACE ] [ TEMP | TEMPORARY ] VIEW view_name [ ( column_name [, ... ] ) ]  
[ WITH ( {view_option_name [= view_option_value]} [, ... ] ) ]  
AS query;
```

### NOTE

When creating a view, you can use WITH (security\_barriers) to create a relatively secure view. This prevents attackers from printing hidden base table data by using the RAISE statement of low costs functions.

## Parameter Description

- **OR REPLACE**  
Redefines a view if there is already a view.
- **TEMP | TEMPORARY**  
Creates a temporary view.
- **view\_name**  
Specifies the name of a view to be created. It is optionally schema-qualified.  
Value range: A string. It must comply with the naming convention.

- **column\_name**  
Specifies an optional list of names to be used for columns of the view. If not given, the column names are deduced from the query.  
Value range: A string. It must comply with the naming convention.
- **view\_option\_name [= view\_option\_value]**  
This clause specifies optional parameters for a view.  
Currently, the only parameter supported by **view\_option\_name** is **security\_barrier**, which should be enabled when a view is intended to provide row-level security.  
Value range: Boolean type. It can be **TRUE** or **FALSE**.
- **query**  
A **SELECT** or **VALUES** statement which will provide the columns and rows of the view.

## Examples

```
-- Create a view consisting of columns whose spcname is pg_default:
CREATE VIEW myView AS
  SELECT * FROM pg_tablespace WHERE spcname = 'pg_default';

-- Query a view:
SELECT * FROM myView ;

-- Delete the myView view:
DROP VIEW myView;
```

## Helpful Links

[ALTER VIEW](#), [DROP VIEW](#)

# 12.59 CURSOR

## Function

**CURSOR** defines a cursor. This command retrieves few rows of data in a query.

To process SQL statements, the stored procedure process assigns a memory segment to store context association. Cursors are handles or pointers to context regions. With cursors, stored procedures can control alterations in context regions.

## Precautions

- **CURSOR** is used only in transaction blocks.
- Generally, **CURSOR** and **SELECT** both have text returns. Since data is stored in binary format in the system, the system needs to convert the data from the binary format to the text format. If data is returned in text format, the client-end application needs to convert the data back to a binary format for processing. **FETCH** implements conversion between binary data and text data.
- Use a binary cursor unless necessary, since a text cursor occupies larger storage space than a binary cursor. A binary cursor returns internal binary data, which is easier to operate. To return data in text format, it is advisable to retrieve data in text format, therefore reducing workload at the client end.



For example, the value 1 in an integer column of a query is returned as a character string 1 if a default cursor is used, but is returned as a 4-byte binary value (big-endian) if a binary cursor is used.

## Syntax

```
CURSOR cursor_name  
[ BINARY ] [ NO SCROLL ] [ { WITH | WITHOUT } HOLD ]  
FOR query ;
```

## Parameter Description

- **cursor\_name**  
Specifies the name of a cursor to be created.  
Value range: Its value must comply with the database naming convention.
- **BINARY**  
Specifies that data retrieved by the cursor will be returned in binary format, not in text format.
- **NO SCROLL**  
Specifies the mode of data retrieval by the cursor.
  - **NO SCROLL**: If **NO SCROLL** is specified, backward fetches will be rejected.
  - Not stated: The system automatically determines whether the cursor can be used for backward fetches based on the execution plan.
- **WITH HOLD | WITHOUT HOLD**  
Specifies whether the cursor can still be used after the cursor creation event.
  - **WITH HOLD** indicates that the cursor can still be used.
  - **WITHOUT HOLD** indicates that the cursor cannot be used.
  - If neither **WITH HOLD** nor **WITHOUT HOLD** is specified, the default value is **WITHOUT HOLD**.
- **query**  
The **SELECT** or **VALUES** clause specifies the row to return the cursor value.  
Value range: **SELECT** or **VALUES** clause

## Examples

See [Examples](#) in **FETCH**.

## Helpful Links

[FETCH](#)

# 12.60 DROP DATABASE

## Function

**DROP DATABASE** deletes a database.

## Precautions

- Only the owner of a database or a system administrator has the permission to run the **DROP DATABASE** command.
- **DROP DATABASE** does not take effect for the three preinstalled system databases (**POSTGRES**, **TEMPLATE0**, and **TEMPLATE1**) because they are protected. To check databases in the current service, run the `\l` command of **gsql**.
- This command cannot be run while the database to be deleted is associated with a user. You can check the current database connections in the **v\$session** view.
- **DROP DATABASE** cannot be run inside a transaction block.
- If **DROP DATABASE** fails to be run and is rolled back, run **DROP DATABASE IF EXISTS**.
- **DROP DATABASE** cannot be undone.
- If a "database is being accessed by other users" error is displayed when you run **DROP DATABASE**, it might be that threads cannot respond to signals in a timely manner during the **CLEAN CONNECTION** process. As a result, connections are not completely cleared. In this case, you need to run **CLEAN CONNECTION** again.

## Syntax

```
DROP DATABASE [ IF EXISTS ] database_name ;
```

## Parameter Description

- **IF EXISTS**  
Sends a notice instead of an error if the specified database does not exist.
- **database\_name**  
Specifies the name of the database to be deleted.  
Value range: A string indicating an existing database name.

## Examples

See [Examples](#) in **CREATE DATABASE**.

## Helpful Links

[CREATE DATABASE](#)

# 12.61 DROP FOREIGN TABLE

## Function

**DROP FOREIGN TABLE** deletes a specified foreign table.

## Precautions

**DROP FOREIGN TABLE** forcibly deletes a specified table. After a table is deleted, any indexes that exist for the table will be deleted. The functions and stored procedures used in this table cannot be run.

## Syntax

```
DROP FOREIGN TABLE [ IF EXISTS ]  
table_name [, ...] [ CASCADE | RESTRICT ];
```

## Parameter Description

- **IF EXISTS**  
Sends a notice instead of an error if the specified table does not exist.
- **table\_name**  
Specifies the name of the table.  
Value range: An existing table name.
- **CASCADE | RESTRICT**
  - **CASCADE**: automatically deletes all objects (such as views) that depend on the table to be deleted.
  - **RESTRICT**: refuses to delete the table if any objects depend on it. This is the default.

## Examples

See [Examples](#) and [Examples](#) for **CREATE FOREIGN TABLE**.

## Helpful Links

[ALTER FOREIGN TABLE \(For GDS\)](#), [ALTER FOREIGN TABLE \(For HDFS or OBS\)](#), [CREATE FOREIGN TABLE \(for GDS Import and Export\)](#), [CREATE FOREIGN TABLE \(SQL on Hadoop or OBS\)](#)

# 12.62 DROP FUNCTION

## Function

**DROP FUNCTION** deletes an existing function.

## Precautions

If a function involves operations on temporary tables, the function cannot be deleted by running **DROP FUNCTION**.

## Syntax

```
DROP FUNCTION [ IF EXISTS ] function_name  
[ ( [ [ argmode ] [ argname ] argtype ] [, ...] ) ) [ CASCADE | RESTRICT ];
```

## Parameter Description

- **IF EXISTS**  
Sends a notice instead of an error if the function does not exist.
- **function\_name**  
Specifies the name of the function to be deleted.  
Value range: An existing function name.
- **argmode**  
Specifies the mode of a function parameter.
- **argname**  
Specifies the name of a function parameter.
- **argtype**  
Specifies the data types of a function parameter.
- **CASCADE | RESTRICT**
  - **CASCADE**: automatically deletes all objects that depend on the function to be deleted (such as operators).
  - **RESTRICT**: refuses to delete the function if any objects depend on it. This is the default.

## Examples

See [Examples](#) in **CREATE FUNCTION**.

## Helpful Links

[ALTER FUNCTION](#), [CREATE FUNCTION](#)

# 12.63 DROP GROUP

## Function

**DROP GROUP** deletes a user group.

**DROP GROUP** is the alias for **DROP ROLE**.

## Precautions

**DROP GROUP** is the internal interface encapsulated in the **gs\_om** tool. You are not advised to use this interface, because doing so affects the cluster.

## Syntax

```
DROP GROUP [ IF EXISTS ] group_name [, ...];
```

## Parameter Description

See [Parameter Description](#) in **DROP ROLE**.

## Helpful Links

[CREATE GROUP, ALTER GROUP, DROP ROLE](#)

# 12.64 DROP INDEX

## Function

**DROP INDEX** deletes an index.

## Precautions

Only the owner of an index or a system administrator can run **DROP INDEX** command.

## Syntax

```
DROP INDEX [ CONCURRENTLY ] [ IF EXISTS ]  
index_name [, ...] [ CASCADE | RESTRICT ];
```

## Parameter Description

- **CONCURRENTLY**  
Deletes an index without locking it. In general, during the deletion, the access of other statements to the tables that the index depends on is blocked. This keyword allows the access of other statements in this case.  
This option only specifies one index name, and **CASCADE** cannot be used in this case.  
The **DROP INDEX** command can be run within a transaction, but **DROP INDEX CONCURRENTLY** cannot be run in a transaction.
- **IF EXISTS**  
Sends a notice instead of an error if the specified index does not exist.
- **index\_name**  
Specifies the name of the index to be deleted.  
Value range: An existing index.
- **CASCADE | RESTRICT**
  - **CASCADE**: automatically deletes all objects that depend on the index to be deleted.
  - **RESTRICT** (default): refuses to delete the index if any objects depend on it.

## Examples

See [Examples](#) in **CREATE INDEX**.

## Helpful Links

[ALTER INDEX, CREATE INDEX](#)

## 12.65 DROP NODE

### Function

**DROP NODE** deletes a node.

### Precautions

**CREATE NODE** is the internal interface encapsulated in **gs\_om**. You are not advised to use this interface, because doing so affects the cluster.

### Syntax

```
DROP NODE [ IF EXISTS ] nodename [WITH ( cnnodename [, ... ] )];
```

### Parameter Description

#### **IF EXISTS**

Sends a notice instead of an error if the specified node does not exist.

#### **nodename**

Specifies the name of the node to be deleted.

Value range: An existing node name.

#### **cnnodename**

Specifies the CN name. If it is specified, **DROP NODE** will be executed on both the connected CN and the specified CN. If it is not specified, DN deletion must be performed on all CNs, and CN deletion must be performed on all CNs except the CN to be deleted.

Value range: An existing CN node name.

### Helpful Links

[CREATE NODE, ALTER NODE](#)

## 12.66 DROP NODE GROUP

### Function

**DROP NODE GROUP** deletes a node group.

### Precautions

- **DROP NODE GROUP** is the internal interface encapsulated in **gs\_om**.
- This interface is available only to administrators.

## Syntax

```
DROP NODE GROUP groupname [DISTRIBUTE FROM src_group_name];
```

### Parameter Description

#### **groupname**

Specifies the name of the node group to be deleted.

Value range: An existing node group.

#### **DISTRIBUTE FROM src\_group\_name**

If the Node Group to be deleted originated from the Node Group specified by **src\_group\_name**, set **src\_group\_name** to specify the source Node Group, to which the node information should be synchronized after redistribution. This statement is used only for redistribution during scale-out. You are not advised to use it, because it may lead to data distribution errors or Node Group unavailability.

### Helpful Links

[CREATE NODE GROUP](#)

## 12.67 DROP OWNED

### Function

**DROP OWNED** deletes the database objects of a database role.

### Precautions

The role's permissions on all the database objects in the current database and shared objects (databases and tablespaces) will be revoked.

### Syntax

```
DROP OWNED BY name [, ...] [ CASCADE | RESTRICT ];
```

### Parameter Description

- **name**  
Specifies the role name.  
Valid value:
- **CASCADE | RESTRICT**
  - **CASCADE**: automatically deletes all objects that depend on the objects to be deleted.
  - **RESTRICT** (default): refuses to delete the object if any objects depend on it.

## 12.68 DROP ROW LEVEL SECURITY POLICY

### Function

**DROP ROW LEVEL SECURITY POLICY** deletes a row-level access control policy from a table.

### Precautions

Only the table owner or administrators can delete a row-level access control policy from the table.

### Syntax

```
DROP [ ROW LEVEL SECURITY ] POLICY [ IF EXISTS ] policy_name ON table_name [ CASCADE | RESTRICT ]
```

### Parameter Description

- **IF EXISTS**  
Reports a notice instead of an error if the specified row-level access control policy does not exist.
- *policy\_name*  
Specifies the name of a row-level access control policy to be deleted.
  - *table\_name*  
Specifies the name of a table to which a row-level access control policy is applied.
  - CASCADE/RESTRICT  
The two parameters are used only for syntax compatibility. No objects depend on access control policies and thereby **CASCADE** is equivalent to **RESTRICT**.

### Examples

```
-- Create the data table all_data.  
CREATE TABLE all_data(id int, role varchar(100), data varchar(100));  
  
-- Create a row-level access control policy.  
CREATE ROW LEVEL SECURITY POLICY all_data_rls ON all_data USING(role = CURRENT_USER);  
  
-- Delete the row-level access control policy.  
DROP ROW LEVEL SECURITY POLICY all_data_rls ON all_data;
```

### Helpful Links

[ALTER ROW LEVEL SECURITY POLICY](#), [CREATE ROW LEVEL SECURITY POLICY](#)

## 12.69 DROP PROCEDURE

### Function

**DROP PROCEDURE** deletes an existing stored procedure.



## Precautions

None.

## Syntax

```
DROP PROCEDURE [ IF EXISTS ] procedure_name ;
```

## Parameter Description

- **IF EXISTS**  
Sends a notice instead of an error if the stored procedure does not exist.
- **procedure\_name**  
Specifies the name of the stored procedure to be deleted.  
Value range: An existing stored procedure name.

## Examples

See [Examples](#) in **CREATE PROCEDURE**.

## Helpful Links

[CREATE PROCEDURE](#)

# 12.70 DROP RESOURCE POOL

## Function

**DROP RESOURCE POOL** deletes a resource pool.

### NOTE

The resource pool cannot be deleted if it is associated with a role.

## Precautions

The user must have the DROP permission in order to delete a resource pool.

## Syntax

```
DROP RESOURCE POOL [ IF EXISTS ] pool_name;
```

## Parameter Description

- **IF EXISTS**  
Sends a notice instead of an error if the stored procedure does not exist.
- **pool\_name**  
Specifies the name of a created resource pool.  
Value range: A string compliant with the identifier naming rules.

 NOTE

A resource pool can be independently deleted only when it is not associated with any users.

## Examples

See [Examples](#) in **CREATE RESOURCE POOL**.

## Helpful Links

[ALTER RESOURCE POOL](#), [CREATE RESOURCE POOL](#)

# 12.71 DROP ROLE

## Function

**DROP ROLE** deletes a specified role.

## Precautions

If a "role is being used by other users" error is displayed when you run **DROP ROLE**, it might be that threads cannot respond to signals in a timely manner during the **CLEAN CONNECTION** process. As a result, connections are not completely cleared. In this case, you need to run **CLEAN CONNECTION** again.

## Syntax

```
DROP ROLE [ IF EXISTS ] role_name [, ...];
```

## Parameter Description

- **IF EXISTS**  
Sends a notice instead of an error if the specified role does not exist.
- **role\_name**  
Specifies the name of the role to be deleted.  
Value range: An existing role.

## Examples

See [Examples](#) in **CREATE ROLE**.

## Helpful Links

[CREATE ROLE](#), [ALTER ROLE](#), [SET ROLE](#)

# 12.72 DROP SCHEMA

## Function

**DROP SCHEMA** deletes a schema in a database.

## Precautions

Only a schema owner or a system administrator can run the **DROP SCHEMA** command.

## Syntax

```
DROP SCHEMA [ IF EXISTS ] schema_name [, ...] [ CASCADE | RESTRICT ];
```

## Parameter Description

- **IF EXISTS**  
Sends a notice instead of an error if the specified schema does not exist.
- **schema\_name**  
Specifies the name of a schema.  
Value range: An existing schema name.
- **CASCADE | RESTRICT**
  - **CASCADE**: automatically deletes all objects that are contained in the schema to be deleted.
  - **RESTRICT**: refuses to delete the schema that contains any objects. This is the default.

---

### NOTICE

Do not delete the schemas with the beginning of **pg\_temp** or **pg\_toast\_temp**. They are internal system schemas, and deleting them may cause unexpected errors.

---

### NOTE

A user cannot delete the schema in use. To delete the schema in use, switch to another schema.

## Examples

See [Examples](#) in **CREATE SCHEMA**.

## Helpful Links

[ALTER SCHEMA](#), [CREATE SCHEMA](#)

# 12.73 DROP SEQUENCE

## Function

**DROP SEQUENCE** deletes a sequence from the current database.

## Precautions

Only a sequence owner or a system administrator can delete a sequence.

## Syntax

```
DROP SEQUENCE [ IF EXISTS ] {[schema.]sequence_name} [ , ... ] [ CASCADE | RESTRICT ];
```

## Parameter Description

- **IF EXISTS**  
Sends a notice instead of an error if the specified sequence does not exist.
- **name**  
Specifies the name of the sequence.
- **CASCADE**  
Automatically deletes objects that depend on the sequence to be deleted.
- **RESTRICT**  
Refuses to delete the sequence if any objects depend on it. This is the default.

## Examples

```
-- Create a sequence named serial that starts from 101 and increases in ascending order:  
CREATE SEQUENCE serial START 101;  
  
-- Delete the sequence:  
DROP SEQUENCE serial;
```

## Helpful Links

[CREATE SEQUENCE ALTER SEQUENCE](#)

# 12.74 DROP SERVER

## Function

**DROP SERVER** deletes an existing data server.

## Precautions

Only the server owner can delete a server.

## Syntax

```
DROP SERVER [ IF EXISTS ] server_name [ {CASCADE | RESTRICT} ] ;
```

## Parameter Description

- **IF EXISTS**  
Sends a notice instead of an error if the specified table does not exist.
- **server\_name**  
Specifies the name of a server.
- **CASCADE | RESTRICT**
  - **CASCADE**: automatically drops objects that depend on the server to be deleted.

- **RESTRICT** (default): refuses to delete the server if any objects depend on it.

## Examples

See [Examples](#) in **CREATE SERVER**.

## Helpful Links

[CREATE SERVER](#), [ALTER SERVER](#)

# 12.75 DROP SYNONYM

## Function

**DROP SYNONYM** is used to delete a synonym object.

## Precautions

Only a synonym owner or a system administrator can run the **DROP SYNONYM** command.

## Syntax

```
DROP SYNONYM [ IF EXISTS ] synonym_name [ CASCADE | RESTRICT ];
```

## Parameter Description

- **IF EXISTS**  
Send a notice instead of reporting an error if the specified synonym does not exist.
- **synonym\_name**  
Name of a synonym (optionally with schema names)
- **CASCADE | RESTRICT**
  - **CASCADE**: automatically deletes objects (such as views) that depend on the synonym to be deleted.
  - **RESTRICT**: refuses to delete the synonym if any objects depend on it. This is the default.

## Examples

For details about **CREATE YNONYM**, see [Examples](#).

## Helpful Links

[ALTER SYNONYM](#) and [CREATE SYNONYM](#)

## 12.76 DROP TABLE

### Function

**DROP TABLE** deletes a specified table.

### Precautions

**DROP TABLE** forcibly deletes a specified table. After a table is deleted, any indexes that exist for the table will be deleted; any functions or stored procedures that use this table cannot be run. Deleting a partitioned table also deletes all partitions in the table.

### Syntax

```
DROP TABLE [ IF EXISTS ]  
{ [schema.]table_name } [, ...] [ CASCADE | RESTRICT ];
```

### Parameter Description

- **IF EXISTS**  
Sends a notice instead of an error if the specified table does not exist.
- **schema**  
Specifies the schema name.
- **table\_name**  
Specifies the name of the table.
- **CASCADE | RESTRICT**
  - **CASCADE**: automatically deletes objects (such as views) that depend on the table to be deleted.
  - **RESTRICT** (default): refuses to delete the table if any objects depend on it. This is the default.

### Examples

For details, see [Examples](#) of **CREATE TABLE**.

### Helpful Links

[ALTER TABLE](#), [CREATE TABLE](#)

## 12.77 DROP TEXT SEARCH CONFIGURATION

### Function

**DROP TEXT SEARCH CONFIGURATION** deletes an existing text search configuration.

## Precautions

Only the owner of the configuration can run this command.

## Syntax

```
DROP TEXT SEARCH CONFIGURATION [ IF EXISTS ] name [ CASCADE | RESTRICT ];
```

## Parameter Description

- **IF EXISTS**  
Sends a notice instead of an error if the specified text search configuration does not exist.
- **name**  
Specifies the name (optionally schema-qualified) of a text search configuration to be deleted.
- **CASCADE**  
Automatically deletes objects that depend on the text search configuration to be deleted.
- **RESTRICT**  
Refuses to delete the text search configuration if any objects depend on it. This is the default.

## Examples

See [Examples](#) in **CREATE TEXT SEARCH CONFIGURATION**.

## Helpful Links

[ALTER TEXT SEARCH CONFIGURATION](#), [CREATE TEXT SEARCH CONFIGURATION](#)

# 12.78 DROP TEXT SEARCH DICTIONARY

## Function

**DROP TEXT SEARCH DICTIONARY** deletes a full-text retrieval dictionary.

## Precautions

- **DROP** is not supported by predefined dictionaries.
- Only the owner of a dictionary can do **DROP** to the dictionary. System administrators have this permission by default.
- Execute **DROP..CASCADE** only when necessary because this operation will delete the text search configuration that uses this dictionary.

## Syntax

```
DROP TEXT SEARCH DICTIONARY [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

## Parameter Description

- **IF EXISTS**  
Reports a notice instead of throwing an error if the specified full-text retrieval dictionary does not exist.
- *name*  
Specifies the name of a dictionary to be deleted. (If you do not specify a schema name, the dictionary in the current schema will be deleted by default.)  
Value range: name of an existing dictionary
- **CASCADE**  
Automatically deletes dependent objects of a dictionary and then deletes all dependent objects of these objects in sequence.  
If any text search configuration that uses the dictionary exists, **DROP** execution will fail. You can add **CASCADE** to delete all text search configurations and dictionaries that use the dictionary.
- **RESTRICT**  
Rejects the deletion of a dictionary if any object depends on the dictionary. This is the default.

## Examples

```
-- Delete the english dictionary.  
DROP TEXT SEARCH DICTIONARY english;
```

## Helpful Links

[ALTER TEXT SEARCH DICTIONARY, CREATE TEXT SEARCH DICTIONARY](#)

# 12.79 DROP TRIGGER

## Function

**DROP TRIGGER** deletes a trigger.

## Precautions

Only the owner of a trigger and system administrators can run the **DROP TRIGGER** statement.

## Syntax

```
DROP TRIGGER [ IF EXISTS ] trigger_name ON table_name [ CASCADE | RESTRICT ];
```

## Parameter Description

- **IF EXISTS**  
Sends a notice instead of an error if the specified trigger does not exist.
- **trigger\_name**  
Specifies the name of the trigger to be deleted.



Value range: an existing trigger

- **table\_name**

Specifies the name of the table where the trigger to be deleted is located.

Value range: an existing table having a trigger

- **CASCADE | RESTRICT**

- **CASCADE**: Deletes objects that depend on the trigger.

- **RESTRICT**: Refuses to delete the trigger if any objects depend on it. This is the default.

## Examples

For details, see [CREATE TRIGGER](#).

## Helpful Links

[CREATE TRIGGER](#), [ALTER TRIGGER](#), [ALTER TABLE](#)

# 12.80 DROP TYPE

## Function

**DROP TYPE** deletes a user-defined data type. Only the type owner has permission to run this statement.

## Syntax

```
DROP TYPE [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

## Parameter Description

- **IF EXISTS**

Sends a notice instead of an error if the specified type does not exist.

- **name**

Specifies the name of the type to be deleted (schema-qualified).

- **CASCADE**

Deletes objects (such as columns, functions, and operators) that depend on the type.

- **RESTRICT**

Refuses to delete the type if any objects depend on it. This is the default.

## Examples

For details about **CREATE TYPE**, see [Examples](#).

## Helpful Links

[ALTER TYPE](#), [CREATE TYPE](#)

## 12.81 DROP USER

### Function

Deleting a user will also delete the schema having the same name as the user.

### Precautions

- **CASCADE** is used to delete objects (excluding databases) that depend on the user. **CASCADE** cannot delete locked objects unless the locked objects are unlocked or the processes that lock the objects are killed.
- When deleting a user in the database, if the object that the user depends on is in another database or the object of the dependent user is another database, you need to manually delete the dependent objects in other databases or delete the dependent database. Then, delete the user. Cross-database cascading deletion cannot be performed.
- In a multi-tenant scenario, the service user will also be deleted when you delete a user group. If the specified **CASCADE** concatenation is deleted, **CASCADE** will be specified upon the deletion of the service user. If you fail to delete a user, an error is reported, and you cannot delete other users either.
- If the user has an error table specified when the GDS foreign table is created, the user cannot be deleted by specifying the **CASCADE** keyword in the **DROP USER** command.
- If a "role is being used by other users" error is displayed when you run **DROP USER**, it might be that threads cannot respond to signals in a timely manner during the **CLEAN CONNECTION** process. As a result, connections are not completely cleared. In this case, you need to run **CLEAN CONNECTION** again.

### Syntax

```
DROP USER [ IF EXISTS ] user_name [, ...] [ CASCADE | RESTRICT ];
```

### Parameter Description

- **IF EXISTS**  
Sends a notice instead of an error if the specified user does not exist.
- **user\_name**  
Specifies the name of a user to be deleted.  
Value range: An existing user name.
- **CASCADE | RESTRICT**
  - **CASCADE**: automatically deletes all objects that depend on the user to be deleted.
  - **RESTRICT**: refuses to delete the user if any objects depend on it. This is the default.

 NOTE

In GaussDB(DWS), the `postgresql.conf` file contains the `enable_kill_query` parameter. This parameter affects the action of deleting user objects using `CASCADE`.

- If `enable_kill_query` is `on` and `CASCADE` is used to delete user objects, the processes will be automatically killed and the user will be deleted at the same time.
- If `enable_kill_query` is `off` and `CASCADE` is used to delete user objects, the user will be deleted after the processes are automatically killed.

## Examples

See [Examples](#) in `CREATE USER`.

## Helpful Links

[ALTER USER](#), [CREATE USER](#)

# 12.82 DROP VIEW

## Function

`DROP VIEW` forcibly deletes an existing view in a database.

## Precautions

Only a view owner or a system administrator can run `DROP VIEW` command.

## Syntax

```
DROP VIEW [ IF EXISTS ] view_name [ ... ] [ CASCADE | RESTRICT ];
```

## Parameter Description

- **IF EXISTS**  
Sends a notice instead of an error if the specified view does not exist.
- **view\_name**  
Specifies the name of the view to be deleted.  
Value range: An existing view.
- **CASCADE | RESTRICT**
  - **CASCADE**: deletes objects (such as other views) that depend on a view to be deleted.
  - **RESTRICT**: refuses to delete the view if any objects depend on it. This is the default.

## Examples

See [Examples](#) in `CREATE VIEW`.

## Helpful Links

[ALTER VIEW](#), [CREATE VIEW](#)

# 12.83 FETCH

## Function

**FETCH** retrieves data using a previously-created cursor.

A cursor has an associated position, which is used by **FETCH**. The cursor position can be before the first row of the query result, on any particular row of the result, or after the last row of the result.

- When created, a cursor is positioned before the first row.
- After fetching some rows, the cursor is positioned on the row most recently retrieved.
- If **FETCH** runs off the end of the available rows then the cursor is left positioned after the last row, or before the first row if fetching backward.
- **FETCH ALL** or **FETCH BACKWARD ALL** will always leave the cursor positioned after the last row or before the first row.

## Precautions

- If **NO SCROLL** is defined for the cursor, a backward fetch like **FETCH BACKWARD** is not allowed.
- The forms **NEXT**, **PRIOR**, **FIRST**, **LAST**, **ABSOLUTE**, and **RELATIVE** appropriately fetch a record after moving the cursor. If the cursor is already after the last row before being moved, an empty result is returned, and the cursor is left positioned before the first row (backward fetch) or after the last row (forward fetch) as appropriate.
- The forms using **FORWARD** and **BACKWARD** retrieve the indicated number of rows moving in the forward or backward direction, leaving the cursor positioned on the last-returned row (or after (backward fetch)/before (forward fetch) all rows, if the count exceeds the number of rows available).
- **RELATIVE 0**, **FORWARD 0**, and **BACKWARD 0** all request fetching the current row without moving the cursor, that is, re-fetching the most recently fetched row. This will succeed unless the cursor is positioned before the first row or after the last row, in which case, no row is returned.
- If the cursor of **FETCH** involves a column-store table or, backward fetches like **BACKWARD**, **PRIOR**, and **FIRST** are not supported.

## Syntax

```
FETCH [ direction { FROM | IN } ] cursor_name;
```

The **direction** clause specifies optional parameters.

```
NEXT  
| PRIOR  
| FIRST  
| LAST  
| ABSOLUTE count
```

```
| RELATIVE count  
| count  
| ALL  
| FORWARD  
| FORWARD count  
| FORWARD ALL  
| BACKWARD  
| BACKWARD count  
| BACKWARD ALL
```

## Parameter Description

- **direction\_clause**

Defines the fetch direction.

Valid value:

- **NEXT** (default value)  
Fetches the next row.
- **PRIOR**  
Fetches the prior row.
- **FIRST**  
Fetches the first row of the query (same as **ABSOLUTE 1**).
- **LAST**  
Fetches the last row of the query (same as **ABSOLUTE -1**).
- **ABSOLUTE count**

Fetches the (**count**)'th row of the query.

**ABSOLUTE** fetches are not any faster than navigating to the desired row with a relative move: the underlying implementation must traverse all the intermediate rows anyway.

**count** is a possibly-signed integer constant:

- If **count** is a positive integer, fetches the (count)'th row of the query, starting from the first row. If **count** is less than the current cursor position, a **rewind** operation is required, which is currently not supported.
- If **count** is a negative value or 0, a backward scanning is required, which is currently not supported.
- **RELATIVE count**  
Fetches the (count)'th succeeding row, or the abs(count)'th prior row if count is negative.

**count** is a possibly-signed integer constant:

- If **count** is a positive integer, fetches the (count)'th succeeding row.
- If **count** is a negative value, a backward scanning is required, which is currently not supported.
- **RELATIVE 0** fetches the current row.
- **count**  
Fetches the next **count** rows (same as **FORWARD count**).

- ALL  
Fetches all remaining rows (same as **FORWARD ALL**).
- FORWARD  
Fetches the next row (same as **NEXT**).
- FORWARD count  
Fetches the next **count** rows (same as **RELATIVE count**). **FORWARD 0** re-fetches the current row.
- FORWARD ALL  
Fetches all remaining rows.
- BACKWARD  
Fetches the prior row (same as **PRIOR**).
- BACKWARD count  
Fetches the prior **count** rows (scanning backwards).  
**count** is a possibly-signed integer constant:
  - If **count** is a positive integer, fetches the (count)'th prior row.
  - If **count** is a negative integer, fetches the abs(count)'th succeeding row.
  - **BACKWARD 0** re-fetches the current row.
- BACKWARD ALL  
Fetches all prior rows (scanning backwards).
- **{ FROM | IN } cursor\_name**  
Specifies the cursor name using the keyword **FROM** or **IN**.  
Value range: an existing cursor name.

## Examples

```
-- For the SELECT statement, use a cursor to read a table. Start a transaction:
START TRANSACTION;

-- Set up the cursor1 cursor:
CURSOR cursor1 FOR SELECT * FROM tpcds.customer_address ORDER BY 1;

-- Fetch the first three rows from cursor1:
FETCH FORWARD 3 FROM cursor1;
ca_address_sk | ca_address_id | ca_street_number | ca_street_name | ca_street_type | ca_suite_number
| ca_city | ca_county | ca_state | ca_zip | ca_country | ca_gmt_offset | ca_location_type
-----+-----+-----+-----+-----+-----+-----
+-----+-----+-----+-----+-----+-----+-----
1 | AAAAAAAAABAAAAAAA | 18 | Jackson | Parkway | Suite 280 |
Fairfield | Maricopa County | AZ | 86192 | United States | -7.00 | condo
2 | AAAAAAACAACAAAAAA | 362 | Washington 6th | RD | Suite 80 |
Fairview | Taos County | NM | 85709 | United States | -7.00 | condo
3 | AAAAAAADAAAAAAA | 585 | Dogwood Washington | Circle | Suite Q |
Pleasant Valley | York County | PA | 12477 | United States | -5.00 | single family
(3 rows)

-- Close the cursor and commit the transaction:
CLOSE cursor1;

-- End the transaction:
END;
```

```

-- For the VALUES clause, use a cursor to read the content of the clause. Start a transaction:
START TRANSACTION;

-- Set up the cursor cursor2:
CURSOR cursor2 FOR VALUES(1,2),(0,3) ORDER BY 1;

-- Fetch the first two rows from cursor2:
FETCH FORWARD 2 FROM cursor2;
column1 | column2
-----+-----
0 | 3
1 | 2
(2 rows)

-- Close the cursor and commit the transaction:
CLOSE cursor2;

-- End the transaction:
END;

-- Use the WITH HOLD cursor to start a transaction:
START TRANSACTION;

-- Set up a WITH HOLD cursor:
DECLARE cursor1 CURSOR WITH HOLD FOR SELECT * FROM tpceds.customer_address ORDER BY 1;

-- Fetch the first two rows from cursor1:
FETCH FORWARD 2 FROM cursor1;
ca_address_sk | ca_address_id | ca_street_number | ca_street_name | ca_street_type | ca_suite_number
| ca_city | ca_county | ca_state | ca_zip | ca_country | ca_gmt_offset | ca_location_type
-----+-----+-----+-----+-----+-----+-----
1 | AAAAAAAAAAAAAAAAA | 18 | Jackson | Parkway | Suite 280 |
Fairfield | Maricopa County | AZ | 86192 | United States | -7.00 | condo
2 | AAAAAAAAAAAAAAAAA | 362 | Washington 6th | RD | Suite 80 |
Fairview | Taos County | NM | 85709 | United States | -7.00 | condo
(2 rows)

-- End the transaction:
END;

-- Fetch the next row from cursor1:
FETCH FORWARD 1 FROM cursor1;
ca_address_sk | ca_address_id | ca_street_number | ca_street_name | ca_street_type | ca_suite_number
| ca_city | ca_county | ca_state | ca_zip | ca_country | ca_gmt_offset | ca_location_type
-----+-----+-----+-----+-----+-----+-----
3 | AAAAAAAAAAAAAAAAA | 585 | Dogwood Washington | Circle | Suite Q |
Pleasant Valley | York County | PA | 12477 | United States | -5.00 | single family
(1 row)

-- Close the cursor:
CLOSE cursor1;

```

## Helpful Links

[CLOSE, MOVE](#)

## 12.84 MOVE

### Function

**MOVE** repositions a cursor without retrieving any data. **MOVE** works exactly like the **FETCH** command, except it only repositions the cursor and does not return rows.

## Precautions

None

## Syntax

```
MOVE [ direction [ FROM | IN ] ] cursor_name;
```

The **direction** clause specifies optional parameters.

```
NEXT
| PRIOR
| FIRST
| LAST
| ABSOLUTE count
| RELATIVE count
| count
| ALL
| FORWARD
| FORWARD count
| FORWARD ALL
| BACKWARD
| BACKWARD count
| BACKWARD ALL
```

## Parameter Description

**MOVE** command parameters are the same as **FETCH** command parameters. For details, see [Parameter Description](#) in **FETCH**.

### NOTE

On successful completion, a **MOVE** command returns a command tag of the form **MOVE count**. The **count** is the number of rows that a **FETCH** command with the same parameters would have returned (possibly zero).

## Examples

```
-- Start a transaction:
START TRANSACTION;

-- Define the cursor1 cursor:
CURSOR cursor1 FOR SELECT * FROM tpcds.reason;

-- Skip the first three rows of cursor1:
MOVE FORWARD 3 FROM cursor1;

-- Fetch the first four rows from cursor1:
FETCH 4 FROM cursor1;
r_reason_sk | r_reason_id |                               r_reason_desc
-----+-----
4 | AAAAAAAAAEAAAAAAAA | Not the product that was
ordred
5 | AAAAAAAAFAAAAAAAA | Parts missing
6 | AAAAAAAGAAAAAAAA | Does not work with a product that I
have
7 | AAAAAAAHAAAAAAAA | Gift
exchange
(4 rows)

-- Close the cursor:
CLOSE cursor1;

-- End the transaction:
END;
```



## Helpful Links

[CLOSE, FETCH](#)

# 12.85 REINDEX

## Function

**REINDEX** rebuilds an index using the data stored in the index's table, replacing the old copy of the index.

There are several scenarios in which **REINDEX** can be used:

- An index has become corrupted, and no longer contains valid data.
- An index has become "bloated", that is, it contains many empty or nearly-empty pages.
- You have altered a storage parameter (such as fillfactor) for an index, and wish to ensure that the change has taken full effect.

An index build with the **CONCURRENTLY** option failed, leaving an "invalid" index.

## Precautions

Index reconstruction of the **REINDEX DATABASE** or **SYSTEM** type cannot be performed in transaction blocks.

## Syntax

- Rebuild a general index.  
`REINDEX { INDEX | TABLE | DATABASE | SYSTEM } name [ FORCE ];`
- Rebuild an index partition.  
`REINDEX { | TABLE } name  
PARTITION partition_name [ FORCE ];`

## Parameter Description

- **INDEX**  
Recreates the specified index.
- **TABLE**  
Recreates all indexes of the specified table. If the table has a secondary TOAST table, that is reindexed as well.
- **DATABASE**  
Recreates all indexes within the current database.
- **SYSTEM**  
Recreates all indexes on system catalogs within the current database. Indexes on user tables are not processed.
- **name**  
Name of the specific index, table, or database to be reindexed. Index and table names can be schema-qualified.

 NOTE

**REINDEX DATABASE** and **SYSTEM** can create indexes for only the current database. Therefore, **name** must be the same as the current database name.

- **FORCE**  
This is an obsolete option. It is ignored if specified.
- **partition\_name**  
Specifies the name of the partition or index partition to be reindexed.  
Value range:
  - If it is **REINDEX INDEX**, specify the name of an index partition.
  - If it is **REINDEX TABLE**, specify the name of a partition.

---

**NOTICE**

Index reconstruction of the **REINDEX DATABASE** or **SYSTEM** type cannot be performed in transaction blocks.

---

## Examples

```
-- Create a row-store table tpcds.customer_t1 and create an index on the c_customer_sk column in the table:
CREATE TABLE tpcds.customer_t1
(
  c_customer_sk      integer      not null,
  c_customer_id     char(16)     not null,
  c_current_demo_sk integer              ,
  c_current_hdemo_sk integer        ,
  c_current_addr_sk integer          ,
  c_first_shipto_date_sk integer      ,
  c_first_sales_date_sk integer      ,
  c_salutation      char(10)      ,
  c_first_name      char(20)      ,
  c_last_name       char(30)      ,
  c_preferred_cust_flag char(1)    ,
  c_birth_day       integer        ,
  c_birth_month     integer        ,
  c_birth_year      integer        ,
  c_birth_country   varchar(20)    ,
  c_login           char(13)       ,
  c_email_address   char(50)       ,
  c_last_review_date char(10)
)
WITH (orientation = row)
DISTRIBUTE BY HASH (c_customer_sk);

CREATE INDEX tpcds_customer_index1 ON tpcds.customer_t1 (c_customer_sk);

INSERT INTO tpcds.customer_t1 SELECT * FROM tpcds.customer WHERE c_customer_sk < 10;

-- Rebuild a single index:
REINDEX INDEX tpcds.tpcds_customer_index1;

-- Rebuild all indexes on the tpcds.customer_t1 table:
REINDEX TABLE tpcds.customer_t1;

-- Delete the tpcds.customer_t1 table:
DROP TABLE tpcds.customer_t1;
```

## 12.86 RESET

### Function

**RESET** restores run-time parameters to their default values. The default values are parameter default values compiled in the **postgresql.conf** configuration file.

**RESET** is an alternative spelling for:

**SET configuration\_parameter TO DEFAULT**


### Precautions

**RESET** and **SET** have the same transaction behavior. Their impact will be rolled back.

### Syntax

```
RESET {configuration_parameter | CURRENT_SCHEMA | TIME_ZONE | TRANSACTION ISOLATION LEVEL |  
SESSION AUTHORIZATION | ALL};
```

### Parameter Description

- **configuration\_parameter**  
Specifies the name of a settable run-time parameter.  
Value range: Run-time parameters. You can view them by running the **SHOW ALL** command.  
 **NOTE**  
Some parameters that viewed by **SHOW ALL** cannot be set by **SET**. For example, **max\_datanodes**.
- **CURRENT\_SCHEMA**  
Specifies the current schema.
- **TIME\_ZONE**  
Specifies the time zone.
- **TRANSACTION ISOLATION LEVEL**  
Specifies the transaction isolation level.
- **SESSION AUTHORIZATION**  
Specifies the session authorization.
- **ALL**  
Resets all settable run-time parameters to default values.

### Examples

```
-- Reset timezone to the default value:  
RESET timezone;  
  
-- Set all parameters to their default values:  
RESET ALL;
```

## Helpful Links

[SET](#), [SHOW](#)

# 12.87 SET

## Function

**SET** modifies a run-time parameter.

## Precautions

Most run-time parameters can be modified by executing **SET**. Some parameters cannot be modified after a server or session starts.

## Syntax

- Set the system time zone.  
`SET [ SESSION | LOCAL ] TIME ZONE { timezone | LOCAL | DEFAULT };`
- Set the schema of the table.  
`SET [ SESSION | LOCAL ]  
{CURRENT_SCHEMA { TO | = } { schema | DEFAULT }  
| SCHEMA 'schema'};`
- Set client encoding.  
`SET [ SESSION | LOCAL ] NAMES encoding_name;`
- Set XML parsing mode.  
`SET [ SESSION | LOCAL ] XML OPTION { DOCUMENT | CONTENT };`
- Set other running parameters.  
`SET [ LOCAL | SESSION ]  
{ {config_parameter { { TO | = } { value | DEFAULT }  
| FROM CURRENT }}};`

## Parameter Description

- **SESSION**  
Indicates that the specified parameters take effect for the current session. This is the default value if neither **SESSION** nor **LOCAL** appears.  
If **SET** or **SET SESSION** is executed within a transaction that is later aborted, the effects of the **SET** command disappear when the transaction is rolled back. Once the surrounding transaction is committed, the effects will persist until the end of the session, unless overridden by another **SET**.
- **LOCAL**  
Indicates that the specified parameters take effect for the current transaction. After **COMMIT** or **ROLLBACK**, the session-level setting takes effect again.  
The effects of **SET LOCAL** last only till the end of the current transaction, whether committed or not. A special case is **SET** followed by **SET LOCAL** within a single transaction: the **SET LOCAL** value will be seen until the end of the transaction, but afterwards (if the transaction is committed) the **SET** value will take effect.
- **TIME ZONE timezone**  
Indicates the local time zone for the current session.

Value range: A valid local time zone. The corresponding run-time parameter is **TimeZone**. The default value is **PRC**.

- **CURRENT\_SCHEMA**

**schema**

Indicates the current schema.

Value range: An existing schema name.

- **SCHEMA schema**

Indicates the current schema. Here the schema is a string.

Example: set schema 'public';

- **NAMES encoding\_name**

Indicates the client character encoding name. This command is equivalent to **set client\_encoding to encoding\_name**.

Value range: A valid character encoding name. The run-time parameter corresponding to this option is **client\_encoding**. The default encoding is **UTF8**.

- **XML OPTION option**

Indicates the XML resolution mode.

Value range: **CONTENT** (default), **DOCUMENT**

- **config\_parameter**

Indicates the configurable run-time parameters. You can use **SHOW ALL** to view available run-time parameters.

 **NOTE**

Some parameters that viewed by **SHOW ALL** cannot be set by **SET**. For example, **max\_datanodes**.

- **value**

Indicates the new value of the **config\_parameter** parameter. This parameter can be specified as string constants, identifiers, numbers, or comma-separated lists of these. **DEFAULT** can be written to indicate resetting the parameter to its default value.

## Examples

```
-- Set the search path of a schema:  
SET search_path TO tpeds, public;  
  
-- Set the date style to the traditional POSTGRES style (date placed before month):  
SET datestyle TO postgres;
```

## Helpful Links

[RESET, SHOW](#)

## 12.88 SET CONSTRAINTS

### Function

**SET CONSTRAINTS** sets the behavior of constraint checking within the current transaction.

**IMMEDIATE** constraints are checked at the end of each statement. **DEFERRED** constraints are not checked until transaction commit. Each constraint has its own **IMMEDIATE** or **DEFERRED** mode.

Upon creation, a constraint is given one of three characteristics **DEFERRABLE INITIALLY DEFERRED**, **DEFERRABLE INITIALLY IMMEDIATE**, or **NOT DEFERRABLE**. The third class is always **IMMEDIATE** and is not affected by the **SET CONSTRAINTS** command. The first two classes start every transaction in specified modes, but its behaviors can be changed within a transaction by **SET CONSTRAINTS**.

**SET CONSTRAINTS** with a list of constraint names changes the mode of just those constraints (which must all be deferrable). If multiple constraints match a name, the name is affected by all of these constraints. **SET CONSTRAINTS ALL** changes the modes of all deferrable constraints.

When **SET CONSTRAINTS** changes the mode of a constraint from **DEFERRED** to **IMMEDIATE**, the new mode takes effect retroactively: any outstanding data modifications that would have been checked at the end of the transaction are instead checked during the execution of the **SET CONSTRAINTS** command. If any such constraint is violated, the **SET CONSTRAINTS** fails (and does not change the constraint mode). Therefore, **SET CONSTRAINTS** can be used to force checking of constraints to occur at a specific point in a transaction.

Only foreign key constraints are affected by this setting. Check and unique constraints are always checked immediately when a row is inserted or modified.

### Precautions

**SET CONSTRAINTS** sets the behavior of constraint checking only within the current transaction. Therefore, if you execute this command outside of a transaction block (**START TRANSACTION/COMMIT** pair), it will not appear to have any effect.

### Syntax

```
SET CONSTRAINTS { ALL | { name } [, ...] } { DEFERRED | IMMEDIATE };
```

### Parameter Description

- **name**  
Specifies the constraint name.  
Value range: an existing constraint name, which can be found in the system catalog **pg\_constraint**.
- **ALL**

- Indicates all constraints.
- **DEFERRED**  
Indicates that constraints are not checked until transaction commit.
- **IMMEDIATE**  
Indicates that constraints are checked at the end of each statement.

## Examples

```
-- Set that constraints are checked when a transaction is committed.  
SET CONSTRAINTS ALL DEFERRED;
```

# 12.89 SET ROLE

## Function

**SET ROLE** sets the current user identifier of the current session.

## Precautions

- Users of the current session must be members of specified **rolename**, but the system administrator can choose any roles.
- Executing this command may add rights of a user or restrict rights of a user. If the role of a session user has the **INHERITS** attribute, it automatically has all rights of roles that **SET ROLE** enables the role to be. In this case, **SET ROLE** physically deletes all rights directly granted to session users and rights of its belonging roles and only leaves rights of the specified roles. If the role of the session user has the **NOINHERITS** attribute, **SET ROLE** deletes rights directly granted to the session user and obtains rights of the specified role.

## Syntax

- **SET ROLE** sets the current user identifier of the current session.  
`SET [ SESSION | LOCAL ] ROLE role_name PASSWORD 'password';`
- Reset the current user identifier to that of the current session.  
`RESET ROLE;`

## Parameter Description

- **SESSION**  
Specifies that the command takes effect only for the current session. This parameter is used by default.  
Value range: A string. It must comply with the naming convention rule.
- **LOCALE**  
Indicates that the specified command takes effect only for the current transaction.
- **role\_name**  
Specifies the role name.  
Value range: A string. It must comply with the naming convention rule.
- **password**

Specifies the password of a role. It must comply with the password convention.

- **RESET ROLE**  
Resets the current user identifier.

## Examples

```
-- Create role paul:  
CREATE ROLE paul IDENTIFIED BY 'Bigdata123@';  
  
-- Set the current user to paul:  
SET ROLE paul PASSWORD 'Bigdata123@';  
  
-- View the current session user and the current user:  
SELECT SESSION_USER, CURRENT_USER;  
  
-- Reset the current user:  
RESET role;  
  
-- Delete the user:  
DROP USER paul;
```

## 12.90 SET SESSION AUTHORIZATION

### Function

**SET SESSION AUTHORIZATION** sets the session user identifier and the current user identifier of the current SQL session to a specified user.

### Precautions

The session identifier can be changed only when the initial session user has the system administrator rights. Otherwise, the system supports the command only when the authenticated user name is specified.

### Syntax

- **SET SESSION AUTHORIZATION** sets the session user identifier and the current user identifier of the current session.  
`SET [ SESSION | LOCAL ] SESSION AUTHORIZATION role_name PASSWORD 'password';`
- Reset the identifiers of the session and current users to the initially authenticated user names.  
`{SET [ SESSION | LOCAL ] SESSION AUTHORIZATION DEFAULT  
| RESET SESSION AUTHORIZATION};`

### Parameter Description

- **SESSION**  
Indicates that the specified parameters take effect for the current session.  
Value range: A string. It must comply with the naming convention.
- **LOCALE**  
Indicates that the specified command takes effect only for the current transaction.
- **role\_name**



User name.

Value range: A string. It must comply with the naming convention.

- **password**  
Specifies the password of a role. It must comply with the password convention.
- **DEFAULT**  
Reset the identifiers of the session and current users to the initially authenticated user names.

## Examples

```
-- Create role paul:  
CREATE ROLE paul IDENTIFIED BY 'Bigdata123@';  
  
-- Set the current user to paul:  
SET SESSION AUTHORIZATION paul password 'Bigdata123@';  
  
-- View the current session user and the current user:  
SELECT SESSION_USER, CURRENT_USER;  
  
-- Reset the current user:  
RESET SESSION AUTHORIZATION;  
  
-- Delete the user:  
DROP USER paul;
```

## Helpful Links

[SET ROLE](#)

# 12.91 SHOW

## Function

**SHOW** shows the current value of a run-time parameter.

## Precautions

None

## Syntax

```
SHOW  
{  
  configuration_parameter |  
  CURRENT_SCHEMA |  
  TIME_ZONE |  
  TRANSACTION_ISOLATION_LEVEL |  
  SESSION_AUTHORIZATION |  
  ALL  
};
```

## Parameter description

See [Parameter Description](#) in **RESET**.

## Examples

```
-- Show the value of timezone:  
SHOW timezone;  
  
-- Show all parameters:  
SHOW ALL;
```

## Helpful Links

[SET, RESET](#)

# 12.92 TRUNCATE

## Function

**TRUNCATE** quickly removes all rows from a database table.

It has the same effect as an unqualified **DELETE** on each table, but it is faster since it does not actually scan the tables. This is most useful on large tables.

## Precautions

- **TRUNCATE TABLE** has the same function as a **DELETE** statement with no **WHERE** clause, emptying a table.
- **TRUNCATE TABLE** uses less system and transaction log resources as compared with **DELETE**.
  - **DELETE** deletes a row each time, and records the deletion of each row in the transaction log.
  - **TRUNCATE TABLE** deletes all rows in a table by releasing the data page storing the table data, and records the releasing of the data page only in the transaction log.
- The differences between **TRUNCATE**, **DELETE**, and **DROP** are as follows:
  - **TRUNCATE TABLE** deletes content, releases space, but does not delete definitions.
  - **DELETE TABLE** deletes content, but does not delete definitions nor release space.
  - **DROP TABLE** deletes content and definitions, and releases space.

## Syntax

- **TRUNCATE** empties a table or set of tables.

```
TRUNCATE [ TABLE ] [ ONLY ] { [[database_name.]schema_name.]table_name [ * ] } [, ... ]  
[ CONTINUE IDENTITY ] [ CASCADE | RESTRICT ];
```

- Truncate the data in a partition.

```
ALTER TABLE [ IF EXISTS ] { [ ONLY ] [[database_name.]schema_name.]table_name  
| table_name *  
| ONLY ( table_name ) }  
TRUNCATE PARTITION { partition_name  
| FOR ( partition_value [, ...] ) };
```

## Parameter Description

- **ONLY**  
If **ONLY** is specified, only the specified table is cleared. Otherwise, the table and all its subtables (if any) are cleared.
- **database\_name**  
Database name of the target table
- **schema\_name**  
Schema name of the target table
- **table\_name**  
Specifies the name (optionally schema-qualified) of a target table.  
Value range: An existing table name.
- **CONTINUE IDENTITY**  
Does not change the values of sequences. This is the default.
- **CASCADE | RESTRICT**
  - **CASCADE**: automatically truncates all tables that have foreign-key references to any of the named tables, or to any tables added to the group due to **CASCADE**.
  - **RESTRICT** (default): refuses to truncate if any of the tables have foreign-key references from tables that are not listed in the command.
- **partition\_name**  
Indicates the partition in the target partition table.  
Value range: An existing partition name.
- **partition\_value**  
Specifies the value of the specified partition key.  
The value specified by **PARTITION FOR** can uniquely identify a partition.  
Value range: The partition key of the partition to be deleted.

---

### NOTICE

When the **PARTITION FOR** clause is used, the entire partition where **partition\_value** is located is cleared.

---

## Examples

```
-- Create a table:
CREATE TABLE tpcds.reason_t1 AS TABLE tpcds.reason;

-- Truncate the tpcds.reason_t1 table:
TRUNCATE TABLE tpcds.reason_t1;

-- Delete the tables:
DROP TABLE tpcds.reason_t1;
-- Create a partitioned table:
CREATE TABLE tpcds.reason_p
(
  r_reason_sk integer,
  r_reason_id character(16),
  r_reason_desc character(100)
```

```
)PARTITION BY RANGE (r_reason_sk)
(
  partition p_05_before values less than (05),
  partition p_15 values less than (15),
  partition p_25 values less than (25),
  partition p_35 values less than (35),
  partition p_45_after values less than (MAXVALUE)
);

-- Insert data:
INSERT INTO tpcds.reason_p SELECT * FROM tpcds.reason;

-- Clear the p_05_before partition:
ALTER TABLE tpcds.reason_p TRUNCATE PARTITION p_05_before;

-- Clear the p_15 partition:
ALTER TABLE tpcds.reason_p TRUNCATE PARTITION for (13);

-- Clear the partitioned table:
TRUNCATE TABLE tpcds.reason_p;

-- Delete the tables:
DROP TABLE tpcds.reason_p;
```

## 12.93 VACUUM

### Function

**VACUUM** reclaims storage space occupied by tables or B-tree indexes. In normal database operation, rows that have been deleted are not physically removed from their table; they remain present until a **VACUUM** is done. Therefore, it is necessary to execute **VACUUM** periodically, especially on frequently-updated tables.

### Precautions

- With no table specified, **VACUUM** processes all the tables that the current user has permission to vacuum in the current database. With a table specified, **VACUUM** processes only that table.
- To vacuum a table, you must ordinarily be the table's owner or the system administrator. However, database owners are allowed to **VACUUM** all tables in their databases, except shared catalogs. (The restriction for shared catalogs means that a true database-wide **VACUUM** can only be executed by the system administrator). **VACUUM** skips over any tables that the calling user does not have the permission to vacuum.
- **VACUUM** cannot be executed inside a transaction block.
- It is recommended that active production databases be vacuumed frequently (at least nightly), in order to remove dead rows. After adding or deleting a large number of rows, it might be a good idea to execute the **VACUUM ANALYZE** command for the affected table. This will update the system catalogs with the results of all recent changes, and allow the query planner to make better choices in planning queries.
- **FULL** is recommended only in special scenarios. For example, you wish to physically narrow the table to decrease the occupied disk space after deleting most rows of a table. **VACUUM FULL** usually shrinks more table size than **VACUUM**. If the physical space usage does not decrease after you run the command, check whether there are other active transactions (that have

started before you delete data transactions and not ended before you run **VACUUM FULL**). If there are such transactions, run this command again when the transactions quit.

- **VACUUM** causes a substantial increase in I/O traffic, which might cause poor performance for other active sessions. Therefore, it is sometimes advisable to use the cost-based **VACUUM** delay feature.
- When **VERBOSE** is specified, **VACUUM** prints progress messages to indicate which table is currently being processed. Various statistics about the tables are printed as well. However, if the **VERBOSE** option is specified in **VACUUM** executed for column-store tables, no output will be displayed.
- When the option list is surrounded by parentheses, the options can be written in any order. If there are no brackets, the options must be given in the order displayed in the syntax.
- **VACUUM** and **VACUUM FULL** clear deleted tuples after the delay specified by **vacuum\_defer\_cleanup\_age**.
- **VACUUM ANALYZE** executes a **VACUUM** operation and then an **ANALYZE** operation for each selected table. This is a handy combination form for routine maintenance scripts.
- Plain **VACUUM** (without **FULL**) recycles space and makes it available for reuse. This form of the command can operate in parallel with normal reading and writing of the table, as an exclusive lock is not obtained. **VACUUM FULL** executes wider processing, including moving rows across blocks to compress tables so they occupy minimum number of disk blocks. This form is much slower and requires an exclusive lock on each table while it is being processed.
- When you do **VACUUM** to a column-store table, the following operations are internally performed: data in the delta table is migrated to the primary table, and the delta and desc tables of the primary table are vacuumed. **VACUUM** does not reclaim the storage space of the delta table. To reclaim it, do **VACUUM DELTAMERGE** to the column-store table.
- If you perform **VACUUM FULL** when a long-running query accesses a system table, the long-running query may prevent **VACUUM FULL** from accessing the system table. As a result, the connection times out and an error is reported.

## Syntax

- Reclaim space and update statistics information, with no requirements for the order of keywords.  
`VACUUM [ ( { FULL | FREEZE | VERBOSE | {ANALYZE | ANALYSE } } [... ] ) ]  
[ table_name [ ( column_name [, ... ] ) ] ] [ PARTITION ( partition_name ) ] ;`
- Reclaim space, without updating statistics information.  
`VACUUM [ FULL [COMPACT] ] [ FREEZE ] [ VERBOSE ] [ table_name ] [ PARTITION  
( partition_name ) ] ;`
- Reclaim space and update statistics information, with a specific order of keywords required.  
`VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] { ANALYZE | ANALYSE } [ VERBOSE ]  
[ table_name [ ( column_name [, ... ] ) ] ] [ PARTITION ( partition_name ) ] ;`
- For HDFS and column-store tables, migrate data from the delta table to the primary table.  
`VACUUM DELTAMERGE [ table_name ] ;`
- For HDFS tables, delete the empty value partition directory of HDFS table in HDFS storage.

```
VACUUM HDFSDIRECTORY [ table_name ];
```

## Parameter Description

- **FULL**

Selects "FULL" vacuum, which can reclaim more space, but takes much longer and exclusively locks the table.

**FULL** options can also contain the **COMPACT** parameter, which is only used for the HDFS table. Specifying the **COMPACT** parameter improves **VACUUM FULL** operation performance.

**COMPACT** and **PARTITION** cannot be used at the same time.

 **NOTE**

Using **FULL** will cause statistics missing. To collect statistics, add the keyword **ANALYZE** to **VACUUM FULL**.

- **FREEZE**

Is equivalent to executing **VACUUM** with the **vacuum\_freeze\_min\_age** parameter set to **zero**.

- **VERBOSE**

Prints a detailed vacuum activity report for each table.

- **ANALYZE | ANALYSE**

Updates statistics used by the planner to determine the most efficient way to execute a query.

- **table\_name**

Indicates the name (optionally schema-qualified) of a specific table to vacuum.

Value range: The name of a specific table to vacuum. Defaults are all tables in the current database.

- **column\_name**

Indicates the name of a specific field to analyze.

Value range: Indicates the name of a specific field to analyze. Defaults are all columns.

- **PARTITION**

HDFS table does not support **PARTITION**. **COMPACT** and **PARTITION** cannot be used at the same time.

- **partition\_name**

Indicates the partition name of a specific table to vacuum. Defaults are all partitions.

- **DELTAMERGE**

(For HDFS and column-store tables) Migrates data from the delta table to primary tables. If the data volume of the delta table is less than 60,000 rows, the data will not be migrated. Otherwise, the data will be migrated to HDFS, and the delta table will be cleared by **TRUNCATE**. For a column-store table, this operation is controlled by **enable\_delta\_store** and **deltarow\_threshold** (for details, see [Parameter Description](#)).

 **NOTE**

The following DFX functions are provided to return the data storage in the delta table of a column-store table (for an HDFS table, it can be returned by **EXPLAIN ANALYZE**):

- `pgxc_get_delta_info(TEXT)`: The input parameter is a column-store table name. The delta table information on each node is collected and displayed, including the number of active tuples, table size, and maximum block ID.
  - `get_delta_info(TEXT)`: The input parameter is a column-store table name. The system summarizes the results returned from `pgxc_get_delta_info` and returns the total number of active tuples, total table size, and maximum block ID in the delta table.
- **HDFSDIRECTORY**  
Deletes the empty value partition directory of HDFS table in HDFS storage for HDFS table.

## Examples

```
-- Create an index on the tpcds.reason table:  
CREATE UNIQUE INDEX ds_reason_index1 ON tpcds.reason(r_reason_sk);  
  
-- Do VACUUM to the tpcds.reason table that has indexes:  
VACUUM (VERBOSE, ANALYZE) tpcds.reason;  
  
-- Drop an index:  
DROP INDEX ds_reason_index1 CASCADE;  
DROP TABLE tpcds.reason;
```

# 13 DML Syntax

---

## 13.1 DML Syntax Overview

Data Manipulation Language (DML) is used to perform operations on data in database tables, such as inserting, updating, querying, or deleting data.

### Insert Data

Inserting data refers to adding one or multiple records to a database table. For details, see [INSERT](#).

### Updating Data

Modifying data refers to modifying one or multiple records in a database table. For details, see [UPDATE](#).

### Querying Data

The database query statement **SELECT** is used to search required information in a database. For details, see [SELECT](#).

### Deleting Data

GaussDB(DWS) provides two statements for deleting data from database tables. To delete data meeting specified conditions from a database table, see [DELETE](#). To delete all data from a database table, see [TRUNCATE](#).

**TRUNCATE** can quickly delete all data from a database table, which achieves the effect same as that running **DELETE** to delete data without specifying conditions from each table. Deletion efficiency using **TRUNCATE** is faster because **TRUNCATE** does not scan tables. Therefore, **TRUNCATE** is useful in large tables.

### Copying Data

GaussDB(DWS) provides a statement for copying data between tables and files. For details, see [COPY](#).



## Locking a Table

GaussDB(DWS) provides multiple lock modes to control concurrent accesses to table data. For details, see [LOCK](#).

## Invoking a Function

GaussDB(DWS) provides three statements for invoking functions. These statements are the same in the syntax structure. For details, see [CALL](#).

# 13.2 CALL

## Function

**CALL** calls defined functions or stored procedures.

## Precautions

None

## Syntax

```
CALL [schema.] {func_name| procedure_name} ( param_expr );
```

## Parameter Description

- **schema**  
Specifies the name of the schema where a function or stored procedure is located.
- **func\_name**  
Specifies the name of the function or stored procedure to be called.  
Value range: an existing function name
- **param\_expr**  
Specifies a list of parameters in the function. Use **:=** or **=>** to separate a parameter name and its value. This method allows parameters to be placed in any order. If only parameter values are in the list, the value order must be the same as that defined in the function or stored procedure.  
Value range: names of existing function or stored procedure parameters

### NOTE

The parameters include input parameters (whose name and type are separated by **IN**) and output parameters (whose name and type are separated by **OUT**). When you run the **CALL** statement to call a function or stored procedure, the parameter list must contain an output parameter for non-overloaded functions. You can set the output parameter to a variable or any constant. For details, see [Examples](#). For an overloaded package function, the parameter list can have no output parameter, but the function may not be found. If an output parameter is contained, it must be a constant.

## Examples

```
-- Create the func_add_sql function to compute the sum of two integers and return the result:  
CREATE FUNCTION func_add_sql(num1 integer, num2 integer) RETURN integer
```

```
AS
BEGIN
RETURN num1 + num2;
END;
/

-- Transfer based on parameter values:
CALL func_add_sql(1, 3);

-- Transfer based on the naming flags:
CALL func_add_sql(num1 => 1,num2 => 3);
CALL func_add_sql(num2 := 2, num1 := 3);

-- Delete the function:
DROP FUNCTION func_add_sql;

-- Create a function with output parameters:
CREATE FUNCTION func_increment_sql(num1 IN integer, num2 IN integer, res OUT integer)
RETURN integer
AS
BEGIN
res := num1 + num2;
END;
/

-- Set output parameters to constants:
CALL func_increment_sql(1,2,1);

-- Set output parameters to variables:
DECLARE
res int;
BEGIN
func_increment_sql(1, 2, res);
dbms_output.put_line(res);
END;
/

-- Create overloaded functions:
create or replace procedure package_func_overload(col int, col2 out int) package
as
declare
col_type text;
begin
col := 122;
dbms_output.put_line('two out parameters ' || col2);
end;
/

create or replace procedure package_func_overload(col int, col2 out varchar)
package
as
declare
col_type text;
begin
col2 := '122';
dbms_output.put_line('two varchar parameters ' || col2);
end;
/

-- Call the functions:
call package_func_overload(1, 'test');
call package_func_overload(1, 1);

-- Delete the functions:
DROP FUNCTION func_increment_sql;
```

## 13.3 COPY

### Function

**COPY** copies data between tables and files.

**COPY FROM** copies data from a file to a table. **COPY TO** copies data from a table to a file.

### Precautions

- **COPY FROM FILENAME** and **COPY TO FILENAME** options are unavailable.
- **COPY** applies to only tables and does not apply to views.
- To insert data to a table, you must have the permission to insert data.
- If a list of columns are specified, **COPY** will only copy the data in the specified columns to or from the file. If there are any columns in the table that are not in the column list, **COPY FROM** will insert the default values for those columns.
- If a source data file is specified, the file must be accessible from the server. If **STDIN** is specified, data is transmitted between the client and server. Separate columns by pressing **Tab**. Enter **\.** in a new line to indicate the end of input.
- If the number of columns in a row of the data file is smaller or larger than the expected number, **COPY FROM** displays an error message.
- A backslash and a period (**\.**) indicate the end of data. The end identifier is not required for reading data from a file and is required for copying data between client applications.
- In **COPY FROM**, **"\N"** indicates an empty character string, and **"\\N"** indicates the actual data **"\N"**.
- **COPY FROM** does not support pre-processing of data during data import, for example, expression calculation and default value filling. If you need to perform pre-processing during data import, you need to import the data to a temporary table, and then run SQL statements to insert data to the table using expression or function operations. However, this method may cause I/O expansion, deteriorating data import performance.
- Transactions will be rolled back when data format errors occur during **COPY FROM** execution. In this case, error information is insufficient so you cannot easily locate the incorrect data from a large amount of raw data.
- **COPY FROM** and **COPY TO** apply to low concurrency and local data import and export in small amount.

### Syntax

- Copy the data from a file to a table.

```
COPY table_name [ ( column_name [, ...] ) ]
FROM { 'filename' | STDIN }
[ [ USING ] DELIMITERS 'delimiters' ]
[ WITHOUT ESCAPING ]
[ LOG ERRORS ]
[ LOG ERRORS data ]
[ REJECT LIMIT 'limit' ]
```

```
[ [ WITH ] ( option [, ...] ) ]
| copy_option
| FIXED FORMATTER ( { column_name( offset, length ) } [, ...] ) [ ( option [, ...] ) | copy_option
[ ...] ]];
```

#### NOTE

In the SQL syntax, **FIXED, FORMATTER ( { column\_name( offset, length ) } [, ...] )**, and **[ ( option [, ...] ) | copy\_option [ ...] ]** can be in any sequence.

- Copy the data from a table to a file.

```
COPY table_name [ ( column_name [, ...] ) ]
TO { 'filename' | STDOUT }
[ [ USING ] DELIMITERS 'delimiters' ]
[ WITHOUT ESCAPING ]
[ [ WITH ] ( option [, ...] ) ]
| copy_option
| FIXED FORMATTER ( { column_name( offset, length ) } [, ...] ) [ ( option [, ...] ) | copy_option
[ ...] ]];
```

```
COPY query
TO { 'filename' | STDOUT }
[ WITHOUT ESCAPING ]
[ [ WITH ] ( option [, ...] ) ]
| copy_option
| FIXED FORMATTER ( { column_name( offset, length ) } [, ...] ) [ ( option [, ...] ) | copy_option
[ ...] ]];
```

#### NOTE

1. The syntax constraints of **COPY TO** are as follows:
  - (**query**) is incompatible with **[USING] DELIMITER**. If the data of **COPY TO** comes from a query result, **COPY TO** cannot specify **[USING] DELIMITERS**.
2. Use spaces to separate **copy\_option** following **FIXED FORMATTER**.
3. **copy\_option** is the native parameter, while **option** is the parameter imported by a compatible foreign table.
4. In the SQL syntax, **FIXED, FORMATTER ( { column\_name( offset, length ) } [, ...] )**, and **[ ( option [, ...] ) | copy\_option [ ...] ]** can be in any sequence.

The syntax of the optional parameter **option** is as follows:

```
FORMAT 'format_name'
| OIDS [ boolean ]
| DELIMITER 'delimiter_character'
| NULL 'null_string'
| HEADER [ boolean ]
| FILEHEADER 'header_file_string'
| FREEZE [ boolean ]
| QUOTE 'quote_character'
| ESCAPE 'escape_character'
| EOL 'newline_character'
| NOESCAPING [ boolean ]
| FORCE_QUOTE { ( column_name [, ...] ) | * }
| FORCE_NOT_NULL ( column_name [, ...] )
| ENCODING 'encoding_name'
| IGNORE_EXTRA_DATA [ boolean ]
| FILL_MISSING_FIELDS [ boolean ]
| COMPATIBLE_ILLEGAL_CHARS [ boolean ]
| DATE_FORMAT 'date_format_string'
| TIME_FORMAT 'time_format_string'
| TIMESTAMP_FORMAT 'timestamp_format_string'
| SMALLDATETIME_FORMAT 'smalldatetime_format_string'
```

The syntax of optional parameter in the **copy\_option** is as follows:

```
IDS
| NULL 'null_string'
| HEADER
| FILEHEADER 'header_file_string'
| FREEZE
```

```
| FORCE_NOT_NULL column_name [, ...]
| FORCE_QUOTE { column_name [, ...] | * }
| BINARY
| CSV
| QUOTE [ AS ] 'quote_character'
| ESCAPE [ AS ] 'escape_character'
| EOL 'newline_character'
| ENCODING 'encoding_name'
| IGNORE_EXTRA_DATA
| FILL_MISSING_FIELDS
| COMPATIBLE_ILLEGAL_CHARS
| DATE_FORMAT 'date_format_string'
| TIME_FORMAT 'time_format_string'
| TIMESTAMP_FORMAT 'timestamp_format_string'
| SMALLDATETIME_FORMAT 'smalldatetime_format_string'
```

## Parameter Description

- **query**  
Indicates that the results are to be copied.  
Value range: a **SELECT** or **VALUES** command in parentheses
- **table\_name**  
Specifies the name (optionally schema-qualified) of an existing table.  
Value range: An existing table name.
- **column\_name**  
Indicates an optional list of columns to be copied.  
Value range: If no column list is specified, all columns of the table will be copied.
- **STDIN**  
Indicates that the input comes from the client application.
- **STDOUT**  
Indicates that output goes to the client application.
- **FIXED**  
Fixes column length. When the column length is fixed, **DELIMITER**, **NULL**, and **CSV** cannot be specified. When **FIXED** is specified, **BINARY**, **CSV**, and **TEXT** cannot be specified by **option** or **copy\_option**.

### NOTE

The definition of fixed length:

1. The column length of each record is the same.
  2. Spaces are added to short columns. Digit type columns must be left-aligned, and character columns must be right-aligned.
  3. No delimiters are used between columns.
- **[USING] DELIMITER 'delimiters'**  
The string that separates columns within each row (line) of the file, and it cannot be larger than 10 bytes.  
Value range: The delimiter cannot include any of the following characters:  
\  
abcdefghijklmnopqrstuvwxyz0123456789  
Value range: The default value is a tab character in text format and a comma in CSV format.

- **WITHOUT ESCAPING**

In **TEXT**, do not escape a backslash (\) and the characters that follow it.  
Value range: text only.

- **LOG ERRORS**

If this parameter is specified, the error tolerance mechanism for data type errors in the **COPY FROM** statement is enabled. Row errors are recorded in the **public.pgxc\_copy\_error\_log** table in the database for future reference.

Value range: A value set while data is imported using **COPY FROM**.

 **NOTE**

The restrictions of this error tolerance parameter are as follows:

- This error tolerance mechanism captures only the data type errors (**DATA\_EXCEPTION**) that occur during data parsing of **COPY FROM** on a CN. Other errors, such as network errors between CNs and DN or expression conversion errors on DNs, are not captured.
  - Before enabling error tolerance for **COPY FROM** for the first time in a database, check whether the **public.pgxc\_copy\_error\_log** table exists. If it does not, call the **copy\_error\_log\_create()** function to create it. If it does, copy its data elsewhere and call the **copy\_error\_log\_create()** function to create the table. For details about columns in the **public.pgxc\_copy\_error\_log** table, see [Table 6-15](#).
  - While a **COPY FROM** statement with specified **LOG ERRORS** is being executed, if **public.pgxc\_copy\_error\_log** does not exist or does not have the table definitions compliant with the predefined in **copy\_error\_log\_create()**, an error will be reported. Ensure that the error table is created using the **copy\_error\_log\_create()** function. Otherwise, **COPY FROM** statements with error tolerance may fail to be run.
  - If existing error tolerance parameters (for example, **IGNORE\_EXTRA\_DATA**) of the **COPY** statement are enabled, the error of the corresponding type will be processed as specified by the parameters and no error will be reported. Therefore, the error table does not contain such error data.
  - The coverage scope of this error tolerance mechanism is the same as that of a GDS foreign table. You are advised to filter query results based on table names or the timestamp of marking the start of **COPY FROM** statement execution. For details about how to process error data, see handling error tables.
- **LOG ERRORS DATA**

The differences between **LOG ERRORS DATA** and **LOG ERRORS** are as follows:

- a. **LOG ERRORS DATA** fills the **rawrecord** field in the error tolerance table.
- b. Only users with the super permission can use the **LOG ERRORS DATA** parameter.

---

 **CAUTION**

If error content is too complex, it may fail to be written to the error tolerance table by using **LOG ERRORS DATA**, causing the task failure.

- **REJECT LIMIT 'limit'**

Used with the **LOG ERROR** parameter to set the upper limit of the tolerated errors in the **COPY FROM** statement. If the number of errors exceeds the limit, later errors will be reported based on the original mechanism.

Value range: a positive integer (1 to INTMAX) or **unlimited**

Default value: If **LOG ERRORS** is not specified, an error will be reported. If **LOG ERRORS** is specified, the default value is **0**.

 **NOTE**

Different from the GDS error tolerance mechanism, in the error tolerance mechanism described in the description of **LOG ERRORS**, the count of **REJECT LIMIT** is calculated based on the number of data parsing errors on the CN where the **COPY FROM** statement is run, not based on the number of errors on each DN.

- **FORMATTER**

Defining the location of each column in the data file in fixed length mode.  
Defining the place of each column in the data file based on column (offset, length) format.

Value range:

- The value of **offset** must be larger than 0. The unit is byte.
- The value of **length** must be larger than 0. The unit is byte.

The total length of all columns must be less than 1 GB.

Replace columns that are not in the file with NULL.

- **OPTION { option\_name ' value ' }**

Specifies all types of parameters of a compatible foreign table.

- **FORMAT**

Specifies the format of the source data file in the foreign table.

Value range: CSV, TEXT, FIXED, and BINARY.

- The CSV file can process newline characters efficiently, but cannot process certain special characters well.
- The TEXT file can process special characters efficiently, but cannot process newline character well.
- The FIXED file can process newline characters in data columns efficiently, but cannot process special characters well.
- All data in the BINARY file is stored/read as binary format rather than as text. It is faster than the text and CSV formats, but a binary-format file is less portable.

Default value: **TEXT**

- **OIDS**

Copies the OID for each row.

 **NOTE**

An error is raised if OIDs are specified for a table that does not have OIDs, or in the case of copying a query.

Value range: **true, on, false, and off**

Default value: **false**

- **DELIMITER**

Specifies the character that separates columns within each row (line) of the file.

 NOTE

- A delimiter cannot be `\r` or `\n`.
- A delimiter cannot be the same as null. The delimiter for CSV cannot be same as quote.
- The delimiter for the TEXT format data cannot contain lowercase letters, digits, or dot (`.`).
- The data length of a single row should be less than 1 GB. If the delimiters are too long and there are too many rows, the length of valid data will be affected.
- You are advised to use multi-characters and invisible characters for delimiters. For example, you can use multi-characters (such as `$$&`) and invisible characters (such as `0x07`, `0x08`, and `0x1b`).
- For a multi-character delimiter, do not use the same characters, for example, `---`.

Value range: multi-character delimiter within 10 bytes.

Default value:

- A tab character in TEXT format
- A comma (`,`) in CSV format
- No delimiter in FIXED format

– NULL

Specifies the string that represents a null value.

Value range:

- The null value cannot be `\r` or `\n`. The maximum length is 100 characters.
- The null value cannot be the same as the delimiter or quote parameter.

Default value:

- an empty string without quotation marks in CSV format
- `\N` in TEXT format

– HEADER

Specifies whether a data file contains a table header. header is available only for CSV and FIXED files.

When data is imported, if **header** is **on**, the first row of a data file will be identified as the header and ignored. If **header** is **off**, the first row will be identified as a data row.

When data is exported, if **header** is **on**, **fileheader** must be specified. If **header** is **off**, an exported file does not contain a header.

Value range: true, on, false, and off

Default value: **false**

– QUOTE

Specifies the quote character for a CSV file.

Default value: double quotation mark (`""`)



 NOTE

- The quote parameter cannot be the same as the delimiter or null parameter.
- The **quote** parameter must be a single one-byte character.
- Invisible characters are recommended as **quote** values, such as 0x07, 0x08, and 0x1b.

## – ESCAPE

This option is allowed only when using CSV format. This must be a single one-byte character.

Default value: double quotation mark (""). If it is the same as the value of **quote**, it will be replaced with \0.

## – EOL 'newline\_character'

Specifies the newline character style of the imported or exported data file.

Value range: multi-character newline characters within 10 bytes. Common newline characters include \r (0x0D), \n (0x0A), and \r\n (0x0D0A). Special newline characters include \$ and #.

 NOTE

- The **EOL** parameter supports only the TEXT format for data import and export and does not support the CSV or FIXED format for data import. For forward compatibility, the **EOL** parameter can be set to **0x0D** or **0x0D0A** for data export in the CSV and FIXED formats.
  - The value of the **EOL** parameter cannot be the same as that of **DELIMITER** or **NULL**.
  - The **EOL** parameter value cannot contain lowercase letters, digits, or dot (.).
- FORCE\_QUOTE { ( column\_name [, ...] ) | \* }
- Forces quoting to be used for all non-null values in each specified column. This option is allowed only in **COPY TO**, and only when using the CSV format. **NULL** values are not quoted.
- Value range: an existing column
- FORCE\_NOT\_NULL ( column\_name [, ...] )
- Does not match the specified columns' values against the null string. This option is allowed only in **COPY FROM**, and only when using the CSV format.
- Value range: an existing column
- ENCODING
- Specifies that the file is encoded in the **encoding\_name**. If this option is omitted, the current encoding format is used by default.
- IGNORE\_EXTRA\_DATA
- When the number of data source files exceeds the number of foreign table columns, whether ignoring excessive columns at the end of the row. This parameter is available only during data importing.
- Value range: true/on, false/off.
- When this parameter is **true** or **on** and the number of data source files exceeds the number of foreign table columns, excessive columns will be ignored.

- If the parameter is set to **false** or **off**, and the number of data source files exceeds the number of foreign table columns, the following error information will be displayed:  
extra data after last expected column

Default value: **false**

---

#### NOTICE

If the newline character at the end of the row is lost, setting the parameter to **true** will ignore data in the next row.

---

#### - COMPATIBLE\_ILLEGAL\_CHARS

Enables or disables fault tolerance on invalid characters during importing. This parameter is available only for **COPY FROM**.

Value range: true, on, false, and off

- When the parameter is **true** or **on**, invalid characters are tolerated and imported to the database after conversion.
- If the parameter is **false** or **off**, and an error occurs when there are invalid characters, the import will be interrupted.

Default value: **false** or **off**

#### NOTE

The rule of error tolerance when you import invalid characters is as follows:

(1) \0 is converted to a space.

(2) Other invalid characters are converted to question marks.

(3) If **compatible\_illegal\_chars** is set to **true** or **on**, invalid characters are tolerated. If **NULL**, **DELIMITER**, **QUOTE**, and **ESCAPE** are set to a space or question marks. Errors like "illegal chars conversion may confuse COPY escape 0x20" will be displayed to prompt user to modify parameter values that cause confusion, preventing import errors.

#### - FILL\_MISSING\_FIELD

Specifies whether to generate an error message when the last column in a row in the source file is lost during data loading.

Value range: **true**, **on**, **false**, and **off**

Default value: **false** or **off**


#### - DATE\_FORMAT

Imports data of the **DATE** type. The **BINARY** format is not supported. When data of such format is imported, error "cannot specify bulkload compatibility options in BINARY mode" will occur. The parameter is valid only for data importing using the **COPY FROM** option.

Value range: any valid DATE value. For details, see [Date and Time Processing Functions and Operators](#).

#### NOTE

If ORACLE is specified as the compatible database, the DATE format is TIMESTAMP. For details, see **timestamp\_format** below.

- **TIME\_FORMAT**  
Imports data of the TIME type. The BINARY format is not supported. When data of such format is imported, error "cannot specify bulkload compatibility options in BINARY mode" will occur. The parameter is valid only for data importing using the **COPY FROM** option.  
Value range: Valid TIME. Time zones cannot be used. For details, see [Date and Time Processing Functions and Operators](#).
- **TIMESTAMP\_FORMAT**  
Imports data of the TIMESTAMP type. The BINARY format is not supported. When data of such format is imported, error "cannot specify bulkload compatibility options in BINARY mode" will occur. The parameter is valid only for data importing using the **COPY FROM** option.  
Value range: any valid TIMESTAMP value. Time zones are not supported. For details, see [Date and Time Processing Functions and Operators](#).
- **SMALLDATETIME\_FORMAT**  
Imports data of the SMALLDATETIME type. The BINARY format is not supported. When data of such format is imported, error "cannot specify bulkload compatibility options in BINARY mode" will occur. The parameter is valid only for data importing using the **COPY FROM** option.  
Value range: any valid SMALLDATETIME value. For details, see [Date and Time Processing Functions and Operators](#).
- **COPY\_OPTION { option\_name ' value ' }**  
Specifies all types of native parameters of **COPY**.
  - **oids**  
Copies the OID for each row.  
  
 **NOTE**  

An error is raised if OIDs are specified for a table that does not have OIDs, or in the case of copying a query.
  - **NULL null\_string**  
Specifies the string that represents a null value.

---

**NOTICE**

When using **COPY FROM**, any data item that matches this string will be stored as a **NULL** value, so you should make sure that you use the same string as you used with **COPY TO**.

---

Value range:

- The null value cannot be `\r` or `\n`. The maximum length is 100 characters.
- The null value cannot be the same as the delimiter or quote parameter.

Default value:

- `\N` in TEXT format
- an empty string without quotation marks in CSV format
- HEADER  
Specifies whether a data file contains a table header. **header** is available only for CSV and FIXED files.  
When data is imported, if **header** is **on**, the first row of a data file will be identified as the header and ignored. If **header** is **off**, the first row will be identified as a data row.  
When data is exported, if header is **on**, **fileheader** must be specified. If **header** is **off**, an exported file does not contain a header.
- FILEHEADER  
Specifies a file that defines the content in the header for exported data. The file contains data description of each column.

---

**NOTICE**

- This parameter is available only when **header** is **on** or **true**.
- **fileheader** specifies an absolute path.
- The file can contain only one row of header information, and ends with a linefeed. Excess rows will be discarded. (Header information cannot contain linefeeds.)
- The length of the file including the linefeed cannot exceed 1 MB.

- 
- FREEZE  
Sets the **COPY** loaded data row as **frozen**, like these data have executed **VACUUM FREEZE**.  
This is a performance option of initial data loading. The data will be frozen only when the following three requirements are met:
    - The table being loaded has been created or truncated in the current subtransaction before copying.
    - There are no cursors open in the current transaction.
    - There are no original snapshots in the current transaction.

** NOTE**

When **COPY** is completed, all the other sessions will see the data immediately. This violates the normal rules of MVCC visibility and users should be aware of the potential problems this might cause.

- FORCE NOT NULL column\_name [, ...]  
Does not match the specified columns' values against the null string. This option is allowed only in **COPY FROM**, and only when using the CSV format.  
Value range: an existing column
- FORCE QUOTE { column\_name [, ...] | \* }

Forces quoting to be used for all non-NULL values in each specified column. This option is allowed only in **COPY TO**, and only when using the CSV format. **NULL** values are not quoted.

Value range: an existing column

– **BINARY**

The binary format option causes all data to be stored/read as binary format rather than as text. In binary mode, you cannot declare **DELIMITER**, **NULL**, or **CSV**. After specifying **BINARY**, **CSV**, **FIXED** and **TEXT** cannot be specified through **option** or **copy\_option**.

– **CSV**

Enables the CSV mode. After **CSV** is specified, **BINARY**, **FIXED** and **TEXT** cannot be specified through **option** or **copy\_option**.

– **QUOTE [AS] 'quote\_character'**

Specifies the quote character for a CSV file.

Default value: double quotation mark ("").

 **NOTE**

- The quote parameter cannot be the same as the delimiter or null parameter.
- The **quote** parameter must be a single one-byte character.
- Invisible characters are recommended as **quote** values, such as 0x07, 0x08, and 0x1b.

– **ESCAPE [AS] 'escape\_character'**

This option is allowed only when using CSV format. This must be a single one-byte character.

The default value is a double quotation mark ("). If it is the same as the value of **quote**, it will be replaced with **\0**.

– **EOL 'newline\_character'**

Specifies the newline character style of the imported or exported data file.

Value range: multi-character newline characters within 10 bytes.

Common newline characters include **\r** (0x0D), **\n** (0x0A), and **\r\n** (0x0D0A). Special newline characters include **\$** and **#**.

 **NOTE**

- The **EOL** parameter supports only the TEXT format for data import and export. For forward compatibility, the **EOL** parameter can be set to **0x0D** or **0x0D0A** for data export in the CSV and FIXED formats.
  - The value of the **EOL** parameter cannot be the same as that of **DELIMITER** or **NULL**.
  - The **EOL** parameter value cannot contain lowercase letters, digits, or dot (.).
- **ENCODING 'encoding\_name'**
- Specifies that the file is encoded in the **encoding\_name**.
- Value range: a valid encoding format
- Default value: current encoding format of the database
- **IGNORE\_EXTRA\_DATA**

When the number of data source files exceeds the number of foreign table columns, excess columns at the end of the row are ignored. This parameter is available only during data importing.

If you do not use this parameter, and the number of data source files exceeds the number of foreign table columns, the following error information will be displayed:

extra data after last expected column

– **COMPATIBLE\_ILLEGAL\_CHARS**

Specifies error tolerance for invalid characters during importing. Invalid characters are converted before importing. No error message is displayed. The import is not interrupted. The BINARY format is not supported. When data of such format is imported, error "cannot specify bulkload compatibility options in BINARY mode" will occur. The parameter is valid only for data importing using the **COPY FROM** option.

If you do not use this parameter, an error occurs when there is an invalid character, and the import stops.

 **NOTE**

The rule of error tolerance when you import invalid characters is as follows:

(1) \0 is converted to a space.

(2) Other invalid characters are converted to question marks.

(3) Setting **compatible\_illegal\_chars** to **true/on** enables toleration of invalid characters. If **NULL**, **DELIMITER**, **QUOTE**, and **ESCAPE** are set to spaces or question marks, errors like "illegal chars conversion may confuse COPY escape 0x20" will be displayed to prompt the user to modify parameters that may cause confusion, preventing importing errors.

– **FILL\_MISSING\_FIELD**

Specifies whether to generate an error message when the last column in a row in the source file is lost during data loading.

Value range: **true**, **on**, **false**, and **off**

Default value: **false** or **off**

---

**NOTICE**

Do not specify this option. Currently, it does not enable error tolerance, but will make the parser ignore the said errors during data parsing on the CN. Such errors will not be recorded in the COPY error table (enabled using **LOG ERRORS REJECT LIMIT**) but will be reported later by DNs.

---

– **DATE\_FORMAT 'date\_format\_string'**

Imports data of the **DATE** type. The BINARY format is not supported. When data of such format is imported, error "cannot specify bulkload compatibility options in BINARY mode" will occur. The parameter is valid only for data importing using the **COPY FROM** option.

Value range: any valid DATE value. For details, see [Date and Time Processing Functions and Operators](#).

 NOTE

If ORACLE is specified as the compatible database, the DATE format is TIMESTAMP. For details, see [timestamp\\_format](#) below.

- TIME\_FORMAT 'time\_format\_string'  
Imports data of the TIME type. The BINARY format is not supported. When data of such format is imported, error "cannot specify bulkload compatibility options in BINARY mode" will occur. The parameter is valid only for data importing using the **COPY FROM** option.  
Value range: Valid TIME. Time zones cannot be used. For details, see [Date and Time Processing Functions and Operators](#).
- TIMESTAMP\_FORMAT 'timestamp\_format\_string'  
Imports data of the TIMESTAMP type. The BINARY format is not supported. When data of such format is imported, error "cannot specify bulkload compatibility options in BINARY mode" will occur. The parameter is valid only for data importing using the **COPY FROM** option.  
Value range: any valid TIMESTAMP value. Time zones are not supported. For details, see [Date and Time Processing Functions and Operators](#).
- SMALLDATETIME\_FORMAT 'smalldatetime\_format\_string'  
Imports data of the SMALLDATETIME type. The BINARY format is not supported. When data of such format is imported, error "cannot specify bulkload compatibility options in BINARY mode" will occur. The parameter is valid only for data importing using the **COPY FROM** option.  
Value range: any valid SMALLDATETIME value. For details, see [Date and Time Processing Functions and Operators](#).

The following special backslash sequences are recognized by **COPY FROM**:

- **\b**: Backspace (ASCII 8)
- **\f**: Form feed (ASCII 12)
- **\n**: Newline character (ASCII 10)
- **\r**: Carriage return character (ASCII 13)
- **\t**: Tab (ASCII 9)
- **\v**: Vertical tab (ASCII 11)
- **\digits**: Backslash followed by one to three octal digits specifies the ASCII value is the character with that numeric code.
- **\xdigits**: Backslash followed by an x and one or two hex digits specifies the character with that numeric code.

## Examples

```
-- Copy data from the tpcds.ship_mode file to the /home/omm/ds_ship_mode.dat file:  
COPY tpcds.ship_mode TO '/home/omm/ds_ship_mode.dat';  
  
-- Write tpcds.ship_mode as output to stdout:  
COPY tpcds.ship_mode TO stdout;  
  
-- Create the tpcds.ship_mode_t1 table:  
CREATE TABLE tpcds.ship_mode_t1  
(  
    SM_SHIP_MODE_SK      INTEGER      NOT NULL,  
    SM_SHIP_MODE_ID     CHAR(16)     NOT NULL,  
    SM_TYPE              CHAR(30)     ,
```

```
SM_CODE          CHAR(10)          ,
SM_CARRIER      CHAR(20)          ,
SM_CONTRACT      CHAR(20)
)
WITH (ORIENTATION = COLUMN,COMPRESSION=MIDDLE)
DISTRIBUTE BY HASH(SM_SHIP_MODE_SK);

-- Copy data from stdin to the tpcds.ship_mode_t1 table:
COPY tpcds.ship_mode_t1 FROM stdin;

-- Copy data from the /home/omm/ds_ship_mode.dat file to the tpcds.ship_mode_t1 table:
COPY tpcds.ship_mode_t1 FROM '/home/omm/ds_ship_mode.dat';

-- Copy data from the /home/omm/ds_ship_mode.dat file to the tpcds.ship_mode_t1 table, with the
import format set to TEXT (format 'text'), the delimiter set to \t (delimiter E'\t'), excessive columns
ignored (ignore_extra_data 'true'), and characters not escaped (noescaping 'true'):
COPY tpcds.ship_mode_t1 FROM '/home/omm/ds_ship_mode.dat' WITH(format 'text', delimiter E'\t',
ignore_extra_data 'true', noescaping 'true');

-- Copy data from the /home/omm/ds_ship_mode.dat file to the tpcds.ship_mode_t1 table, with the
import format set to FIXED, fixed-length format specified (FORMATTER(SM_SHIP_MODE_SK(0, 2),
SM_SHIP_MODE_ID(2,16), SM_TYPE(18,30), SM_CODE(50,10), SM_CARRIER(61,20),
SM_CONTRACT(82,20))), excessive columns ignored (ignore_extra_data), and headers included (header).
COPY tpcds.ship_mode_t1 FROM '/home/omm/ds_ship_mode.dat' FIXED
FORMATTER(SM_SHIP_MODE_SK(0, 2), SM_SHIP_MODE_ID(2,16), SM_TYPE(18,30), SM_CODE(50,10),
SM_CARRIER(61,20), SM_CONTRACT(82,20)) header ignore_extra_data;

-- Delete the tpcds.ship_mode_t1 table:
DROP TABLE tpcds.ship_mode_t1;
```

## 13.4 DELETE

### Function

**DELETE** deletes rows that satisfy the **WHERE** clause from the specified table. If the **WHERE** clause does not exist, all rows in the table will be deleted. The result is a valid, but an empty table.

### Precautions

- You must have the **DELETE** permission on the table to delete from it, as well as the **SELECT** permission for any table in the **USING** clause or whose values are read in the **condition**.
- **DELETE** can be used for row-store tables if they have primary key constraints or if the execution plan can be pushed down.
- **DELETE** can be used for column-store tables only if the execution plan can be pushed down.
- For column-store tables, the **RETURNING** clause is currently not supported.

### Syntax

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
DELETE FROM [ ONLY ] table_name [ * ] [ [ AS ] alias ]
    [ USING using_list ]
    [ WHERE condition | WHERE CURRENT OF cursor_name ]
    [ RETURNING { * | { output_expr [ [ AS ] output_name ] } [, ...] } ];
```



## Parameter Description

- **WITH [ RECURSIVE ] with\_query [, ...]**

The **WITH** clause allows you to specify one or more subqueries that can be referenced by name in the primary query, equal to temporary table.

If **RECURSIVE** is specified, it allows a **SELECT** subquery to reference itself by name.

The with\_query detailed format is as follows:  
with\_query\_name [ ( column\_name [, ...] ) ] AS  
( {select | values | insert | update | delete} )  
-- **with\_query\_name** specifies the name of the result set generated by a subquery. Such names can be used to access the result sets of subqueries in a query.

**column\_name** specifies the column name displayed in the subquery result set.

Each subquery can be a **SELECT, VALUES, INSERT, UPDATE** or **DELETE** statement.
- **ONLY**

If **ONLY** is specified, only that table is deleted. If **ONLY** is not specified, this table and all its sub-tables are deleted.
- **table\_name**

Specifies the name (optionally schema-qualified) of a target table.  
Value range: An existing table name.
- **alias**

Specifies the alias for the target table.  
Value range: A string. It must comply with the naming convention rule.
- **using\_list**

using clause
- **condition**

An expression that returns a value of type Boolean. Only rows for which this expression returns **true** will be deleted.
- **WHERE CURRENT OF cursor\_name**

Not supported currently. Only syntax interface is provided.
- **output\_expr**

An expression to be computed and returned by the **DELETE** command after each row is deleted. The expression can use any column names of the table. Write \* to return all columns.
- **output\_name**

A name to use for a returned column.  
Value range: A string. It must comply with the naming convention rule.

## Examples

```
-- Create the tpcds.customer_address_bak table:  
CREATE TABLE tpcds.customer_address_bak AS TABLE tpcds.customer_address;
```

```
-- Delete employees whose ca_address_sk is less than 14888 in the tpcds.customer_address_bak table:  
DELETE FROM tpcds.customer_address_bak WHERE ca_address_sk < 14888;  
  
-- Delete all data in the tpcds.customer_address_bak table:  
DELETE FROM tpcds.customer_address_bak;  
  
-- Delete the tpcds.customer_address_bak table:  
DROP TABLE tpcds.customer_address_bak;
```

## 13.5 EXPLAIN

### Function

**EXPLAIN** shows the execution plan of an SQL statement.

The execution plan shows how the tables referenced by the SQL statement will be scanned, for example, by plain sequential scan or index scan. If multiple tables are referenced, the execution plan also shows what join algorithms will be used to bring together the required rows from each input table.

The most critical part of the display is the estimated statement execution cost, which is the planner's guess at how long it will take to run the statement.

The **ANALYZE** option causes the statement to be executed, not only planned. Then actual runtime statistics are added to the display, including the total elapsed time expended within each plan node (in milliseconds) and the total number of rows it actually returned. This is useful to check whether the planner's estimates are close to reality.

### Precautions

The statement is executed when the **ANALYZE** option is used. To use **EXPLAIN ANALYZE** on an **INSERT**, **UPDATE**, **DELETE**, **CREATE TABLE AS**, or **EXECUTE** statement without letting the command affect your data, use this approach:

```
START TRANSACTION;  
EXPLAIN ANALYZE ...;  
ROLLBACK;
```

### Syntax

- Display the execution plan of an SQL statement, which supports multiple options and has no requirements for the order of options.

```
EXPLAIN [ ( option [, ...] ) ] statement;
```

The syntax of the **option** clause is as follows:

```
ANALYZE [ boolean ] |  
ANALYSE [ boolean ] |  
VERBOSE [ boolean ] |  
COSTS [ boolean ] |  
CPU [ boolean ] |  
DETAIL [ boolean ] |  
NODES [ boolean ] |  
NUM_NODES [ boolean ] |  
BUFFERS [ boolean ] |  
TIMING [ boolean ] |  
PLAN [ boolean ] |  
FORMAT { TEXT | XML | JSON | YAML }
```

- Display the execution plan of an SQL statement, where options are in order.

```
EXPLAIN { [ { ANALYZE | ANALYSE } ] [ VERBOSE ] | PERFORMANCE } statement;
```

## Parameter Description

- **statement**  
Specifies the SQL statement to explain.
- **ANALYZE boolean | ANALYSE boolean**  
Displays the actual run times and other statistics.  
Valid value:
  - **TRUE** (default value): Displays the actual run times and other statistics.
  - **FALSE**: No display.
- **VERBOSE boolean**  
Displays additional information regarding the plan.  
Valid value:
  - **TRUE** (default value): Displays additional information.
  - **FALSE**: No display.
- **COSTS boolean**  
Includes information on the estimated total cost of each plan node, as well as the estimated number of rows and the estimated width of each row.  
Valid value:
  - **TRUE** (default): Displays information on the estimated total cost of each plan node and the estimated width of each row.
  - **FALSE**: No display.
- **CPU boolean**  
Prints information on CPU usage.  
Valid value:
  - **TRUE** (default value): Displays CPU usage information.
  - **FALSE**: No display.
- **DETAIL boolean**  
Prints DN information.  
Valid value:
  - **TRUE** (default value): Prints DN information.
  - **FALSE**: No display.
- **NODES boolean**  
Prints information about the nodes executed by query.  
Valid value:
  - **TRUE** (default): Prints information about executed nodes.
  - **FALSE**: No display.
- **NUM\_NODES boolean**  
Prints the quantity of executing nodes.  
Valid value:
  - **TRUE** (default value): Prints the number of DNs.

- **FALSE**: No display.
- **BUFFERS boolean**  
Includes information on buffer usage.  
Valid value:
  - **TRUE**: Displays information on buffer usage.
  - **FALSE** (default): No display.
- **TIMING boolean**  
Includes the startup time and the time spent on the output node.  
Valid value:
  - **TRUE** (Default): Displays the startup time and the time spent on the output node.
  - **FALSE**: No display.
- **PLAN**  
Specifies whether to store the execution plan in **PLAN\_TABLE**. If this parameter is set to **on**, the execution plan is stored in **PLAN\_TABLE** and is not displayed on the screen. Therefore, this parameter cannot be used together with other parameters when it is set to **on**.  
Valid value:
  - **on**: The execution plan is stored in **PLAN\_TABLE** and is not printed on the screen. It is the default value. If the plan is stored successfully, **EXPLAIN SUCCESS** is returned.
  - **off**: The execution plan is not stored in **PLAN\_TABLE** and is printed on the screen.
- **FORMAT**  
Specifies the output format.  
Value range: **TEXT**, **XML**, **JSON**, and **YAML**.  
Default value: **TEXT**
- **PERFORMANCE**  
This option prints all relevant information in execution.

## Examples

```
-- Create the tpcds.customer_address_p1 table:
CREATE TABLE tpcds.customer_address_p1 AS TABLE tpcds.customer_address;

-- Change the value of explain_perf_mode to normal:
SET explain_perf_mode=normal;

-- Display an execution plan for simple queries in the table:
EXPLAIN SELECT * FROM tpcds.customer_address_p1;
QUERY PLAN
-----
Data Node Scan (cost=0.00..0.00 rows=0 width=0)
Node/s: All datanodes
(2 rows)

-- Generate an execution plan in JSON format (assume explain_perf_mode is set to normal):
EXPLAIN(FORMAT JSON) SELECT * FROM tpcds.customer_address_p1;
QUERY PLAN
-----
[
  +
{
  +
```

```

"Plan": {
  "Node Type": "Data Node Scan",+
  "Startup Cost": 0.00, +
  "Total Cost": 0.00, +
  "Plan Rows": 0, +
  "Plan Width": 0, +
  "Node/s": "All datanodes" +
}
}
]
(1 row)

-- If there is an index and we use a query with an indexable WHERE condition, EXPLAIN might show a
different plan:
EXPLAIN SELECT * FROM tpcds.customer_address_p1 WHERE ca_address_sk=10000;
QUERY PLAN
-----
Data Node Scan (cost=0.00..0.00 rows=0 width=0)
Node/s: dn_6005_6006
(2 rows)

-- Generate an execution plan in YAML format (assume explain_perf_mode is set to normal):
EXPLAIN(FORMAT YAML) SELECT * FROM tpcds.customer_address_p1 WHERE ca_address_sk=10000;
QUERY PLAN
-----
- Plan:
  Node Type: "Data Node Scan"+
  Startup Cost: 0.00 +
  Total Cost: 0.00 +
  Plan Rows: 0 +
  Plan Width: 0 +
  Node/s: "dn_6005_6006"
(1 row)

-- Here is an example of an execution plan with cost estimates suppressed:
EXPLAIN(COSTS FALSE)SELECT * FROM tpcds.customer_address_p1 WHERE ca_address_sk=10000;
QUERY PLAN
-----
Data Node Scan
Node/s: dn_6005_6006
(2 rows)

-- Here is an example of an execution plan for a query that uses an aggregate function:
EXPLAIN SELECT SUM(ca_address_sk) FROM tpcds.customer_address_p1 WHERE ca_address_sk<10000;
QUERY PLAN
-----
Aggregate (cost=18.19..14.32 rows=1 width=4)
-> Streaming (type: GATHER) (cost=18.19..14.32 rows=3 width=4)
Node/s: All datanodes
-> Aggregate (cost=14.19..14.20 rows=3 width=4)
-> Seq Scan on customer_address_p1 (cost=0.00..14.18 rows=10 width=4)
Filter: (ca_address_sk < 10000)
(6 rows)

-- Delete the tpcds.customer_address_p1 table:
DROP TABLE tpcds.customer_address_p1;

```

## Helpful Links

[ANALYZE | ANALYSE](#)

## 13.6 EXPLAIN PLAN

### Function

You can run the **EXPLAIN PLAN** statement to save the information about an execution plan to the **PLAN\_TABLE** table. Different from the **EXPLAIN** statement, **EXPLAIN PLAN** only stores plan information and does not print it on the screen.

### Syntax

```
EXPLAIN PLAN  
[ SET STATEMENT_ID = string ]  
FOR statement ;
```

### Parameter Description

- **PLAN**  
Stores plan information in **PLAN\_TABLE**. If the storing is successful, **EXPLAIN SUCCESS** is returned.
- **STATEMENT\_ID**  
Tags a query. The tag information will be stored in **PLAN\_TABLE**.

#### NOTE

If the **EXPLAIN PLAN** statement does not contain **SET STATEMENT\_ID**, the value of **STATEMENT\_ID** is empty by default. In addition, the value of **STATEMENT\_ID** cannot exceed 30 bytes. Otherwise, an error will be reported.

### Precautions

- **EXPLAIN PLAN** cannot be executed on DNs.
- Plan information cannot be collected for SQL statements that failed to be executed.
- Data in **PLAN\_TABLE** is in a session-level life cycle. Sessions are isolated from users and thereby users can view data of only the current session and current user.
- **PLAN\_TABLE** cannot be joined with GDS foreign tables.
- For a query that cannot be pushed down, object information cannot be collected and only such information as **REMOTE\_QUERY** and **CTE** can be collected. For details, see [Example 2](#).

### Example 1

You can perform the following steps to collect execution plans of SQL statements by running **EXPLAIN PLAN**:

- Step 1** Run the **EXPLAIN PLAN** statement.

#### NOTE

After the **EXPLAIN PLAN** statement is executed, plan information is automatically stored in **PLAN\_TABLE**. **INSERT**, **UPDATE**, and **ANALYZE** cannot be performed on **PLAN\_TABLE**.

For details about **PLAN\_TABLE**, see the **PLAN\_TABLE** system view.

```

explain plan set statement_id='TPCH-Q4' for
select
o_orderpriority,
count(*) as order_count
from
orders
where
o_orderdate >= '1993-07-01'::date
and o_orderdate < '1993-07-01'::date + interval '3 month'
and exists (
select
*
from
lineitem
where
l_orderkey = o_orderkey
and l_commitdate < l_receiptdate
)
group by
o_orderpriority
order by
o_orderpriority;

```

### Step 2 Query PLAN\_TABLE.

```
SELECT * FROM PLAN_TABLE;
```

| statement_id | plan_id  | id | operation           | options     | object_name | object_type | object_owner | projection                                    |
|--------------|----------|----|---------------------|-------------|-------------|-------------|--------------|-----------------------------------------------|
| TPCH-Q4      | 16781167 | 1  | ROW ADAPTER         |             |             |             |              | ORDERS.O_ORDERPRIORITY, (PG_CATALOG.COUNT(*)) |
| TPCH-Q4      | 16781167 | 2  | VECTOR SORT         |             |             |             |              | ORDERS.O_ORDERPRIORITY, (PG_CATALOG.COUNT(*)) |
| TPCH-Q4      | 16781167 | 3  | VECTOR AGGREGATE    | HASHED      |             |             |              | ORDERS.O_ORDERPRIORITY, PG_CATALOG.COUNT(*)   |
| TPCH-Q4      | 16781167 | 4  | VECTOR STREAMING    | GATHER      |             |             |              | ORDERS.O_ORDERPRIORITY, (COUNT(*))            |
| TPCH-Q4      | 16781167 | 5  | VECTOR AGGREGATE    | HASHED      |             |             |              | ORDERS.O_ORDERPRIORITY, COUNT(*)              |
| TPCH-Q4      | 16781167 | 6  | VECTOR NESTED LOOPS | SEMI        |             |             |              | ORDERS.O_ORDERPRIORITY                        |
| TPCH-Q4      | 16781167 | 7  | TABLE ACCESS        | CSTORE SCAN | ORDERS      | TABLE       | TPCH         | ORDERS.O_ORDERPRIORITY, ORDERS.O_ORDERKEY     |
| TPCH-Q4      | 16781167 | 8  | VECTOR MATERIALIZE  |             |             |             |              | LINEITEM.L_ORDERKEY                           |
| TPCH-Q4      | 16781167 | 9  | TABLE ACCESS        | CSTORE SCAN | LINEITEM    | TABLE       | TPCH         | LINEITEM.L_ORDERKEY                           |

### Step 3 Delete data from PLAN\_TABLE.

```
DELETE FROM PLAN_TABLE WHERE xxx;
```

----End

## Example 2

For a query that cannot be pushed down, only such information as **REMOTE\_QUERY** and **CTE** can be collected from **PLAN\_TABLE** after **EXPLAIN PLAN** is executed.

Scenario 1: The optimizer generates a plan for pushing down statements. In this case, only **REMOTE\_QUERY** can be collected.

```

explain plan set statement_id = 'test remote query' for
select
current_user
from
customer;

```

### Query PLAN\_TABLE.

```
SELECT * FROM PLAN_TABLE;
```

| statement_id      | plan_id  | id | operation      | options   | object_name      | object_type  | object_owner | projection    |
|-------------------|----------|----|----------------|-----------|------------------|--------------|--------------|---------------|
| test remote query | 29360133 | 1  | NESTED LOOPS   | CARTESIAN |                  |              |              | 'apple'::name |
| test remote query | 29360133 | 2  | DATA NODE SCAN |           | customer         | REMOTE_QUERY |              |               |
| test remote query | 29360133 | 3  | DATA NODE SCAN |           | customer_address | REMOTE_QUERY |              |               |

(3 rows)

Scenario 2: For a query with **WITH RECURSIVE** that cannot be pushed down, only **CTE** can be collected.

Disable **enable\_stream\_recursive** so that the query cannot be pushed down.  
set enable\_stream\_recursive = off;

Run the **EXPLAIN PLAN** SQL statement.

```
explain plan set statement_id = 'cte can not be push down'
for
with recursive rq as
(
  select id, name from t where id = 11
  union all
  select origin.id, rq.name || '>' || origin.name
  from rq join t origin on origin.pid = rq.id
)
select id, name from rq order by 1;
```

Query **PLAN\_TABLE**.

```
SELECT * FROM PLAN_TABLE;
```

| statement_id             | plan_id  | id | operation | options | object_name | object_type | object_owner | projection     |
|--------------------------|----------|----|-----------|---------|-------------|-------------|--------------|----------------|
| cte can not be push down | 25166071 | 1  | SORT      |         |             |             |              | rq.id, rq.name |
| cte can not be push down | 25166071 | 2  | CTE SCAN  |         | rq          | CTE         |              | rq.id, rq.name |

(2 rows)

## 13.7 LOCK

### Function

**LOCK TABLE** obtains a table-level lock.

GaussDB(DWS) always tries to select the lock mode with minimum constraints when automatically requesting a lock for a command referenced by a table. Use **LOCK** if users need a more strict lock mode. For example, suppose an application runs a transaction at the Read Committed isolation level and needs to ensure that data in a table remains stable in the duration of the transaction. To achieve this, you could obtain **SHARE** lock mode over the table before the query. This will prevent concurrent data changes and ensure subsequent reads of the table see a stable view of committed data. It is because the **SHARE** lock mode conflicts with the **ROW EXCLUSIVE** lock acquired by writers, and your **LOCK TABLE name IN SHARE MODE** statement will wait until any concurrent holders of **ROW EXCLUSIVE** mode locks commit or roll back. Therefore, once you obtain the lock, there are no uncommitted writes outstanding; furthermore none can begin until you release the lock.

### Precautions

- **LOCK TABLE** is useless outside a transaction block: the lock would remain held only to the completion of the statement. If **LOCK TABLE** is out of any transaction block, an error is reported.
- If no lock mode is specified, then **ACCESS EXCLUSIVE**, the most restrictive mode, is used.
- **LOCK TABLE ... IN ACCESS SHARE MODE** requires the **SELECT** permission on the target table. All other forms of **LOCK** require table-level **UPDATE** and/or the **DELETE** permission.
- There is no **UNLOCK TABLE** command. Locks are always released at transaction end.
- **LOCK TABLE** only deals with table-level locks, and so the mode names involving **ROW** are all misnomers. These mode names should generally be read as indicating the intention of the user to acquire row-level locks within the locked table. Also, **ROW EXCLUSIVE** mode is a shareable table lock. Keep



in mind that all the lock modes have identical semantics so far as **LOCK TABLE** is concerned, differing only in the rules about which modes conflict with which. For details about the rules, see [Table 13-1](#).

## Syntax

```
LOCK [ TABLE ] {[ ONLY ] name [, ...]} {name [ * ]} [, ...]
  [ IN {ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE | SHARE | SHARE
ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE} MODE ]
  [ NOWAIT ];
```

## Parameter Description

**Table 13-1** Lock mode conflicts

| Requested Lock Mode/<br>Current Lock Mode | ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE | SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE |
|-------------------------------------------|--------------|-----------|---------------|------------------------|-------|---------------------|-----------|------------------|
| ACCESS SHARE                              | -            | -         | -             | -                      | -     | -                   | -         | X                |
| ROW SHARE                                 | -            | -         | -             | -                      | -     | -                   | X         | X                |
| ROW EXCLUSIVE                             | -            | -         | -             | -                      | X     | X                   | X         | X                |
| SHARE UPDATE EXCLUSIVE                    | -            | -         | -             | X                      | X     | X                   | X         | X                |
| SHARE                                     | -            | -         | X             | X                      | -     | X                   | X         | X                |
| SHARE ROW EXCLUSIVE                       | -            | -         | X             | X                      | X     | X                   | X         | X                |
| EXCLUSIVE                                 | -            | X         | X             | X                      | X     | X                   | X         | X                |
| ACCESS EXCLUSIVE                          | X            | X         | X             | X                      | X     | X                   | X         | X                |

**LOCK** parameters are as follows:

- **name**  
The name (optionally schema-qualified) of an existing table to lock.  
The tables are locked one-by-one in the order specified in the **LOCK TABLE** command.  
Value range: An existing table name.
- **ONLY**  
**Only** locks only this table. If **Only** is not specified, this table and all its sub-tables are locked.
- **ACCESS SHARE**  
**ACCESS SHARE** allows only read operations on a table. In general, any SQL statements that only read a table and do not modify it will acquire this lock mode. The **SELECT** command acquires a lock of this mode on referenced tables.
- **ROW SHARE**  
**ROW SHARE** allows concurrent read of a table but does not allow any other operations on the table.  
**SELECT FOR UPDATE** and **SELECT FOR SHARE** automatically acquire the **ROW SHARE** lock on the target table and add the **ACCESS SHARE** lock to other referenced tables except **FOR SHARE** and **FOR UPDATE**.
- **ROW EXCLUSIVE**  
Like **ROW SHARE**, **ROW EXCLUSIVE** allows concurrent read of a table but does not allow modification of data in the table. **UPDATE**, **DELETE**, and **INSERT** automatically acquire the **ROW SHARE** lock on the target table and add the **ACCESS SHARE** lock to other referenced tables. Generally, all commands that modify table data acquire the **ROW EXCLUSIVE** lock for tables.
- **SHARE UPDATE EXCLUSIVE**  
This mode protects a table against concurrent schema changes and **VACUUM** runs.  
Acquired by **VACUUM** (without **FULL**), **ANALYZE**, **CREATE INDEX CONCURRENTLY**, and some forms of **ALTER TABLE**.
- **SHARE**  
**SHARE** allows concurrent queries of a table but does not allow modification of the table.  
Acquired by **CREATE INDEX** (without **CONCURRENTLY**).
- **SHARE ROW EXCLUSIVE**  
**SHARE ROW EXCLUSIVE** protects a table against concurrent data changes, and is self-exclusive so that only one session can hold it at a time.  
No SQL statements automatically acquire this lock mode.
- **EXCLUSIVE**  
**EXCLUSIVE** allows concurrent queries of the target table but does not allow any other operations.  
This mode allows only concurrent **ACCESS SHARE** locks; that is, only reads from the table can proceed in parallel with a transaction holding this lock mode.

No SQL statements automatically acquire this lock mode on user tables. However, it will be acquired on some system tables in case of some operations.

- **ACCESS EXCLUSIVE**

This mode guarantees that the holder is the only transaction accessing the table in any way.

Acquired by the **ALTER TABLE**, **DROP TABLE**, **TRUNCATE**, **REINDEX**, **CLUSTER**, and **VACUUM FULL** commands.

This is also the default lock mode for **LOCK TABLE** statements that do not specify a mode explicitly.

- **NOWAIT**

Specifies that **LOCK TABLE** should not wait for any conflicting locks to be released: if the specified lock(s) cannot be acquired immediately without waiting, the transaction is aborted.

If **NOWAIT** is not specified, **LOCK TABLE** obtains a table-level lock, waiting if necessary for any conflicting locks to be released.

## Examples

```
-- Obtain a SHARE lock on a primary key table when going to perform inserts into a foreign key table:
START TRANSACTION;

LOCK TABLE tpcds.reason IN SHARE MODE;

SELECT r_reason_desc FROM tpcds.reason WHERE r_reason_sk=5;
r_reason_desc
-----
Parts missing
(1 row)

COMMIT;

-- Obtain a SHARE ROW EXCLUSIVE lock on a primary key table when going to perform a delete operation:
CREATE TABLE tpcds.reason_t1 AS TABLE tpcds.reason;

START TRANSACTION;

LOCK TABLE tpcds.reason_t1 IN SHARE ROW EXCLUSIVE MODE;

DELETE FROM tpcds.reason_t1 WHERE r_reason_desc IN(SELECT r_reason_desc FROM tpcds.reason_t1
WHERE r_reason_sk < 6 );

DELETE FROM tpcds.reason_t1 WHERE r_reason_sk = 7;

COMMIT;

-- Delete the tpcds.reason_t1 table:
DROP TABLE tpcds.reason_t1;
```

## 13.8 MERGE INTO

### Function

The **MERGE INTO** statement is used to conditionally match data in a target table with that in a source table. If data matches, **UPDATE** is executed on the target table; if data does not match, **INSERT** is executed. You can use this syntax to run **UPDATE** and **INSERT** at a time for convenience.

## Precautions

- To run **MERGE INTO**, you must have the UPDATE and INSERT permissions for the target table, as well as the SELECT permission for the source table.
- **PREPARE** is not supported.
- **MERGE INTO** cannot be executed during redistribution.

## Syntax

```
MERGE INTO table_name [ [ AS ] alias ]
USING { { table_name | view_name } | subquery } [ [ AS ] alias ]
ON ( condition )
[
  WHEN MATCHED THEN
  UPDATE SET { column_name = { expression | DEFAULT } |
    ( column_name [, ...] ) = ( { expression | DEFAULT } [, ...] ) } [, ...]
  [ WHERE condition ]
]
[
  WHEN NOT MATCHED THEN
  INSERT { DEFAULT VALUES |
    [ ( column_name [, ...] ) ] VALUES ( { expression | DEFAULT } [, ...] ) [, ...] [ WHERE condition ] }
];
```

## Parameter Description

- **INTO** clause  
Specifies the target table that is being updated or has data being inserted. It cannot be a replication table.
  - **table\_name**  
Specifies the name of the target table.
  - **alias**  
Specifies the alias of the target table.  
Value range: a string that complies with the naming convention.
- **USING** clause  
Specifies the source table, which can be a table, view, or subquery.
- **ON** clause  
Specifies the condition used to match data between the source and target tables. Columns in the condition cannot be updated.
- **WHEN MATCHED** clause  
Performs the UPDATE operation if data in the source table matches that in the target table based on the condition.  
Distribution keys cannot be updated. System catalogs and system columns cannot be updated.
- **WHEN NOT MATCHED** clause  
Specifies that the INSERT operation is performed if data in the source table does not match that in the target table based on the condition.  
The **INSERT** clause is not allowed to contain multiple **VALUES**.  
The order of **WHEN MATCHED** and **WHEN NOT MATCHED** clauses can be reversed. One of them can be used by default, but they cannot be both used at one time. Two **WHEN MATCHED** or **WHEN NOT MATCHED** clauses cannot be specified at the same time.

- **DEFAULT**  
Specifies the default value of a column.  
It will be **NULL** if no specific default value has been assigned to it.
- **WHERE condition**  
Specifies the conditions for the **UPDATE** and **INSERT** clauses. The two clauses will be executed only when the conditions are met. The default value can be used. System columns cannot be referenced in **WHERE condition**.

## Examples

```
-- Create the target table products and source table newproducts, and insert data to them.
CREATE TABLE products
(
  product_id INTEGER,
  product_name VARCHAR2(60),
  category VARCHAR2(60)
);

INSERT INTO products VALUES (1501, 'vivitar 35mm', 'electrnrcs');
INSERT INTO products VALUES (1502, 'olympus is50', 'electrnrcs');
INSERT INTO products VALUES (1600, 'play gym', 'toys');
INSERT INTO products VALUES (1601, 'lamaze', 'toys');
INSERT INTO products VALUES (1666, 'harry potter', 'dvd');

CREATE TABLE newproducts
(
  product_id INTEGER,
  product_name VARCHAR2(60),
  category VARCHAR2(60)
);

INSERT INTO newproducts VALUES (1502, 'olympus camera', 'electrnrcs');
INSERT INTO newproducts VALUES (1601, 'lamaze', 'toys');
INSERT INTO newproducts VALUES (1666, 'harry potter', 'toys');
INSERT INTO newproducts VALUES (1700, 'wait interface', 'books');

-- Run MERGE INTO.
MERGE INTO products p
USING newproducts np
ON (p.product_id = np.product_id)
WHEN MATCHED THEN
  UPDATE SET p.product_name = np.product_name, p.category = np.category WHERE p.product_name !=
'play gym'
WHEN NOT MATCHED THEN
  INSERT VALUES (np.product_id, np.product_name, np.category) WHERE np.category = 'books';
MERGE 4

-- Query updates.
SELECT * FROM products ORDER BY product_id;
product_id | product_name | category
-----+-----+-----
      1501 | vivitar 35mm | electrncs
      1502 | olympus camera | electrncs
      1600 | play gym | toys
      1601 | lamaze | toys
      1666 | harry potter | toys
      1700 | wait interface | books
(6 rows)

-- Delete the tables.
DROP TABLE products;
DROP TABLE newproducts;
```

## 13.9 INSERT

### Function

INSERT inserts new rows into a table.

### Precautions

- You must have the **INSERT** permission on a table in order to insert into it.
- Use of the **RETURNING** clause requires the **SELECT** permission on all columns mentioned in **RETURNING**.
- If you use the **query** clause to insert rows from a query, you of course need to have the **SELECT** permission on any table or column used in the query.
- When you connect to a database compatible to Teradata and **td\_compatible\_truncation** is **on**, a long character string will be automatically truncated. If later **INSERT** statements (not involving foreign tables) insert long strings to columns of char- and varchar-typed columns in the target table, the system will truncate the long strings to ensure no strings exceed the maximum length defined in the target table.

#### NOTE

If inserting multi-byte character data (such as Chinese characters) to database with the character set byte encoding (SQL\_ASCII, LATIN1), and the character data crosses the truncation position, the string is truncated based on its bytes instead of characters. Unexpected result will occur in tail after the truncation. If you want correct truncation result, you are advised to adopt encoding set such as UTF8, which has no character data crossing the truncation position.

### Syntax

```
[ WITH [ RECURSIVE ] with_query [, ...] ]  
INSERT INTO table_name [ ( column_name [, ...] ) ]  
  { DEFAULT VALUES  
  | VALUES (( { expression | DEFAULT } [, ...] ) )[, ...]  
  | query }  
  [ RETURNING { * | {output_expression [ [ AS ] output_name ] }[, ...] }];
```

### Parameter Description

- **WITH [ RECURSIVE ] with\_query [, ...]**

The **WITH** clause allows you to specify one or more subqueries that can be referenced by name in the primary query, equal to temporary table.

If **RECURSIVE** is specified, it allows a **SELECT** subquery to reference itself by name.

The detailed format of **with\_query** is as follows: with\_query\_name  
[ (column\_name [,...]) ] AS

( {select | values | insert | update | delete} )

-- **with\_query\_name** specifies the name of the result set generated by a subquery. Such names can be used to access the result sets of subqueries in a query.

**column\_name** specifies the column name displayed in the subquery result set.

Each subquery can be a **SELECT**, **VALUES**, **INSERT**, **UPDATE** or **DELETE** statement.

- **table\_name**

Specifies the name of the target table.

Value range: An existing table name.

- **column\_name**

The name of a column in a table.

- The column name can be qualified with a subfield name or array subscript, if needed.
- Each column not present in the explicit or implicit column list will be filled with a default value, either its declared default value or **NULL** if there is none. (Inserting into only some fields of a composite column leaves the other fields **NULL**.)
- The target column names **column\_name** can be listed in any order. If no list of column names is given at all, the default is all the columns of the table in their declared order.
- The target columns are the first N column names, if there are only N columns supplied by the value clause or query.
- The values supplied by the **value** clause or **query** are associated with the explicit or implicit column list left-to-right.

Value range: an existing column name

- **expression**

Specifies an expression or a value to assign to the corresponding column.

- If single-quotation marks are inserted in a column, the single-quotation marks need to be used for escape.
- If the expression for any column is not of the correct data type, automatic type conversion will be attempted. If the attempt fails, data insertion fails and the system returns an error message.

Example:

```
postgres=# create table tt01 (id int,content varchar(50));
NOTICE: The 'DISTRIBUTE BY' clause is not specified. Using 'id' as the distribution column by default.
HINT: Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
postgres=# insert into tt01 values (1,'Jack say "hello"');
INSERT 0 1
postgres=# insert into tt01 values (2,'Rose do 50%');
INSERT 0 1
postgres=# insert into tt01 values (3,'Lilei say "world"');
INSERT 0 1
postgres=# insert into tt01 values (4,'Hanmei do 100%');
INSERT 0 1
postgres=# select * from tt01;
 id | content
-----+-----
  3 | Lilei say 'world'
  4 | Hanmei do 100%
  1 | Jack say 'hello'
  2 | Rose do 50%
(4 rows)
postgres=# drop table tt01;
DROP TABLE
```

- **DEFAULT**  
All columns will be filled with their default values. The value is **NULL** if no specified default value has been assigned to it.
- **query**  
A query (**SELECT** statement) that supplies the rows to be inserted.
- **RETURNING**  
Returns the inserted rows. The syntax of the **RETURNING** list is identical to that of the output list of **SELECT**.
- **output\_expression**  
An expression used to calculate the output of the **INSERT** command after each row is inserted.  
Value range: The expression can use any field in the table. Write \* to return all columns of the inserted row(s).
- **output\_name**  
A name to use for a returned column.  
Value range: A string. It must comply with the naming convention rule.

## Examples

```
-- Create the tpcds.reason_t2 table:
CREATE TABLE tpcds.reason_t2
(
  r_reason_sk integer,
  r_reason_id character(16),
  r_reason_desc character(100)
);

-- Insert a record into a table:
INSERT INTO tpcds.reason_t2(r_reason_sk, r_reason_id, r_reason_desc) VALUES (1, 'AAAAAAAAABAAAAAAAA', 'reason1');

-- Insert a record into a table. This command is equivalent to the last one.
INSERT INTO tpcds.reason_t2 VALUES (2, 'AAAAAAAAABAAAAAAAA', 'reason2');

-- Insert records into the table:
INSERT INTO tpcds.reason_t2 VALUES (3, 'AAAAAAAACAAAAAAA', 'reason3'),(4, 'AAAAAAAADAAAAAAA', 'reason4'),(5, 'AAAAAAAEEAAAAAAA', 'reason5');

-- Insert records whose r_reason_sk in the tpcds.reason table is less than 5:
INSERT INTO tpcds.reason_t2 SELECT * FROM tpcds.reason WHERE r_reason_sk <5;

-- Delete the tpcds.reason_t2 table:
DROP TABLE tpcds.reason_t2;
```

## 13.10 SELECT

### Function

**SELECT** retrieves data from a table or view.

Serving as an overlaid filter for a database table, **SELECT** using SQL keywords retrieves required data from data tables.



## Precautions

- Using **SELECT** can join HDFS and ordinary tables, but cannot join ordinary and GDS foreign tables. That is, A **SELECT** statement cannot contain both ordinary and GDS foreign tables.
- The user must have the SELECT permission on every column used in the **SELECT** command.
- UPDATE permission is required when using **FOR UPDATE** or **FOR SHARE**.

## Syntax

- Querying data

```
[ WITH [ RECURSIVE ] with_query [, ... ] ]
SELECT [ /*+ plan_hint */ ] [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
{ * | {expression [ [ AS ] output_name ]} [, ...] }
[ FROM from_item [, ...] ]
[ WHERE condition ]
[ GROUP BY grouping_element [, ...] ]
[ HAVING condition [, ...] ]
[ WINDOW {window_name AS ( window_definition )} [, ...] ]
[ { UNION | INTERSECT | EXCEPT | MINUS } [ ALL | DISTINCT ] select ]
[ ORDER BY {expression [ [ ASC | DESC | USING operator ] | nlssort_expression_clause } [ NULLS { FIRST | LAST } ]} [, ...] ]
[ LIMIT { count | ALL } ]
[ OFFSET start [ ROW | ROWS ] ]
[ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ]
[ {FOR { UPDATE | SHARE } [ OF table_name [, ...] ] [ NOWAIT ]} [...] ];
```

### NOTE

In condition and expression, you can use the aliases of expressions in **targetlist** in compliance with the following rules:

- Reference only in the same level.
- Only reference aliases in **targetlist**.
- Reference a prior expression in a subsequent expression.
- The **volatile** function cannot be used.
- The **Window** function cannot be used.
- Do not reference an alias in the **join on** condition.
- An error is reported if **targetlist** contains multiple referenced aliases.
- The subquery **with\_query** is as follows:  
with\_query\_name [ ( column\_name [, ...] ) ]  
AS ( {select | values | insert | update | delete} )
- The specified query source **from\_item** is as follows:  
{ [ ONLY ] table\_name [ \* ] [ partition\_clause ] [ [ AS ] alias [ ( column\_alias [, ...] ) ] ]  
| ( select ) [ AS ] alias [ ( column\_alias [, ...] ) ]  
| {with\_query\_name [ [ AS ] alias [ ( column\_alias [, ...] ) ] ] }  
| function\_name ( [ argument [, ...] ] ) [ AS ] alias [ ( column\_alias [, ...] | column\_definition [, ...] ) ]  
| function\_name ( [ argument [, ...] ] ) AS ( column\_definition [, ...] )  
| from\_item [ NATURAL ] join\_type from\_item [ ON join\_condition | USING ( join\_column [, ...] ) ] }
- The **group** clause is as follows:  
( )  
| expression  
| ( expression [, ...] )  
| ROLLUP ( { expression | ( expression [, ...] ) } [, ...] )  
| CUBE ( { expression | ( expression [, ...] ) } [, ...] )  
| GROUPING SETS ( grouping\_element [, ...] )
- The specified partition **partition\_clause** is as follows:  
PARTITION { ( partition\_name ) |  
FOR ( partition\_value [, ...] ) }

 **NOTE**

You can specify partitions only for ordinary tables.

- The sorting order **nlssort\_expression\_clause** is as follows:  
NLSSORT ( column\_name, ' NLS\_SORT = { SCHINESE\_PINYIN\_M | generic\_m\_ci } ' )
- Simplified query syntax, equivalent to **select \* from table\_name**.  
TABLE { ONLY {(table\_name)| table\_name} | table\_name [ \* ]};

## Parameter Description

- **WITH [ RECURSIVE ] with\_query [, ...]**  
The **WITH** clause allows you to specify one or more subqueries that can be referenced by name in the primary query, equal to temporary table.  
If **RECURSIVE** is specified, it allows a **SELECT** subquery to reference itself by name.  
The detailed format of **with\_query** is as follows: **with\_query\_name** [ ( **column\_name** [, ...] ) ] **AS** ( {**select** | **values** | **insert** | **update** | **delete** } )
  - **with\_query\_name** specifies the name of the result set generated by a subquery. Such names can be used to access the result sets of subqueries in a query.
  - **column\_name** specifies a column name displayed in the subquery result set.
  - Each subquery can be a **SELECT**, **VALUES**, **INSERT**, **UPDATE** or **DELETE** statement.
- **plan\_hint** clause  
Follows the **SELECT** keyword in the */\*+<Plan hint>\*/* format. It is used to optimize the plan of a **SELECT** statement block. For details, see section "Hint-based Tuning."
- **ALL**  
Specifies that all rows meeting the requirements are returned. This is the default behavior, so you can omit this keyword.
- **DISTINCT [ ON ( expression [, ...] ) ]**  
Removes all duplicate rows from the **SELECT** result so one row is kept from each group of duplicates.  
**ON ( expression [, ...] )** is only reserved for the first row among all the rows with the same result calculated using given expressions.

---

**NOTICE**

**DISTINCT ON** expression is explained with the same rule of **ORDER BY**. Unless you use **ORDER BY** to guarantee that the required row appears first, you cannot know what the first row is.

- 
- **SELECT list**  
Indicates columns to be queried. Some or all columns (using wildcard character \*) can be queried.  
You may use the **AS output\_name** clause to give an alias for an output column. The alias is used for the displaying of the output column.

Column names may be either of:

- Manually input column names which are spaced using commas (,).
- Fields computed in the **FROM** clause.

- **FROM** clause

Indicates one or more source tables for **SELECT**.

The **FROM** clause can contain the following elements:

- table\_name

Indicates the name (optionally schema-qualified) of an existing table or view, for example, **schema\_name.table\_name**.

- alias

Gives a temporary alias to a table to facilitate the quotation by other queries.

An alias is used for brevity or to eliminate ambiguity for self-joins. When an alias is provided, it completely hides the actual name of the table or function.

- column\_alias

Specifies the column alias.

- PARTITION

Queries data in the specified partition in a partition table.

- partition\_name

Specifies the name of a partition.

- partition\_value

Specifies the value of the specified partition key. If there are many partition keys, use the **PARTITION FOR** clause to specify the value of the only partition key you want to use.

- subquery

Performs a subquery in the **FROM** clause. A temporary table is created to save subquery results.

- with\_query\_name

**WITH** clause can also be the source of **FROM** clause and can be referenced with the name queried by executing **WITH**.

- function\_name

Function name. Function calls can appear in the **FROM** clause.

- join\_type

There are five types below:

- [ INNER ] JOIN

A **JOIN** clause combines two **FROM** items. Use parentheses if necessary to determine the order of nesting. In the absence of parentheses, **JOIN** nests left-to-right.

In any case, **JOIN** binds more tightly than the commas separating **FROM** items.

- LEFT [ OUTER ] JOIN

Returns all rows in the qualified Cartesian product (all combined rows that pass its join condition), and pluses one copy of each row in

the left-hand table for which there was no right-hand row that passed the join condition. This left-hand row is extended to the full width of the joined table by inserting **NULL** values for the right-hand columns. Note that only the **JOIN** clause's own condition is considered while deciding which rows have matches. Outer conditions are applied afterwards.

- **RIGHT [ OUTER ] JOIN**

Returns all the joined rows, plus one row for each unmatched right-hand row (extended with **NULL** on the left).

This is just a notational convenience, since you could convert it to a **LEFT OUTER JOIN** by switching the left and right inputs.

- **FULL [ OUTER ] JOIN**

Returns all the joined rows, plus one row for each unmatched left-hand row (extended with **NULL** on the right), and plus one row for each unmatched right-hand row (extended with **NULL** on the left).

- **CROSS JOIN**

**CROSS JOIN** is equivalent to **INNER JOIN ON (TRUE)**, which means no rows are removed by qualification. These join types are just a notational convenience, since they do nothing you could not do with plain **FROM** and **WHERE**.

 **NOTE**

For the **INNER** and **OUTER** join types, a join condition must be specified, namely exactly one of **NATURAL ON**, **join\_condition**, or **USING (join\_column [, ...])**. For **CROSS JOIN**, none of these clauses can appear.

**CROSS JOIN** and **INNER JOIN** produce a simple Cartesian product, the same result as you get from listing the two items at the top level of **FROM**.

- **ON join\_condition**

A join condition to define which rows have matches in joins. Example: **ON left\_table.a = right\_table.a**

- **USING(join\_column[, ...])**

**ON left\_table.a = right\_table.a AND left\_table.b = right\_table.b ...** abbreviation. Corresponding columns must have the same name.

- **NATURAL**

**NATURAL** is a shorthand for a **USING** list that mentions all columns in the two tables that have the same names.

- **from item**

Specifies the name of the query source object connected.

- **WHERE clause**

The **WHERE** clause forms an expression for row selection to narrow down the query range of **SELECT**. The condition is any expression that evaluates to a result of Boolean type. Rows that do not satisfy this condition will be eliminated from the output.

In the **WHERE** clause, you can use the operator (+) to convert a table join to an outer join. However, this method is not recommended because it is not the

standard SQL syntax and may raise syntax compatibility issues during platform migration. There are many restrictions on using the operator (+):

- a. It can appear only in the **WHERE** clause.
- b. If a table join has been specified in the **FROM** clause, the operator (+) cannot be used in the **WHERE** clause.
- c. The operator (+) can work only on columns of tables or views, instead of on expressions.
- d. If table A and table B have multiple join conditions, the operator (+) must be specified in all the conditions. Otherwise, the operator (+) will not take effect, and the table join will be converted into an inner join without any prompt information.
- e. Tables specified in a join condition where the operator (+) works cannot cross queries or subqueries. If tables where the operator (+) works are not in the **FROM** clause of the current query or subquery, an error will be reported. If a peer table for the operator (+) does not exist, no error will be reported and the table join will be converted into an inner join.
- f. Expressions where the operator (+) is used cannot be directly connected through **OR**.
- g. If a column where the operator (+) works is compared with a constant, the expression becomes a part of the join condition.
- h. A table cannot have multiple foreign tables.
- i. The operator (+) can appear only in the following expressions: comparison, NOT, ANY, ALL, IN, NULLIF, IS DISTINCT FROM, and IS OF expressions. It is not allowed in other types of expressions. In addition, these expressions cannot be connected through **AND** or **OR**.
- j. The operator (+) can be used to convert a table join only to a left or right outer join, instead of a full join. That is, the operator (+) cannot be specified on both tables of an expression.

---

**NOTICE**

For the **WHERE** clause, if a special character % \_ or \ is queried in **LIKE**, add the slash (\) before each character.

---

Example:

```
postgres=# create table tt01 (id int,content varchar(50));
NOTICE: The 'DISTRIBUTE BY' clause is not specified. Using 'id' as the distribution column by default.
HINT: Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
postgres=# insert into tt01 values (1,'Jack say "hello"');
INSERT 0 1
postgres=# insert into tt01 values (2,'Rose do 50%');
INSERT 0 1
postgres=# insert into tt01 values (3,'Lilei say "world"');
INSERT 0 1
postgres=# insert into tt01 values (4,'Hanmei do 100%');
INSERT 0 1
postgres=# select * from tt01;
 id | content
----+-----
  3 | Lilei say 'world'
  4 | Hanmei do 100%
```

```
1 | Jack say 'hello'
2 | Rose do 50%
(4 rows)

postgres=# select * from tt01 where content like '%"he%';
id | content
-----+-----
1 | Jack say 'hello'
(1 row)

postgres=# select * from tt01 where content like '%50\%';
id | content
-----+-----
2 | Rose do 50%
(1 row)

postgres=# drop table tt01;
DROP TABLE
```

- **GROUP BY clause**

Condenses query results into a single row or selected rows that share the same values for the grouped expressions.

- CUBE ( { expression | ( expression [, ...] ) } [, ...] )

A CUBE grouping is an extension to the GROUP BY clause that creates subtotals for all of the possible combinations of the given list of grouping columns (or expressions). In terms of multidimensional analysis, CUBE generates all the subtotals that could be calculated for a data cube with the specified dimensions. For example, given three expressions ( $n=3$ ) in the CUBE clause, the operation results in  $2^n = 2^3 = 8$  groupings. Rows grouped on the values of  $n$  expressions are called regular rows, and the rest are called superaggregate rows.

- GROUPING SETS ( grouping\_element [, ...] )

**GROUPING SETS** is another extension to the **GROUP BY** clause. It allows users to specify multiple **GROUP BY** clauses. This improves efficiency by trimming away unnecessary data. After you specify the set of groups that you want to create using a GROUPING SETS expression within a GROUP BY clause, the database does not need to compute a whole ROLLUP or CUBE.

---

**NOTICE**

If the **SELECT** list expression quotes some ungrouped fields and no aggregate function is used, an error is displayed. This is because multiple values may be returned for ungrouped fields.

- **HAVING clause**

Selects special groups by working with the **GROUP BY** clause. The **HAVING** clause compares some attributes of groups with a constant. Only groups that matching the logical expression in the **HAVING** clause are extracted.

- **WINDOW clause**

The general format is **WINDOW window\_name AS ( window\_definition )** [, ...]. **window\_name** is a name can be referenced by **window\_definition**. **window\_definition** can be expressed in the following forms:

[ existing\_window\_name ]

[ PARTITION BY expression [, ...] ]  
[ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ] [, ...] ]  
[ frame\_clause ]

**frame\_clause** defines a **window frame** for the window function. The window function (not all window functions) depends on **window frame** and **window frame** is a set of relevant rows of the current query row. **frame\_clause** can be expressed in the following forms:

[ RANGE | ROWS ] frame\_start  
[ RANGE | ROWS ] BETWEEN frame\_start AND frame\_end

**frame\_start** and **frame\_end** can be expressed in the following forms:

UNBOUNDED PRECEDING  
value PRECEDING (not supported for **RANGE**)  
CURRENT ROW  
value FOLLOWING (not supported for **RANGE**)  
UNBOUNDED FOLLOWING

---

#### NOTICE

For the query of column storage table, only **row\_number** window function is supported, **frame\_clause** is not supported.

---

- **UNION clause**

Computes the set union of the rows returned by the involved **SELECT** statements.

The **UNION** clause has the following constraints:

- By default, the result of **UNION** does not contain any duplicate rows unless the **ALL** option is specified.
- Multiple **UNION** operators in the same **SELECT** statement are evaluated left to right, unless otherwise specified by parentheses.
- **FOR UPDATE** cannot be specified either for a **UNION** result or for any input of a **UNION**.

General expression:

select\_statement UNION [ALL] select\_statement

- **select\_statement** can be any **SELECT** statement without an **ORDER BY**, **LIMIT**, **FOR UPDATE**, or **FOR SHARE** statement.
- **ORDER BY** and **LIMIT** in parentheses can be attached in a sub-expression.

- **INTERSECT clause**

Computes the set intersection of rows returned by the involved **SELECT** statements. The result of **INTERSECT** does not contain any duplicate rows.

The **INTERSECT** clause has the following constraints:

- Multiple **INTERSECT** operators in the same **SELECT** statement are evaluated left to right, unless otherwise specified by parentheses.

- Processing **INTERSECT** preferentially when **UNION** and **INTERSECT** operations are executed for results of multiple **SELECT** statements.

General format:

```
select_statement INTERSECT select_statement
```

**select\_statement** can be any **SELECT** statement without a **FOR UPDATE** clause.

- **EXCEPT clause**

**EXCEPT** clause has the following common form:

```
select_statement EXCEPT [ ALL ] select_statement
```

**select\_statement** can be any **SELECT** statement without a **FOR UPDATE** clause.

The **EXCEPT** operator computes the set of rows that are in the result of the left **SELECT** statement but not in the result of the right one.

The result of **EXCEPT** does not contain any duplicate rows unless the **ALL** option is specified. To execute **ALL**, a row that has  $m$  duplicates in the left table and  $n$  duplicates in the right table will appear  $\text{MAX}(m-n, 0)$  times in the result set.

Multiple **EXCEPT** operators in the same **SELECT** statement are evaluated left to right, unless parentheses dictate otherwise. **EXCEPT** binds at the same level as **UNION**.

Currently, **FOR UPDATE** and **FOR SHARE** cannot be specified either for an **EXCEPT** result or for any input of an **EXCEPT**.

- **MINUS clause**

Has the same function and syntax as **EXCEPT** clause.

- **ORDER BY clause**

Sorts data retrieved by **SELECT** in descending or ascending order. If the **ORDER BY** expression contains multiple columns:

- If two columns are equal according to the leftmost expression, they are compared according to the next expression and so on.
- If they are equal according to all specified expressions, they are returned in an implementation-dependent order.
- Columns sorted by **ORDER BY** must be contained in the result retrieved by **SELECT**.

---

**NOTICE**

To support Chinese pinyin order or case insensitive order, specify the **UTF-8** or **GBK** encoding mode during database initiation. The commands are as follows:  
initdb - E UTF8 - D../data - locale=zh\_CN.UTF-8 or initdb - E GBK-D../data - locale=zh\_CN.GBK.

- **LIMIT clause**

Consists of two independent sub-clauses:

```
LIMIT { count | ALL }
```

**OFFSET start count** specifies the maximum number of rows to return, while **start** specifies the number of rows to skip before starting to return rows.



When both are specified, **start** rows are skipped before starting to count the **count** rows to be returned.

- **OFFSET** clause

The SQL: 2008 standard has introduced a different clause:

OFFSET start { ROW | ROWS }

**start** specifies the number of rows to skip before starting to return rows.

- **FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY**

If **count** is omitted in a **FETCH** clause, it defaults to 1.

- **FOR UPDATE** clause

Locks rows retrieved by **SELECT**. This ensures that the rows cannot be modified or deleted by other transactions until the current transaction ends. That is, other transactions that attempt **UPDATE**, **DELETE**, or **SELECT FOR UPDATE** of these rows will be blocked until the current transaction ends.

To avoid waiting for the committing of other transactions, you can apply **NOWAIT**. Rows to which **NOWAIT** applies cannot be immediately locked. After **SELECT FOR UPDATE NOWAIT** is executed, an error is reported.

**FOR SHARE** behaves similarly, except that it acquires a shared rather than exclusive lock on each retrieved row. A share lock blocks other transaction from performing **UPDATE**, **DELETE**, or **SELECT FOR UPDATE** on these rows, but it does not prevent them from performing **SELECT FOR SHARE**.

If specified tables are named in **FOR UPDATE** or **FOR SHARE**, then only rows coming from those tables are locked; any other tables used in **SELECT** are simply read as usual. Otherwise, locking all tables in the command.

If **FOR UPDATE** or **FOR SHARE** is applied to a view or sub-query, it affects all tables used in the view or sub-query.

Multiple **FOR UPDATE** and **FOR SHARE** clauses can be written if it is necessary to specify different locking behaviors for different tables.

If the same table is mentioned (or implicitly affected) by both **FOR UPDATE** and **FOR SHARE** clauses, it is processed as **FOR UPDATE**. Similarly, a table is processed as **NOWAIT** if that is specified in any of the clauses affecting it.

---

**NOTICE**

- For SQL statements containing **FOR UPDATE** or **FOR SHARE**, their execution plans will be pushed down to DNs. If the pushdown fails, an error will be reported.
- The query of column storage table does not support **for update/share**.

- **NLS\_SORT**

Indicates a field to be ordered in a special mode. Currently, only the Chinese Pinyin order and case insensitive order are supported.

Valid value:

- **SCHINESE\_PINYIN\_M**, sorted by Pinyin order. To use this sort method, specify **GBK** as the encoding format when you create the database. If you do not do so, this value is invalid.
- **generic\_m\_ci**, case-insensitive order.

- **PARTITION clause**

Queries data in the specified partition in a partition table.

## Examples

```
-- Obtain the temp_t temporary table by a subquery and query all records in this table:
WITH temp_t(name,isdba) AS (SELECT username,usesuper FROM pg_user) SELECT * FROM temp_t;

-- Query all the r_reason_sk records in the tpcds.reason table and de-duplicate them.
SELECT DISTINCT(r_reason_sk) FROM tpcds.reason;

-- Example of a LIMIT clause: Obtain a record from the table.
SELECT * FROM tpcds.reason LIMIT 1;

-- Query all records and sort them in alphabetic order.
SELECT r_reason_desc FROM tpcds.reason ORDER BY r_reason_desc;

-- Use table aliases to obtain data from the pg_user and pg_user_status tables:
SELECT a.username,b.locktime FROM pg_user a,pg_user_status b WHERE a.usesysid=b.roloid;

-- Example of the FULL JOIN clause: Join data in the pg_user and pg_user_status tables.
SELECT a.username,b.locktime,a.usesuper FROM pg_user a FULL JOIN pg_user_status b on
a.usesysid=b.roloid;

-- Example of the GROUP BY clause: Filter data based on query conditions, and group the results.
SELECT r_reason_id, AVG(r_reason_sk) FROM tpcds.reason GROUP BY r_reason_id HAVING
AVG(r_reason_sk) > 25;

-- Example of the GROUP BY CUBE clause: Filter data based on query conditions, and group the results.
SELECT r_reason_id,AVG(r_reason_sk) FROM tpcds.reason GROUP BY CUBE(r_reason_id,r_reason_sk);

-- Example of the GROUP BY GROUPING SETS clause: Filter data based on query conditions, and group the
results.
SELECT r_reason_id,AVG(r_reason_sk) FROM tpcds.reason GROUP BY GROUPING
SETS((r_reason_id,r_reason_sk),r_reason_sk);

-- Example of the UNION clause: Merge the names started with W and N in the r_reason_desc column in
the tpcds.reason table.
SELECT r_reason_sk, tpcds.reason.r_reason_desc
FROM tpcds.reason
WHERE tpcds.reason.r_reason_desc LIKE 'W%'
UNION
SELECT r_reason_sk, tpcds.reason.r_reason_desc
FROM tpcds.reason
WHERE tpcds.reason.r_reason_desc LIKE 'N%';

-- Example of the NLS_SORT clause: Sort by Chinese Pinyin.
SELECT * FROM tpcds.reason ORDER BY NLSSORT( r_reason_desc, 'NLS_SORT = SCHINESE_PINYIN_M');

-- Case-insensitive order:
SELECT * FROM tpcds.reason ORDER BY NLSSORT( r_reason_desc, 'NLS_SORT = generic_m_ci');

-- Create the table tpcds.reason_p:
CREATE TABLE tpcds.reason_p
(
  r_reason_sk integer,
  r_reason_id character(16),
  r_reason_desc character(100)
)
PARTITION BY RANGE (r_reason_sk)
(
  partition P_05_BEFORE values less than (05),
  partition P_15 values less than (15),
  partition P_25 values less than (25),
  partition P_35 values less than (35),
  partition P_45_AFTER values less than (MAXVALUE)
)
```

```

;

-- Insert data:
INSERT INTO tpcds.reason_p values(3,'AAAAAAAABAAAAAAA','reason 1'),
(10,'AAAAAAAABAAAAAAA','reason 2'),(4,'AAAAAAAABAAAAAAA','reason 3'),
(10,'AAAAAAAABAAAAAAA','reason 4'),(10,'AAAAAAAABAAAAAAA','reason 5'),
(20,'AAAAAAAACAAAAAAA','reason 6'),(30,'AAAAAAAACAAAAAAA','reason 7');

-- Example of the PARTITION clause: Obtain data from the P_05_BEFORE partition in the tpcds.reason_p
table.
SELECT * FROM tpcds.reason_p PARTITION (P_05_BEFORE);
r_reason_sk | r_reason_id | r_reason_desc
-----+-----+-----
          4 | AAAAAAABAAAAAAA | reason 3
          3 | AAAAAAABAAAAAAA | reason 1
(2 rows)

-- Example of the GROUP BY clause: Group records in the tpcds.reason_p table by r_reason_id, and count
the number of records in each group.
SELECT COUNT(*),r_reason_id FROM tpcds.reason_p GROUP BY r_reason_id;
count | r_reason_id
-----+-----
          2 | AAAAAAACAAAAAAA
          5 | AAAAAAABAAAAAAA
(2 rows)

-- Example of the GROUP BY CUBE clause: Filter data based on query conditions, and group the results.
SELECT * FROM tpcds.reason GROUP BY CUBE (r_reason_id,r_reason_sk,r_reason_desc);

-- Example of the GROUP BY GROUPING SETS clause: Filter data based on query conditions, and group the
results.
SELECT * FROM tpcds.reason GROUP BY GROUPING SETS ((r_reason_id,r_reason_sk),r_reason_desc);

-- Example of the HAVING clause: Group records in the tpcds.reason_p table by r_reason_id, count the
number of records in each group, and display only values whose number of r_reason_id is greater than 2.
SELECT COUNT(*) c,r_reason_id FROM tpcds.reason_p GROUP BY r_reason_id HAVING c>2;
c | r_reason_id
-----+-----
          5 | AAAAAAABAAAAAAA
(1 row)

-- Example of the IN clause: Group records in the tpcds.reason_p table by r_reason_id, count the number
of records in each group, and display only the numbers of records whose r_reason_id is
AAAAAABAAAAAAA or AAAAAADAAAAAAA.
SELECT COUNT(*),r_reason_id FROM tpcds.reason_p GROUP BY r_reason_id HAVING r_reason_id
IN('AAAAAABAAAAAAA','AAAAAADAAAAAAA');
count | r_reason_id
-----+-----
          5 | AAAAAAABAAAAAAA
(1 row)

-- Example of the INTERSECT clause: Query records whose r_reason_id is AAAAAABAAAAAAA and
whose r_reason_sk is smaller than 5.
SELECT * FROM tpcds.reason_p WHERE r_reason_id='AAAAAABAAAAAAA' INTERSECT SELECT * FROM
tpcds.reason_p WHERE r_reason_sk<5;
r_reason_sk | r_reason_id | r_reason_desc
-----+-----+-----
          4 | AAAAAAABAAAAAAA | reason 3
          3 | AAAAAAABAAAAAAA | reason 1
(2 rows)

-- Example of the EXCEPT clause: Query records whose r_reason_id is AAAAAABAAAAAAA and whose
r_reason_sk is greater than or equal to 4.
SELECT * FROM tpcds.reason_p WHERE r_reason_id='AAAAAABAAAAAAA' EXCEPT SELECT * FROM
tpcds.reason_p WHERE r_reason_sk<4;
r_reason_sk | r_reason_id | r_reason_desc
-----+-----+-----
          10 | AAAAAAABAAAAAAA | reason 2
          10 | AAAAAAABAAAAAAA | reason 5

```

```

10 | AAAAAAAAAABAAAAAAA | reason 4
4 | AAAAAAAAAABAAAAAAA | reason 3
(4 rows)

-- Specify the operator (+) in the WHERE clause to indicate a left join.
select t1.sr_item_sk ,t2.c_customer_id from store_returns t1, customer t2 where t1.sr_customer_sk =
t2.c_customer_sk(+)
order by 1 desc limit 1;
sr_item_sk | c_customer_id
-----+-----
18000 |
(1 row)

-- Specify the operator (+) in the WHERE clause to indicate a right join.
select t1.sr_item_sk ,t2.c_customer_id from store_returns t1, customer t2 where t1.sr_customer_sk(+) =
t2.c_customer_sk
order by 1 desc limit 1;
sr_item_sk | c_customer_id
-----+-----
| AAAAAAAJNGEBAAA
(1 row)

-- Specify the operator (+) in the WHERE clause to indicate a left join and add a join condition.
select t1.sr_item_sk ,t2.c_customer_id from store_returns t1, customer t2 where t1.sr_customer_sk =
t2.c_customer_sk(+) and t2.c_customer_sk(+) < 1 order by 1 limit 1;
sr_item_sk | c_customer_id
-----+-----
1 |
(1 row)

-- If the operator (+) is specified in the WHERE clause, do not use expressions connected through AND/OR.
select t1.sr_item_sk ,t2.c_customer_id from store_returns t1, customer t2 where not(t1.sr_customer_sk =
t2.c_customer_sk(+) and t2.c_customer_sk(+) < 1);
ERROR: Operator "(+)" can not be used in nesting expression.
LINE 1: ...tomer_id from store_returns t1, customer t2 where not(t1.sr_...
^

-- If the operator (+) is specified in the WHERE clause which does not support expression macros, an error
will be reported.
select t1.sr_item_sk ,t2.c_customer_id from store_returns t1, customer t2 where (t1.sr_customer_sk =
t2.c_customer_sk(+))::bool;
ERROR: Operator "(+)" can only be used in common expression.

-- If the operator (+) is specified on both sides of an expression in the WHERE clause, an error will be
reported.
select t1.sr_item_sk ,t2.c_customer_id from store_returns t1, customer t2 where t1.sr_customer_sk(+) =
t2.c_customer_sk(+);
ERROR: Operator "(+)" can't be specified on more than one relation in one join condition
HINT: "t1", "t2"...are specified Operator "(+)" in one condition.

-- Delete the tables:
DROP TABLE tpcds.reason_p;

```

## 13.11 SELECT INTO

### Function

**SELECT INTO** defines a new table based on a query result and insert data obtained by query to the new table.

Different from **SELECT**, data found by **SELECT INTO** is not returned to the client. The table columns have the same names and data types as the output columns of the **SELECT**.

## Precautions

**CREATE TABLE AS** provides functions similar to **SELECT INTO** in functions and provides a superset of functions provided by **SELECT INTO**. You are advised to use **CREATE TABLE AS**, because **SELECT INTO** cannot be used in a stored procedure.

## Syntax

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    { * | {expression [ [ AS ] output_name ]} [, ...] }
INTO [ UNLOGGED ] [ TABLE ] new_table
[ FROM from_item [, ...] ]
[ WHERE condition ]
[ GROUP BY expression [, ...] ]
[ HAVING condition [, ...] ]
[ WINDOW {window_name AS ( window_definition )} [, ...] ]
[ { UNION | INTERSECT | EXCEPT | MINUS } [ ALL | DISTINCT ] select ]
[ ORDER BY {expression [ [ ASC | DESC | USING operator ] | nlssort_expression_clause ] [ NULLS { FIRST |
LAST } ]} [, ...] ]
[ LIMIT { count | ALL } ]
[ OFFSET start [ ROW | ROWS ] ]
[ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ]
[ {FOR { UPDATE | SHARE } [ OF table_name [, ...] ] [ NOWAIT ]} [...] ];
```

## Parameter Description

### INTO [ UNLOGGED ] [ TABLE ] new\_table

**UNLOGGED** indicates that the table is created as an unlogged table. Data written to unlogged tables is not written to the write-ahead log, which makes them considerably faster than ordinary tables. However, they are not crash-safe: an unlogged table is automatically truncated after a crash or unclean shutdown. The contents of an unlogged table are also not replicated to standby servers. Any indexes created on an unlogged table are automatically unlogged as well.

**new\_table** specifies the name of the new table.

#### NOTE

For details on other **SELECT INTO** parameters, see [Parameter Description](#) in **SELECT**.

## Examples

```
-- Add values that are less than 5 in the r_reason_sk column in the tpcds.reason table to the new table.
SELECT * INTO tpcds.reason_t1 FROM tpcds.reason WHERE r_reason_sk < 5;
INSERT 0 6

-- Delete the tpcds.reason_t1 table.
DROP TABLE tpcds.reason_t1;
```

## Helpful Links

[SELECT](#)

## 13.12 UPDATE

### Function

**UPDATE** updates data in a table. **UPDATE** changes the values of the specified columns in all rows that satisfy the condition. The **WHERE** clause clarifies conditions. The columns to be modified need be mentioned in the **SET** clause; columns not explicitly modified retain their previous values.

### Precautions

- You must have the **UPDATE** permission on a table to be updated.
- You must have the **SELECT** permission on all tables involved in the expressions or conditions.
- The distribution column of a table cannot be modified.
- For column-store tables, the **RETURNING** clause is currently not supported.
- Column-store tables do not support non-deterministic update. If you update data in one row with multiple rows of data in a column-store table, an error is reported.
- Memory space that records update operations in column-store tables is not reclaimed. You need to clean it by executing **VACUUM FULL table\_name**.
- Currently, **UPDATE** cannot be used in column-store replication tables.
- You are not advised to create a table that needs to be frequently updated as a replication table.

### Syntax

```
UPDATE [ ONLY ] table_name [ * ] [ [ AS ] alias ]
SET {column_name = { expression | DEFAULT }
    | ( column_name [, ...] ) = ( ( { expression | DEFAULT } [, ...] ) |sub_query ) } [, ...]
[ FROM from_list ] [ WHERE condition ]
[ RETURNING { *
    | {output_expression [ [ AS ] output_name ] } [, ...] }];
```

where sub\_query can be:

```
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
{ * | {expression [ [ AS ] output_name ] } [, ...] }
[ FROM from_item [, ...] ]
[ WHERE condition ]
[ GROUP BY grouping_element [, ...] ]
[ HAVING condition [, ...] ]
```

### Parameter Description

- **table\_name**  
Name (optionally schema-qualified) of the table to be updated.  
Value range: An existing table name.
- **alias**  
Specifies the alias for the target table.  
Value range: A string. It must comply with the naming convention.

- **column\_name**  
Renames a column.  
You can refer to this column by specifying the table name and column name of the target table. Example:  
UPDATE foo SET foo.col\_name = 'GaussDB';  
You can refer to this column by specifying the target table alias and the column name. For example:  
UPDATE foo AS f SET f.col\_name = 'GaussDB';  
Value range: an existing column name
- **expression**  
An expression or value to assign to the column.
- **DEFAULT**  
Sets the column to its default value.  
The value is **NULL** if no specified default value has been assigned to it.
- **sub\_query**  
Specifies a subquery.  
This command can be executed to update a table with information for other tables in the same database. For details about clauses in the **SELECT** statement, see [SELECT](#).
- **from\_list**  
A list of table expressions, allowing columns from other tables to appear in the **WHERE** condition and the update expressions. This is similar to the list of tables that can be specified in the **FROM** clause of a **SELECT** statement.

---

**NOTICE**

Note that the target table must not appear in the **from\_list**, unless you intend a self-join (in which case it must appear with an alias in the **from\_list**).

- **condition**  
An expression that returns a value of type **Boolean**. Only rows for which this expression returns **true** are updated.
- **output\_expression**  
An expression to be computed and returned by the **UPDATE** command after each row is updated.  
Value range: The expression can use any column names of the table named by **table\_name** or table(s) listed in **FROM**. Write \* to return all columns.
- **output\_name**  
A name to use for a returned column.

## Examples

```
-- Create the student1 table:  
CREATE TABLE student1  
(  
  stuno    int,  
  classno  int
```

```
)  
DISTRIBUTE BY hash(stuno);  
  
-- Insert data:  
INSERT INTO student1 VALUES(1,1);  
INSERT INTO student1 VALUES(2,2);  
INSERT INTO student1 VALUES(3,3);  
  
-- View data:  
SELECT * FROM student1;  
  
-- Update the values of all records:  
UPDATE student1 SET classno = classno*2;  
  
-- View data:  
SELECT * FROM student1;  
  
-- Delete the tables:  
DROP TABLE student1;
```

## 13.13 VALUES

### Function

**VALUES** computes a row or a set of rows based on given values. It is most commonly used to generate a constant table within a large command.

### Precautions

- **VALUES** lists with large numbers of rows should be avoided, as you might encounter out-of-memory failures or poor performance. **VALUES** appearing within **INSERT** is a special case, because the desired column types are known from the **INSERT**'s target table, and need not be inferred by scanning the **VALUES** list. In this case, **VALUE** can handle larger lists than are practical in other contexts.
- If more than one row is specified, all the rows must have the same number of elements.

### Syntax

```
VALUES {( expression [, ...] )} [, ...]  
[ ORDER BY { sort_expression [ ASC | DESC | USING operator ] } [, ...] ]  
[ LIMIT { count | ALL } ]  
[ OFFSET start [ ROW | ROWS ] ]  
[ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ];
```

### Parameter Description

- **expression**  
Specifies a constant or expression to compute and insert at the indicated place in the resulting table or set of rows.  
In a **VALUES** list appearing at the top level of an **INSERT**, an expression can be replaced by **DEFAULT** to indicate that the destination column's default value should be inserted. **DEFAULT** cannot be used when **VALUES** appears in other contexts.
- **sort\_expression**



Specifies an expression or integer constant indicating how to sort the result rows.

- **ASC**  
Indicates ascending sort order.
- **DESC**  
Indicates descending sort order.
- **operator**  
Specifies a sorting operator.
- **count**  
Specifies the maximum number of rows to return.
- **OFFSET start { ROW | ROWS }**  
Specifies the maximum number of returned rows, whereas **start** specifies the number of rows to skip before starting to return rows.
- **FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY**  
The **FETCH** clause restricts the total number of rows starting from the first row of the return query result, and the default value of **count** is **1**.

## Examples

See [Examples](#) in **INSERT**.

# 14 DCL Syntax

---

## 14.1 DCL Syntax Overview

Data control language (DCL) is used to set or modify database users or role rights.

### Granting Rights

GaussDB(DWS) provides a statement for granting rights to data objects and roles. For details, see [GRANT](#).

### Revoking Rights

GaussDB(DWS) provides a statement for revoking rights. For details, see [REVOKE](#).

### Setting Default Rights

GaussDB(DWS) allows users to set rights for objects that will be created. For details, see [ALTER DEFAULT PRIVILEGES](#).

## 14.2 ALTER DEFAULT PRIVILEGES

### Function

**ALTER DEFAULT PRIVILEGES** allows you to set the permissions that will be used for objects created in the future. (It does not affect permissions assigned to existing objects.) To isolate permissions, GaussDB(DWS) disables the **WITH GRANT OPTION** syntax.

### Precautions

Only the permissions for tables (including views), functions, and types (including domains) can be altered.

### Syntax

```
ALTER DEFAULT PRIVILEGES  
[ FOR { ROLE | USER } target_role [, ...] ]
```

- [ IN SCHEMA schema\_name [, ...] ]  
abbreviated\_grant\_or\_revoke;

  - **abbreviated\_grant\_or\_revoke** grants or revokes permissions on certain objects.

```
grant_on_tables_clause
| grant_on_functions_clause
| grant_on_types_clause
| revoke_on_tables_clause
| revoke_on_functions_clause
| revoke_on_types_clause
```
  - **grant\_on\_tables\_clause** grants permissions on tables.

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES }
[, ...] | ALL [ PRIVILEGES ] }
ON TABLES
TO { [ GROUP ] role_name | PUBLIC } [, ...]
[ WITH GRANT OPTION ]
```
  - **grant\_on\_functions\_clause** grants permissions on functions.

```
GRANT { EXECUTE | ALL [ PRIVILEGES ] }
ON FUNCTIONS
TO { [ GROUP ] role_name | PUBLIC } [, ...]
[ WITH GRANT OPTION ]
```
  - **grant\_on\_types\_clause** grants permissions on types.

```
GRANT { USAGE | ALL [ PRIVILEGES ] }
ON TYPES
TO { [ GROUP ] role_name | PUBLIC } [, ...]
[ WITH GRANT OPTION ]
```
  - **revoke\_on\_tables\_clause** revokes permissions on tables.

```
REVOKE [ GRANT OPTION FOR ]
{ { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES }
[, ...] | ALL [ PRIVILEGES ] }
ON TABLES
FROM { [ GROUP ] role_name | PUBLIC } [, ...]
[ CASCADE | RESTRICT | CASCADE CONSTRAINTS ]
```
  - **revoke\_on\_functions\_clause** revokes permissions on functions.

```
REVOKE [ GRANT OPTION FOR ]
{ EXECUTE | ALL [ PRIVILEGES ] }
ON FUNCTIONS
FROM { [ GROUP ] role_name | PUBLIC } [, ...]
[ CASCADE | RESTRICT | CASCADE CONSTRAINTS ]
```
  - **revoke\_on\_types\_clause** revokes permissions on types.

```
REVOKE [ GRANT OPTION FOR ]
{ USAGE | ALL [ PRIVILEGES ] }
ON TYPES
FROM { [ GROUP ] role_name | PUBLIC } [, ...]
[ CASCADE | RESTRICT | CASCADE CONSTRAINTS ]
```

## Parameter Description

- **target\_role**

Specifies the name of an existing role. If **FOR ROLE/USER** is omitted, the current role or user is assumed.

Value range: An existing role name.
- **schema\_name**

Specifies the name of an existing schema.

**target\_role** must have the CREATE permissions for **schema\_name**.

Value range: An existing schema name.
- **role\_name**

Specifies the name of an existing role whose permissions are to be granted or revoked.

Value range: An existing role name.

#### NOTICE

To drop a role for which the default permissions have been assigned, to reverse the changes in its default permissions or use **DROP OWNED BY** to get rid of the default privileges entry for the role.

## Examples

```
-- Grant the SELECT permission on all the tables (and views) in tpcds to every user:
ALTER DEFAULT PRIVILEGES IN SCHEMA tpcds GRANT SELECT ON TABLES TO PUBLIC;

-- Create a common user jack:
CREATE USER jack PASSWORD 'Bigdata123@';

-- Grant the INSERT permission on all the tables in tpcds to the user jack:
ALTER DEFAULT PRIVILEGES IN SCHEMA tpcds GRANT INSERT ON TABLES TO jack;

-- Revoke the preceding permissions:
ALTER DEFAULT PRIVILEGES IN SCHEMA tpcds REVOKE SELECT ON TABLES FROM PUBLIC;
ALTER DEFAULT PRIVILEGES IN SCHEMA tpcds REVOKE INSERT ON TABLES FROM jack;

-- Delete the user jack:
DROP USER jack;

-- Assume that there are two users test1 and test2. If you require that user test2 can query tables created
by user test1, execute the following statements:
Grant user test2 the schema permission of user test1.
grant usage, create on schema test1 to test2;
Grant user test2 the table query permission of user test1.
ALTER DEFAULT PRIVILEGES FOR USER test1 IN SCHEMA test1 GRANT SELECT ON tables TO test2;
Create a table as user test1.
set role test1 password 'Gauss_234';
create table test3( a int, b int);
Execute the following statement as user test2:
set role test2 password 'Gauss_234';
select * from test1.test3;
a | b
---+---
(0 rows)
```

## Helpful Links

[GRANT, REVOKE](#)

## 14.3 ANALYZE | ANALYSE

### Function

**ANALYZE** collects statistics about ordinary tables in a database, and stores the results in the **PG\_STATISTIC** system catalog. The execution plan generator uses these statistics to determine which one is the most effective execution plan.

If no parameters are specified, **ANALYZE** analyzes each table and partitioned table in the current database. You can also specify **table\_name**, **column**, and

**partition\_name** to limit the analysis to a specified table, column, or partitioned table.

**ANALYZE** and **ANALYSE VERIFY** are used to check whether data files of common tables (row-store and column-store tables) in a database are damaged. Currently, this function does not support HDFS tables.

## Precautions

Analyzing non-temporary tables is multi-transaction behavior. Therefore, non-temporary tables cannot be analyzed in an anonymous block, transaction block, function, or stored procedure. In addition, when the **ANALYZE** statement is executed concurrently with some DML statements such as **ALTER TABLE**, **DROP TABLE**, and **UPDATE**, an error may be reported during the analyzing process, for example, an error indicating that the target column cannot be found. In this case, you only need to run the **ANALYZE** statement again to collect statistics. Temporary tables in a stored procedure can be analyzed but their statistics updates cannot be rolled back.

Most **ANALYZE VERIFY** operations are used for abnormal scenario detection, and require a release version. Remote read is not triggered in the **ANALYZE VERIFY** scenario. Therefore, the remote read parameter does not take effect. If the system detects that the page is damaged due to an error in the key system table, the system reports an error and stops the detection.

## Syntax

- Collect statistics information about a table.

```
{ ANALYZE | ANALYSE } [ VERBOSE ]  
[ table_name [ ( column_name [, ...] ) ] ];
```

- Collect statistics about a partitioned table.

```
{ ANALYZE | ANALYSE } [ VERBOSE ]  
[ table_name [ ( column_name [, ...] ) ] ]  
PARTITION ( partition_name );
```

### NOTE

An ordinary partitioned table supports the syntax but not the function of collecting statistics about specified partitions.

- Collect statistics about a foreign table.

```
{ ANALYZE | ANALYSE } [ VERBOSE ]  
{ foreign_table_name | FOREIGN TABLES };
```

- Collect statistics about multiple columns.

```
{ANALYZE | ANALYSE} [ VERBOSE ]  
table_name (( column_1_name, column_2_name [, ...] ));
```

### NOTE

- To sample data in percentage, set **default\_statistics\_target** to a negative number.
- The statistics about a maximum of 32 columns can be collected at a time.
- You are not allowed to collect statistics about multiple columns in system catalogs or HDFS foreign tables.
- Check the data files in the current database.  
{ANALYZE | ANALYSE} VERIFY {FAST|COMPLETE};

 NOTE

- All operations on the database are supported. Because many tables are involved, you are advised to save the result in redirection mode: **gsql -d database -p port -f "verify.sql"> verify\_warning.txt 2>&1**.
  - HDFS tables (internal and foreign tables), temporary tables, and unlog tables are not supported.
  - Note: Only visible tables are checked. Internal table check involves foreign tables on which the internal tables depend and are not displayed or presented externally.
  - This command can be used to process tolerant errors. The assert operation in a debug version may cause the core to fail to execute commands. Therefore, you are advised to perform this operation in a release version.
  - If a key system table is damaged during a full database operation, an error is reported and the operation stops.
- Check the data files of tables and indexes.  
`{ANALYZE | ANALYSE} VERIFY {FAST|COMPLETE} table_name|index_name [CASCADE];`

 NOTE

- You can perform operations on common tables and index tables, but cannot perform **CASCADE** operations on index tables. The reason is that **CASCADE** is used to process all index tables of the primary table. When the index table is checked separately, **CASCADE** is not required.
  - HDFS tables (internal and foreign tables), temporary tables, and unlog tables are not supported.
  - When the primary table is checked, the internal tables of the primary table, such as the **toast** table and **cuDESC** table, are also checked.
  - When the system displays a message indicating that the index table is damaged, you are advised to run the **reindex** command to recreate the index.
- Check the data file of the table partition.  
`{ANALYZE | ANALYSE} VERIFY {FAST|COMPLETE} table_name PARTITION {(partition_name)}[CASCADE];`

 NOTE

- You can detect a single partition of a table, but cannot perform the **CASCADE** operation on index tables.
- HDFS tables (internal and foreign tables), temporary tables, and unlog tables are not supported.

## Parameter Description

- **VERBOSE**  
Enables the display of progress messages.

 NOTE

If this parameter is specified, progress information is displayed by **ANALYZE** to indicate the table that is being processed, and statistics about the table are printed.

- **table\_name**  
Specifies the name (possibly schema-qualified) of a specific table to analyze. If omitted, all regular tables (but not foreign tables) in the current database are analyzed.  
Currently, you can use **ANALYZE** to collect statistics about row-store tables, column-store tables, HDFS tables, ORC- or CARBONDATA-formatted OBS foreign tables, and foreign tables for collaborative analysis.

- Value range: an existing table name
- **column\_name, column\_1\_name, column\_2\_name**  
Specifies the name of a specific column to analyze. All columns are analyzed by default.  
Value range: an existing column name
- **partition\_name**  
Assumes the table is a partitioned table. You can specify **partition\_name** following the keyword **PARTITION** to analyze the statistics of this table. Currently the partitioned table supports the syntax of analyzing a partitioned table, but does not execute this syntax.  
Value range: a partition name in a table
- **foreign\_table\_name**  
Specifies the name (possibly schema-qualified) of a specific table to analyze. The data of the table is stored in HDFS.  
Value range: an existing table name
- **FOREIGN TABLES**  
Analyzes HDFS foreign tables stored in HDFS and accessible to the current user.
- **index\_name**  
Name of the index table to be analyzed. The name may contain the schema name.  
Value range: an existing table name
- **FAST|COMPLETE**  
For row-store tables, the CRC and page header of row-store tables are verified in **FAST** mode. If the verification fails, an alarm is reported. In **COMPLETE** mode, parse and verify the pointers and tuples of row-store tables. For column-store tables, the CRC and magic of column-store tables are verified in **FAST** mode. If the verification fails, an alarm is reported. In **COMPLETE** mode, parse and verify CU of column-store tables.
- **CASCADE**  
In **CASCADE** mode, all indexes of the current table are checked.

## Examples

-- Create a table:

```
CREATE TABLE customer_info
(
  WR_RETURNED_DATE_SK    INTEGER          ,
  WR_RETURNED_TIME_SK    INTEGER          ,
  WR_ITEM_SK             INTEGER          NOT NULL,
  WR_REFUNDED_CUSTOMER_SK INTEGER
)
DISTRIBUTE BY HASH (WR_ITEM_SK);
```

-- Create a partitioned table:

```
CREATE TABLE customer_par
(
  WR_RETURNED_DATE_SK    INTEGER          ,
  WR_RETURNED_TIME_SK    INTEGER          ,
  WR_ITEM_SK             INTEGER          NOT NULL,
```

```
WR_REFUNDED_CUSTOMER_SK INTEGER
)
DISTRIBUTE BY HASH (WR_ITEM_SK)
PARTITION BY RANGE(WR_RETURNED_DATE_SK)
(
PARTITION P1 VALUES LESS THAN(2452275),
PARTITION P2 VALUES LESS THAN(2452640),
PARTITION P3 VALUES LESS THAN(2453000),
PARTITION P4 VALUES LESS THAN(MAXVALUE)
)
ENABLE ROW MOVEMENT;
```

-- Do **ANALYZE** to update statistical information:

```
ANALYZE customer;
```

-- Do **ANALYZE VERBOSE** to update statistics and display table information:

```
ANALYZE VERBOSE customer_info;
INFO: analyzing "cstore.pg_delta_3394584009"(cn_5002 pid=53078)
INFO: analyzing "public.customer_info"(cn_5002 pid=53078)
INFO: analyzing "public.customer_info" inheritance tree(cn_5002 pid=53078)
ANALYZE
```

#### NOTE

If any environment-related fault occurs, check the CN log.

-- Delete the **customer** table:

```
DROP TABLE customer;
DROP TABLE customer_par;
```

## 14.4 DEALLOCATE

### Function

**DEALLOCATE** deallocates a previously prepared statement. If you do not explicitly deallocate a prepared statement, it is deallocated when the session ends.

The **PREPARE** key word is always ignored.

### Precautions

None

### Syntax

```
DEALLOCATE [ PREPARE ] { name | ALL };
```

### Parameter Description

- **name**  
Specifies the name of the prepared statement to deallocate.
- **ALL**  
Deallocates all prepared statements.



## Examples

None

# 14.5 DO

## Function

**DO** executes an anonymous code block.

A code block is a function body without parameters that returns void. It is analyzed and executed at the same time.

## Precautions

- Before using a programming language, install it in the current database using **CREATE LANGUAGE**. If no language is specified, **plpgsql** is installed by default.
- To use an untrusted language, you must be a system administrator or have the USAGE permission for programming languages.

## Syntax

```
DO [ LANGUAGE lang_name ] code;
```

## Parameter Description

- **lang\_name**  
Parses the programming language used by the code. If not specified, the default value **plpgsql** is used.
- **code**  
Specifies executable programming language code. The language is specified as a string.

## Examples

```
-- Create user webuser:  
CREATE USER webuser PASSWORD 'Bigdata123@';  
  
-- Grant to user webuser all the operation permissions on views in the tpcds schema:  
DO $$DECLARE r record;  
BEGIN  
  FOR r IN SELECT c.relname,n.nspname FROM pg_class c,pg_namespace n  
    WHERE c.relnamespace = n.oid AND n.nspname = 'tpcds' AND relkind IN ('r','v')  
  LOOP  
    EXECUTE 'GRANT ALL ON ' || quote_ident(r.table_schema) || '.' || quote_ident(r.table_name) || ' TO  
webuser';  
  END LOOP;  
END$$;  
  
-- Delete user webuser:  
DROP USER webuser CASCADE;
```

## 14.6 EXECUTE

### Function

**EXECUTE** executes a prepared statement. A prepared statement only exists in the lifecycle of a session. Therefore, only prepared statements created using **PREPARE** earlier in the session can be executed.

### Precautions

If the **PREPARE** statement creating the prepared statement declares certain parameters, the parameter set transferred to the **EXECUTE** statement must be compatible. Otherwise, an error occurs.

### Syntax

```
EXECUTE name [ ( parameter [, ...] ) ];
```

### Parameter Description

- **name**  
Specifies the name of the statement to be executed.
- **parameter**  
Specifies a parameter of the prepared statement. It must be an expression that generates a value compatible with the data type specified when the prepared statement is created.

### Examples

```
-- Create the reason table:  
CREATE TABLE tpcds.reason (  
  CD_DEMO_SK      INTEGER      NOT NULL,  
  CD_GENDER       character(16) ,  
  CD_MARITAL_STATUS character(100)  
)  
;  
  
-- Insert data.  
INSERT INTO tpcds.reason VALUES(51, 'AAAAAAAADDAAAAAA', 'reason 51');  
  
-- Create the reason_t1 table:  
CREATE TABLE tpcds.reason_t1 AS TABLE reason;  
  
-- Create and execute a prepared statement for the INSERT statement:  
PREPARE insert_reason(integer,character(16),character(100)) AS INSERT INTO tpcds.reason_t1  
VALUES($1,$2,$3);  
  
EXECUTE insert_reason(52, 'AAAAAAAADDAAAAAA', 'reason 52');  
  
-- Delete tables reason and reason_t1:  
DROP TABLE tpcds.reason;  
DROP TABLE tpcds.reason_t1;
```

## 14.7 EXECUTE DIRECT

### Function

**EXECUTE DIRECT** executes an SQL statement on a specified node. Generally, the cluster automatically allocates an SQL statement to proper nodes. **EXECUTE DIRECT** is mainly used for database maintenance and testing.

### Precautions

- Only a system administrator can run the **EXECUTE DIRECT** statement.
- To ensure data consistency across nodes, only the **SELECT** statement can be used. Transaction statements, DDL, and DML cannot be used.
- When the AVG aggregation calculation is performed on the specified DN using such statements, the result set is returned in array, for example, {4,2}. The result of sum is 4, and that of count is 2.
- Do not run the **SELECT** statement on nodes where CNs reside because user table data is not stored there.
- **EXECUTE DIRECT** cannot be nested. If the inner SQL statement to be executed is also **EXECUTE DIRECT**, run only the bottom-layer **EXECUTE DIRECT** statement.

### Syntax

```
EXECUTE DIRECT ON ( nodename [, ... ] ) query ;
```

### Parameter Description

- **nodename**  
Specifies the node name.  
Value range: An existing node.
- **query**  
Specifies the query SQL statement that you want to execute.

### Examples

```
-- Query node distribution status in the current cluster:
SELECT * FROM pgxc_node;
node_name | node_type | node_port | node_host | node_port1 | node_host1 | hostis_primary |
nodeis_primary | nodeis_preferred | node_id | sctp_port | control_port | sctp_port1 | control_port1
-----+-----+-----+-----+-----+-----+-----+
cn_5001 | C | 8050 | 10.180.155.74 | 8050 | 10.180.155.74 | t | f |
f | 1120683504 | 0 | 0 | 0 | 0 |
cn_5003 | C | 8050 | 10.180.157.130 | 8050 | 10.180.157.130 | t | f |
f | -125853378 | 0 | 0 | 0 | 0 |
dn_6001_6002 | D | 40050 | 10.180.155.74 | 45050 | 10.146.187.231 | t | f |
f | 1644780306 | 40052 | 40052 | 45052 | 45052 |
dn_6003_6004 | D | 40050 | 10.146.187.231 | 45050 | 10.180.157.130 | t | f |
f | -966646068 | 40052 | 40052 | 45052 | 45052 |
dn_6005_6006 | D | 40050 | 10.180.157.130 | 45050 | 10.180.155.74 | t | f |
f | 868850011 | 40052 | 40052 | 45052 | 45052 |
cn_5002 | C | 8050 | localhost | 8050 | localhost | t | f | f |
-1736975100 | 0 | 0 | 0 | 0
```

```
(6 rows)
-- Query records in table tpcds.customer_address on dn_6001_6002:
EXECUTE DIRECT ON(dn_6001_6002) 'select count(*) from tpcds.customer_address';
count
-----
16922
(1 row)

-- Query all records in table tpcds.customer_address:
SELECT count(*) FROM tpcds.customer_address;
count
-----
50000
(1 row)
```

## 14.8 GRANT

### Function

**GRANT** grants rights to roles and users.

**GRANT** is used in the following scenarios:

- **Granting system permissions to roles or users**

System permissions are also called user attributes, including **SYSADMIN**, **CREATEDB**, **CREATEROLE**, **AUDITADMIN**, and **LOGIN**.

They can be specified only by the **CREATE ROLE** or **ALTER ROLE** syntax. The **SYSADMIN** permission can be granted and revoked using **GRANT ALL PRIVILEGE** and **REVOKE ALL PRIVILEGE**, respectively. System permissions cannot be inherited by a user from a role, and cannot be granted using **PUBLIC**.

- **Granting database object permissions to roles or users**

Grant permissions related to database objects (tables, views, specified columns, databases, functions, and schemas) to specified roles or users.

**GRANT** grants specified database object permissions to one or more roles. These permissions are appended to those already granted, if any.

GaussDB(DWS) grants the permissions for objects of certain types to **PUBLIC**. By default, permissions for tables, table columns, sequences, external data sources, external servers, schemas, and tablespace are not granted to **PUBLIC**. However, permissions for the following objects are granted to **PUBLIC**: **CONNECT** and **CREATE TEMP TABLE** permissions for databases, **EXECUTE** permission for functions, and **USAGE** permission for languages and data types (including domains). An object owner can revoke the default permissions granted to **PUBLIC** and grant permissions to other users as needed. For security purposes, you are advised to create an object and set permissions for it in the same transaction so that other users do not have time windows to use the object. In addition, you can run the **ALTER DEFAULT PRIVILEGES** statement to modify the initial default permissions.

- **Granting a role's or user's permissions to other roles or users**

Grant a role's or user's permissions to one or more roles or users. In this case, every role or user can be regarded as a set of one or more database permissions.

If **WITH ADMIN OPTION** is specified, the member can in turn grant permissions in the role to others, and revoke permissions in the role as well. If a role or user granted with certain permissions is changed or revoked, the permissions inherited from the role or user also change.

A database administrator can grant permissions to and revoke them from any role or user. Roles having **CREATEROLE** permission can grant or revoke membership in any role that is not an administrator.

## Precautions

None

To isolate permissions, GaussDB(DWS) disables the **WITH GRANT OPTION** and **TO PUBLIC** syntax.

## Syntax

- Grant the table or view access permission to a specified role or user. **GRANT** to a table partition will cause alarms, and therefore is forbidden.

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES } [, ...]
        | ALL [ PRIVILEGES ] }
ON { [ TABLE ] table_name [, ...]
     | ALL TABLES IN SCHEMA schema_name [, ...] }
TO { [ GROUP ] role_name | PUBLIC } [, ...]
[ WITH GRANT OPTION ];
```

- Grant the column access permission to a specified role or user.

```
GRANT { { SELECT | INSERT | UPDATE | REFERENCES } ( column_name [, ...] ) } [, ...]
        | ALL [ PRIVILEGES ] ( column_name [, ...] ) }
ON [ TABLE ] table_name [, ...]
TO { [ GROUP ] role_name | PUBLIC } [, ...]
[ WITH GRANT OPTION ];
```

- Grant the database access permission to a specified role or user.

```
GRANT { { CREATE | CONNECT | TEMPORARY | TEMP } [, ...]
        | ALL [ PRIVILEGES ] }
ON DATABASE database_name [, ...]
TO { [ GROUP ] role_name | PUBLIC } [, ...]
[ WITH GRANT OPTION ];
```

- Grant the domain access permission to a specified role or user.

```
GRANT { USAGE | ALL [ PRIVILEGES ] }
ON DOMAIN domain_name [, ...]
TO { [ GROUP ] role_name | PUBLIC } [, ...]
[ WITH GRANT OPTION ];
```

### NOTE

The current version does not support granting the domain access permission.

- Grant the external data source access permission to a specified role or user.

```
GRANT { USAGE | ALL [ PRIVILEGES ] }
ON FOREIGN DATA WRAPPER fdw_name [, ...]
TO { [ GROUP ] role_name | PUBLIC } [, ...]
[ WITH GRANT OPTION ];
```

- Grant the external server access permission to a specified role or user.

```
GRANT { USAGE | ALL [ PRIVILEGES ] }
ON FOREIGN SERVER server_name [, ...]
TO { [ GROUP ] role_name | PUBLIC } [, ...]
[ WITH GRANT OPTION ];
```

- Grant the function access permission to a specified role or user.

```
GRANT { EXECUTE | ALL [ PRIVILEGES ] }
ON { FUNCTION {function_name ( [ { [ argmode ] [ arg_name ] arg_type } [, ...] ) } } [, ...]
     | ALL FUNCTIONS IN SCHEMA schema_name [, ...] }
```

```
TO { [ GROUP ] role_name | PUBLIC } [, ...]
[ WITH GRANT OPTION ];
```

- Grant the procedural language access permission to a specified role or user.

```
GRANT { USAGE | ALL [ PRIVILEGES ] }
ON LANGUAGE lang_name [, ...]
TO { [ GROUP ] role_name | PUBLIC } [, ...]
[ WITH GRANT OPTION ];
```

 **NOTE**

The current version does not support granting the procedural language access permission.

- Grant the large object access permission to a specified role or user.

```
GRANT { { SELECT | UPDATE } [, ...] | ALL [ PRIVILEGES ] }
ON LARGE OBJECT loid [, ...]
TO { [ GROUP ] role_name | PUBLIC } [, ...]
[ WITH GRANT OPTION ];
```

 **NOTE**

The current version does not support granting the large object access permission.

- Grant the sub-cluster access permission to a specified role or user. Common users cannot grant or revoke the Node Group access permission.

```
GRANT { CREATE | USAGE | COMPUTE | ALL [ PRIVILEGES ] }
ON NODE GROUP group_name [, ...]
TO { [ GROUP ] role_name | PUBLIC } [, ...]
[ WITH GRANT OPTION ];
```

- Grant the schema access permission to a specified role or user.

```
GRANT { { CREATE | USAGE } [, ...] | ALL [ PRIVILEGES ] }
ON SCHEMA schema_name [, ...]
TO { [ GROUP ] role_name | PUBLIC } [, ...]
[ WITH GRANT OPTION ];
```

 **NOTE**

When you grant table or view rights to other users, you also need to grant the USAGE permission for the schema that the tables and views belong to. Without this permission, the users granted with the table or view rights can only see the object names, but cannot access them.

- Grant the type access permission to a specified role or user.

```
GRANT { USAGE | ALL [ PRIVILEGES ] }
ON TYPE type_name [, ...]
TO { [ GROUP ] role_name | PUBLIC } [, ...]
[ WITH GRANT OPTION ];
```

 **NOTE**

The current version does not support granting the type access permission.

- Grant a role's rights to other users or roles.

```
GRANT role_name [, ...]
TO role_name [, ...]
[ WITH ADMIN OPTION ];
```

- Grant the SYSADMIN permission to a specified role.

```
GRANT ALL { PRIVILEGES | PRIVILEGE }
TO role_name;
```

## Parameter Description

**GRANT** grants the following permissions:

- **SELECT**

Allows **SELECT** from any column, or the specific columns listed, of the specified table, view, or sequence.

- **INSERT**

Allows **INSERT** of a new row into the specified table.

- **UPDATE**

Allows **UPDATE** of any column, or the specific columns listed, of the specified table. **SELECT ... FOR UPDATE** and **SELECT ... FOR SHARE** also require this permission on at least one column, in addition to the **SELECT** permission.

- **DELETE**

Allows **DELETE** of a row from the specified table.

- **TRUNCATE**

Allows **TRUNCATE** on the specified table.

- **REFERENCES**

To create a foreign key constraint, it is necessary to have this permission on both the referencing and referenced columns.

- **CREATE**

- For databases, allows new schemas to be created within the database.
- For schemas, allows new objects to be created within the schema. To rename an existing object, you must own the object and have this permission for the schema where the object is located.
- For sub-clusters, allows tables to be created.

- **CONNECT**

Allows the user to connect to the specified database.

- **EXECUTE**

Allows the use of the specified function and the use of any operators that are implemented on top of the function.

- **USAGE**

- For procedural languages, allows the use of the specified language for the creation of functions in that language.
- For schemas, allows access to objects contained in the specified schema. Without this permission, it is still possible to see the object names.
- For sequences, allows the use of the `nextval` function.
- For sub-clusters, allows users who can access objects contained in the specified schema to access tables in a specified sub-cluster.

- **COMPUTE**

Allows users to perform elastic computing in a computing sub-cluster that they have the compute permission on.

- **ALL PRIVILEGES**

Grants all of the available permissions at once. Only system administrators have permission to run **GRANT ALL PRIVILEGES**.

**GRANT** parameters are as follows:

- **role\_name**

Specifies an existing user name.

- **table\_name**  
Specifies an existing table name.
- **column\_name**  
Specifies an existing column name.
- **schema\_name**  
Specifies an existing schema name.
- **database\_name**  
Specifies an existing database name.
- **function\_name**  
Specifies an existing function name.
- **sequence\_name**  
Specifies an existing sequence name.
- **domain\_name**  
Specifies an existing domain type.
- **fdw\_name**  
Specifies an existing foreign data wrapper name.
- **lang\_name**  
Specifies an existing language name.
- **type\_name**  
Specifies an existing type name.
- **group\_name**  
Specifies an existing sub-cluster name.
- **argmode**  
Specifies the parameter mode.  
Value range: a string. It must comply with the naming convention.
- **arg\_name**  
Indicates the parameter name.  
Value range: A string. It must comply with the naming convention rule.
- **arg\_type**  
Specifies the parameter type.  
Value range: A string. It must comply with the naming convention rule.
- **loid**  
Identifier of the large object that includes this page  
Value range: A string. It must comply with the naming convention rule.
- **directory\_name**  
Specifies a directory name.  
Value range: a string. It must comply with the naming convention.

## Examples

### Example: Grant system rights to a user or role.

Create user **joe** and grant the sysadmin permissions to it.



```
CREATE USER joe PASSWORD 'Bigdata123@';  
GRANT ALL PRIVILEGES TO joe;
```

Afterward, user **joe** has the sysadmin permissions.

### Example: Grant object rights to a user or role.

1. Revoke user **joe**'s sysadmin permissions, and grant it with the usage permission of the **tpcds** schema and all the rights of the **tpcds.reason** table.

```
REVOKE ALL PRIVILEGES FROM joe;  
GRANT USAGE ON SCHEMA tpcds TO joe;  
GRANT ALL PRIVILEGES ON tpcds.reason TO joe;
```

After the granting succeeds, user **joe** has all the rights of the **tpcds.reason** table, including addition, deletion, modification, and query rights.

2. Grant the query permission for **r\_reason\_sk**, **r\_reason\_id**, and **r\_reason\_desc** columns and the update permission for the **r\_reason\_desc** column in the **tpcds.reason** table to user **joe**.

```
GRANT select (r_reason_sk,r_reason_id,r_reason_desc),update (r_reason_desc) ON tpcds.reason TO joe;
```

After the granting succeeds, user **joe** immediately has the query permission of **r\_reason\_sk** and **r\_reason\_id** columns in the **tpcds.reason** table.

```
GRANT select (r_reason_sk, r_reason_id) ON tpcds.reason TO joe ;
```

Grant the **postgres** database connection permission and schema creation permission to user **joe**.

```
GRANT create,connect on database postgres TO joe ;
```

Create role **tpcds\_manager**, grant the **tpcds** schema access permission and object creation permission to this role, but do not enable it to grant these permissions to others.

```
CREATE ROLE tpcds_manager PASSWORD 'Bigdata123@';  
GRANT USAGE,CREATE ON SCHEMA tpcds TO tpcds_manager;
```

### Example: Grant a user's or role's rights to other users or roles.

1. Create role **manager**, grant user **joe**'s rights to this role, and enable this role to grant its rights to others.

```
CREATE ROLE manager PASSWORD 'Bigdata123@';  
GRANT joe TO manager WITH ADMIN OPTION;
```

2. Create user **senior\_manager** and grant **manager**'s rights to it.

```
CREATE ROLE senior_manager PASSWORD 'Bigdata123@';  
GRANT manager TO senior_manager;
```

3. Revoke the rights and delete the user.

```
REVOKE manager FROM joe;  
REVOKE senior_manager FROM manager;  
DROP USER manager;
```

### Example: Revoke the granted rights and delete the roles and users.

```
REVOKE USAGE,CREATE ON SCHEMA tpcds FROM tpcds_manager;  
DROP ROLE tpcds_manager;  
DROP ROLE senior_manager;  
DROP USER joe CASCADE;
```

## Helpful Links

[REVOKE, ALTER DEFAULT PRIVILEGES](#)

## 14.9 PREPARE

### Function

**PREPARE** creates a prepared statement.

A prepared statement is a performance optimizing object on the server. When the **PREPARE** statement is executed, the specified query is parsed, analyzed, and rewritten. When the **EXECUTE** is executed, the prepared statement is planned and executed. This avoids repetitive parsing and analysis. After the **PREPARE** statement is created, it exists throughout the database session. Once it is created (even if in a transaction block), it will not be deleted when a transaction is rolled back. It can only be deleted by explicitly invoking **DEALLOCATE** or automatically deleted when the session ends.

### Precautions

None

### Syntax

```
PREPARE name [ ( data_type [, ...] ) ] AS statement;
```

### Parameter Description

- **name**  
Specifies the name of a prepared statement. It must be unique in the current session.
- **data\_type**  
Specifies the type of a parameter.
- **statement**  
Specifies a **SELECT**, **INSERT**, **UPDATE**, **DELETE**, or **VALUES** statement.

### Examples

See [Examples](#) in **EXECUTE**.

### Helpful Links

[DEALLOCATE](#)

## 14.10 REASSIGN OWNED

### Function

**REASSIGN OWNED** changes the owner of a database.

**REASSIGN OWNED** requires that the system change owners of all the database objects owned by **old\_roles** to **new\_role**.

## Precautions

- **REASSIGN OWNED** is often executed before deleting a rule.
- You must have the permissions on the original and target roles to execute it.

## Syntax

```
REASSIGN OWNED BY old_role [, ...] TO new_role;
```

## Parameter Description

- **old\_role**  
Specifies the role name of the old owner.
- **new\_role**  
Specifies the role name of the new owner.

## Examples

None

# 14.11 REVOKE

## Function

**REVOKE** revokes rights from one or more roles.

## Precautions

If a non-owner user of an object attempts to **REVOKE** rights on the object, the command is executed based on the following rules:

- If the user has no right whatsoever on the object, the command will fail outright.
- If some permissions are available, the command proceeds, but it revokes only those rights for which the user has grant options.
- The **REVOKE ALL PRIVILEGES** forms will issue an error message if no grant options are held, while the other forms will issue a warning if grant options for any of the rights named in the command are not held.
- Do not perform **REVOKE** to a table partition. Performing **REVOKE** to a partitioned table incurs an alarm.

## Syntax

- Revoke the permission of specified table and view.

```
REVOKE [ GRANT OPTION FOR ]  
  { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES }[, ...]  
  | ALL [ PRIVILEGES ] }  
ON { [ TABLE ] table_name [, ...]  
  | ALL TABLES IN SCHEMA schema_name [, ...] }  
FROM { [ GROUP ] role_name | PUBLIC } [, ...]  
[ CASCADE | RESTRICT];
```
- Revoke the permission of specified fields on the table.

```
REVOKE [ GRANT OPTION FOR ]  
  { { { SELECT | INSERT | UPDATE | REFERENCES } ( column_name [, ...] ) }[, ...]
```

- ```
| ALL [ PRIVILEGES ] ( column_name [, ...] ) }  
ON [ TABLE ] table_name [, ...]  
FROM { [ GROUP ] role_name | PUBLIC } [, ...]  
[ CASCADE | RESTRICT ];
```
- Revoke the permission of a specified database.  
REVOKE [ GRANT OPTION FOR ]  
{ { CREATE | CONNECT | TEMPORARY | TEMP } [, ...]  
| ALL [ PRIVILEGES ] }  
ON DATABASE database\_name [, ...]  
FROM { [ GROUP ] role\_name | PUBLIC } [, ...]  
[ CASCADE | RESTRICT ];
  - Revoke the permission of a specified function.  
REVOKE [ GRANT OPTION FOR ]  
{ EXECUTE | ALL [ PRIVILEGES ] }  
ON { FUNCTION {function\_name ( [ { [ argmode ] [ arg\_name ] arg\_type } [, ...] ) } } [, ...]  
| ALL FUNCTIONS IN SCHEMA schema\_name [, ...] }  
FROM { [ GROUP ] role\_name | PUBLIC } [, ...]  
[ CASCADE | RESTRICT ];
  - Revoke the permission of a specified large object.  
REVOKE [ GRANT OPTION FOR ]  
{ { SELECT | UPDATE } [, ...] | ALL [ PRIVILEGES ] }  
ON LARGE OBJECT loid [, ...]  
FROM { [ GROUP ] role\_name | PUBLIC } [, ...]  
[ CASCADE | RESTRICT ];
  - Revoke the permission of a specified schema.  
REVOKE [ GRANT OPTION FOR ]  
{ { CREATE | USAGE } [, ...] | ALL [ PRIVILEGES ] }  
ON SCHEMA schema\_name [, ...]  
FROM { [ GROUP ] role\_name | PUBLIC } [, ...]  
[ CASCADE | RESTRICT ];
  - Revoke the permission of a specified sub-cluster.  
REVOKE [ GRANT OPTION FOR ]  
{ CREATE | USAGE | COMPUTE | ALL [ PRIVILEGES ] }  
ON NODE GROUP group\_name [, ...]  
FROM { [ GROUP ] role\_name | PUBLIC } [, ...]  
[ CASCADE | RESTRICT ];
  - Revoke the permission of roles based on roles.  
REVOKE [ ADMIN OPTION FOR ]  
role\_name [, ...] FROM role\_name [, ...]  
[ CASCADE | RESTRICT ];
  - Revoke the sysadmin permission of roles.  
REVOKE ALL { PRIVILEGES | PRIVILEGE } FROM role\_name;

## Parameter Description

The keyword **PUBLIC** indicates an implicitly defined group that contains all roles.

See the [Parameter Description](#) of the **GRANT** command for the meaning of the privileges and related parameters.

Permissions of a role include the permissions directly granted to the role, permissions inherited from the parent role, and permissions granted to **PUBLIC**. Therefore, revoking the **SELECT** permission for an object from **PUBLIC** does not necessarily mean that the **SELECT** permission for the object has been revoked from all roles, because the **SELECT** permission directly granted to roles and inherited from parent roles still remains. Similarly, if the **SELECT** permission is revoked from a user but is not revoked from **PUBLIC**, the user can still run the **SELECT** statement.

If **GRANT OPTION FOR** is specified, only the grant option for the right is revoked, not the right itself.

If user A holds the **UPDATE** rights on a table and the **WITH GRANT OPTION** and has granted them to user B, the rights that user B holds are called dependent rights. If the rights or the grant option held by user A is revoked, the dependent rights still exist. Those dependent rights are also revoked if **CASCADE** is specified.

A user can only revoke rights that were granted directly by that user. If, for example, user A has granted a right with grant option (**WITH ADMIN OPTION**) to user B, and user B has in turned granted it to user C, then user A cannot revoke the right directly from C. However, user A can revoke the grant option held by user B and use **CASCADE**. In this manner, the rights held by user C are automatically revoked. For another example, if both user A and user B have granted the same right to C, A can revoke his own grant but not B's grant, so C will still effectively have the right.

If the role executing **REVOKE** holds rights indirectly via more than one role membership path, it is unspecified which containing role will be used to execute the command. In such cases, it is best practice to use **SET ROLE** to become the specific role you want to do the **REVOKE** as, and then execute **REVOKE**. Failure to do so may lead to deleting rights not intended to delete, or not deleting any rights at all.

## Examples

See [Examples](#) in **GRANT**.

## Helpful Links

[GRANT](#)

# 15 TCL Syntax

---

## 15.1 TCL Syntax Overview

Transaction Control Language (TCL) controls the time and effect of database transactions and monitors the database.

### Commit

GaussDB(DWS) uses the COMMIT or END statement to commit transactions. For details, see [COMMIT | END](#).

### Setting a Savepoint

GaussDB(DWS) creates a new savepoint in the current transaction. For details, see [SAVEPOINT](#).

### Rollback

GaussDB(DWS) rolls back the current transaction to the last committed state. For details, see [ROLLBACK](#).

## 15.2 ABORT

### Function

**ABORT** rolls back the current transaction and cancels the changes in the transaction.

This command is equivalent to [ROLLBACK](#), and is present only for historical reasons. Now **ROLLBACK** is recommended.

### Precautions

**ABORT** has no impact outside a transaction, but will provoke a warning.

## Syntax

```
ABORT [ WORK | TRANSACTION ] ;
```

## Parameter Description

### WORK | TRANSACTION

Optional keyword has no effect except increasing readability.

## Examples

```
-- Create the customer_demographics_t1 table:
CREATE TABLE customer_demographics_t1
(
  CD_DEMO_SK          INTEGER          NOT NULL,
  CD_GENDER           CHAR(1)         ,
  CD_MARITAL_STATUS  CHAR(1)         ,
  CD_EDUCATION_STATUS CHAR(20)       ,
  CD_PURCHASE_ESTIMATE INTEGER        ,
  CD_CREDIT_RATING   CHAR(10)        ,
  CD_DEP_COUNT        INTEGER         ,
  CD_DEP_EMPLOYED_COUNT INTEGER       ,
  CD_DEP_COLLEGE_COUNT INTEGER
)
WITH (ORIENTATION = COLUMN,COMPRESSION=MIDDLE)
DISTRIBUTE BY HASH (CD_DEMO_SK);

-- Insert data:
INSERT INTO customer_demographics_t1 VALUES(1920801,'M', 'U', 'DOCTOR DEGREE', 200, 'GOOD', 1, 0,0);

-- Start a transaction:
START TRANSACTION;

-- Update the column:
UPDATE customer_demographics_t1 SET cd_education_status= 'Unknown';

-- Abort the transaction. All updates are rolled back.
ABORT;

-- Query data:
SELECT * FROM customer_demographics_t1 WHERE cd_demo_sk = 1920801;
cd_demo_sk | cd_gender | cd_marital_status | cd_education_status | cd_purchase_estimate | cd_credit_rating
| cd_dep_count | cd_dep_employed_count | cd_dep_college_count
+-----+-----+-----+-----+-----+-----+-----+-----+
1920801 | M      | U      | DOCTOR DEGREE      | 200 | GOOD      | 1
| 0 | 0
(1 row)

-- Delete the table:
DROP TABLE customer_demographics_t1;
```

## Helpful Links

[SET TRANSACTION, COMMIT | END, ROLLBACK](#)

## 15.3 BEGIN

### Function

**BEGIN** may be used to initiate an anonymous block or a single transaction. This section describes the syntax of **BEGIN** used to initiate an anonymous block. For

details about the **BEGIN** syntax that initiates transactions, see [START TRANSACTION](#).

An anonymous block is a structure that can dynamically create and execute stored procedure code instead of permanently storing code as a database object in the database.

## Precautions

None

## Syntax

- **Enable an anonymous block:**

```
[DECLARE [declare_statements]]
BEGIN
execution_statements
END;
/
```
- **-- Start a transaction:**

```
BEGIN [ WORK | TRANSACTION ]
[
{
ISOLATION LEVEL { READ COMMITTED | READ UNCOMMITTED | SERIALIZABLE | REPEATABLE
READ }
| { READ WRITE | READ ONLY }
} [, ...]
];
```

## Parameter Description

- **declare\_statements**  
Declares a variable, including its name and type, for example, **sales\_cnt int**.
- **execution\_statements**  
Specifies the statement to be executed in an anonymous block.  
Value range: an existing function name

## Examples

```
-- Generate a string using an anonymous block:
BEGIN
dbms_output.put_line('Hello');
END;
/
```

## Helpful Links

[START TRANSACTION](#)

# 15.4 CHECKPOINT

## Function

A checkpoint is a point in the transaction log sequence at which all data files have been updated to reflect the information in the log. All data files will be flushed to a disk.



**CHECKPOINT** forces a transaction log checkpoint. By default, WALs periodically specify checkpoints in a transaction log. You may use **gs\_guc** to specify run-time parameters **checkpoint\_segments** and **checkpoint\_timeout** to adjust the atomized checkpoint intervals.

## Precautions

- Only a system administrator has the permission to call **CHECKPOINT**.
- **CHECKPOINT** forces an immediate checkpoint when the related command is issued, without waiting for a regular checkpoint scheduled by the system.

## Syntax

```
CHECKPOINT;
```

## Parameter Description

None

## Examples

```
-- Set a checkpoint:  
CHECKPOINT;
```

# 15.5 COMMIT | END

## Function

**COMMIT** or **END** commits all operations of a transaction.

## Precautions

Only the transaction creators or system administrators can run the **COMMIT** command. The creation and commit operations must be in different sessions.

## Syntax

```
{ COMMIT | END } [ WORK | TRANSACTION ] ;
```

## Parameter Description

- **COMMIT | END**  
Commits the current transaction and makes all changes made by the transaction become visible to others.
- **WORK | TRANSACTION**  
Optional keyword has no effect except increasing readability.

## Examples

```
-- Create a table:  
CREATE TABLE tpcds.customer_demographics_t2  
(  
  CD_DEMO_SK      INTEGER      NOT NULL,  
  CD_GENDER      CHAR(1)  
,
```

```
CD_MARITAL_STATUS      CHAR(1)      ,
CD_EDUCATION_STATUS    CHAR(20)     ,
CD_PURCHASE_ESTIMATE    INTEGER      ,
CD_CREDIT_RATING       CHAR(10)    ,
CD_DEP_COUNT           INTEGER      ,
CD_DEP_EMPLOYED_COUNT  INTEGER      ,
CD_DEP_COLLEGE_COUNT   INTEGER
)
WITH (ORIENTATION = COLUMN,COMPRESSION=MIDDLE)
DISTRIBUTE BY HASH (CD_DEMO_SK);

-- Start a transaction:
START TRANSACTION;

-- Insert data:
INSERT INTO tpods.customer_demographics_t2 VALUES(1,'M', 'U', 'DOCTOR DEGREE', 1200, 'GOOD', 1, 0, 0);
INSERT INTO tpods.customer_demographics_t2 VALUES(2,'F', 'U', 'MASTER DEGREE', 300, 'BAD', 1, 0, 0);

-- Commit the transaction to make all changes permanent:
COMMIT;

-- Query data:
SELECT * FROM tpods.customer_demographics_t2;

-- Delete the tpods.customer_demographics_t2 table:
DROP TABLE tpods.customer_demographics_t2;
```

## Helpful Links

[ROLLBACK](#)

# 15.6 COMMIT PREPARED

## Function

**COMMIT PREPARED** commits a prepared two-phase transaction.

## Precautions

- The function is only available in maintenance mode (when GUC parameter **xc\_maintenance\_mode** is **on**). Exercise caution when enabling the mode. It is used by maintenance engineers for troubleshooting. Common users should not use the mode.
- Only the transaction creators or system administrators can run the **COMMIT** command. The creation and commit operations must be in different sessions.
- The transaction function is maintained automatically by the database, and should be not visible to users.

## Syntax

```
COMMIT PREPARED transaction_id ;
COMMIT PREPARED transaction_id WITH CSN;
```

## Parameter Description

- **transaction\_id**  
Specifies the identifier of the transaction to be submitted. The identifier must be different from those for current prepared transactions.

- **CSN(commit sequence number)**  
Specifies the sequence number of the transaction to be committed. It is a 64-bit, incremental, unsigned number.

## Helpful Links

[PREPARE TRANSACTION](#), [ROLLBACK PREPARED](#)

# 15.7 PREPARE TRANSACTION

## Function

**PREPARE TRANSACTION** prepares the current transaction for two-phase commit.

After this command, the transaction is no longer associated with the current session; instead, its state is fully stored on disk, and there is a high probability that it can be committed successfully, even if a database crash occurs before the commit is requested.

Once prepared, a transaction can later be committed or rolled back with **COMMIT PREPARED** or **ROLLBACK PREPARED**, respectively. Those commands can be issued from any session, not only the one that executed the original transaction.

From the point of view of the issuing session, **PREPARE TRANSACTION** is not unlike a **ROLLBACK** command: after executing it, there is no active current transaction, and the effects of the prepared transaction are no longer visible. (The effects will become visible again if the transaction is committed.)

If the **PREPARE TRANSACTION** command fails for any reason, it becomes a **ROLLBACK** and the current transaction is canceled.

## Precautions

- The transaction function is maintained automatically by the database, and should be not visible to users.
- When running the **PREPARE TRANSACTION** command, increasing the value of **max\_prepared\_transactions** in configuration file **postgresql.conf**. You are advised to set **max\_prepared\_transactions** to a value not less than that of **max\_connections** so that one pending prepared transaction is available for each session.

## Syntax

```
PREPARE TRANSACTION transaction_id;
```

## Parameter Description

### **transaction\_id**

An arbitrary identifier that later identifies this transaction for **COMMIT PREPARED** or **ROLLBACK PREPARED**. The identifier must be different from those for current prepared transactions.

Value range: The identifier must be written as a string literal, and must be less than 200 bytes long.

## Helpful Links

[COMMIT PREPARED, ROLLBACK PREPARED](#)

# 15.8 SAVEPOINT

## Function

**SAVEPOINT** establishes a new savepoint within the current transaction.

A savepoint is a special mark inside a transaction that rolls back all commands that are executed after the savepoint was established, restoring the transaction state to what it was at the time of the savepoint.

## Precautions

- Use **ROLLBACK TO SAVEPOINT** to roll back to a savepoint. Use **RELEASE SAVEPOINT** to destroy a savepoint but keep the effects of the commands executed after the savepoint was established.
- Savepoints can only be established when inside a transaction block. There can be multiple savepoints defined within a transaction.
- **SAVEPOINT** cannot be used for functions, anonymous blocks, or stored procedures.
- In the case of an unexpected termination of a distributed thread or process caused by a node or connection failure, or of an error caused by the inconsistency between source and destination table structures in a **COPY FROM** operation, the transaction cannot be rolled back to the established savepoint. Instead, the entire transaction will be rolled back.
- According to the SQL standard, a savepoint is destroyed automatically when another savepoint with the same name is established. In GaussDB(DWS), old savepoints are kept, though only the most recent one will be used for rollback or release. Releasing the newer savepoint with **RELEASE SAVEPOINT** will cause the older one to again become accessible to **ROLLBACK TO SAVEPOINT** and **RELEASE SAVEPOINT**. Except for this, **SAVEPOINT** is fully SQL conforming.

## Syntax

```
SAVEPOINT savepoint_name;
```

## Parameter Description

savepoint\_name

Specifies the name of a new savepoint.

## Examples

```
-- Create a table:  
CREATE TABLE table1(a int);  
  
-- Start a transaction:  
START TRANSACTION;
```

```
-- Insert data:
INSERT INTO table1 VALUES (1);

-- Establish a savepoint:
SAVEPOINT my_savepoint;

-- Insert data:
INSERT INTO table1 VALUES (2);

-- Roll back to the savepoint:
ROLLBACK TO SAVEPOINT my_savepoint;

-- Insert data:
INSERT INTO table1 VALUES (3);

-- Submit the transaction:
COMMIT;

-- Query the table content, which should contain 1 and 3 but not 2, because 2 has been rolled back.
SELECT * FROM table1;

-- Delete the tables:
DROP TABLE table1;

-- Create a table:
CREATE TABLE table2(a int);

-- Start a transaction:
START TRANSACTION;

-- Insert data:
INSERT INTO table2 VALUES (3);

-- Establish a savepoint:
SAVEPOINT my_savepoint;

-- Insert data:
INSERT INTO table2 VALUES (4);

-- Roll back to the savepoint:
RELEASE SAVEPOINT my_savepoint;

-- Submit the transaction:
COMMIT;

-- Query the table content, which should contain both 3 and 4.
SELECT * FROM table2;

-- Delete the tables:
DROP TABLE table2;
```

## Helpful Links

[RELEASE SAVEPOINT, ROLLBACK TO SAVEPOINT](#)

# 15.9 SET TRANSACTION

## Function

**SET TRANSACTION** sets the characteristics of the current transaction. It has no effect on any subsequent transactions. Available transaction characteristics include the transaction separation level and transaction access mode (read/write or read only).

## Precautions

None

## Syntax

Set the isolation level and access mode of the transaction.

```
{ SET [ LOCAL ] TRANSACTION|SET SESSION CHARACTERISTICS AS TRANSACTION }  
{ ISOLATION LEVEL { READ COMMITTED | READ UNCOMMITTED | SERIALIZABLE | REPEATABLE READ }  
| { READ WRITE | READ ONLY } } [, ...]
```

## Parameter Description

- **LOCAL**  
Indicates that the specified command takes effect only for the current transaction.
- **SESSION**  
Indicates that the specified parameters take effect for the current session.  
Value range: a string. It must comply with the naming convention.
- **ISOLATION\_LEVEL\_CLAUSE**  
Specifies the transaction isolation level that determines the data that a transaction can view if other concurrent transactions exist.

### NOTE

- The isolation level of a transaction cannot be reset after the first clause (**INSERT**, **DELETE**, **UPDATE**, **FETCH**, **COPY**) for modifying data is executed in the transaction.

Valid value:

- **READ COMMITTED**: Only committed data is read. This is the default.
- **READ UNCOMMITTED**: GaussDB(DWS) does not support **READ UNCOMMITTED**. If **READ UNCOMMITTED** is set, **READ COMMITTED** is executed instead.
- **REPEATABLE READ**: Only the data committed before transaction start is read. Uncommitted data or data committed in other concurrent transactions cannot be read.
- **SERIALIZABLE**: GaussDB(DWS) does not support **SERIALIZABLE**. If **SERIALIZABLE** is set, **REPEATABLE READ** is executed instead.
- **READ WRITE | READ ONLY**  
Specifies the transaction access mode (read/write or read only).

## Examples

```
-- Start a transaction and set its isolation level to READ COMMITTED and access mode to READ ONLY:  
START TRANSACTION;  
SET LOCAL TRANSACTION ISOLATION LEVEL READ COMMITTED READ ONLY;  
COMMIT;
```

## 15.10 START TRANSACTION

### Function

**START TRANSACTION** starts a transaction. If the isolation level, read/write mode, or deferrable mode is specified, a new transaction will have those characteristics. You can also specify them using [SET TRANSACTION](#).

### Precautions

None

### Syntax

Format 1: START TRANSACTION

```
START TRANSACTION
[
  {
    ISOLATION LEVEL { READ COMMITTED | READ UNCOMMITTED | SERIALIZABLE | REPEATABLE READ }
    | { READ WRITE | READ ONLY }
  } [, ...]
];
```

Format 2: BEGIN

```
BEGIN [ WORK | TRANSACTION ]
[
  {
    ISOLATION LEVEL { READ COMMITTED | READ UNCOMMITTED | SERIALIZABLE | REPEATABLE READ }
    | { READ WRITE | READ ONLY }
  } [, ...]
];
```

### Parameter Description

- **WORK | TRANSACTION**  
Optional keyword in BEGIN format without functions.
- **ISOLATION LEVEL**  
Specifies the transaction isolation level that determines the data that a transaction can view if other concurrent transactions exist.

#### NOTE

The isolation level of a transaction cannot be reset after the first clause (**INSERT**, **DELETE**, **UPDATE**, **FETCH**, **COPY**) for modifying data is executed in the transaction.

Valid value:

- **READ COMMITTED**: Only committed data is read. This is the default.
- **READ UNCOMMITTED**: GaussDB(DWS) does not support **READ UNCOMMITTED**. If **READ UNCOMMITTED** is set, **READ COMMITTED** is executed instead.
- **REPEATABLE READ**: Only the data committed before transaction start is read. Uncommitted data or data committed in other concurrent transactions cannot be read.

- **SERIALIZABLE**: GaussDB(DWS) does not support **SERIALIZABLE**. If **SERIALIZABLE** is set, **REPEATABLE READ** is executed instead.
- **READ WRITE | READ ONLY**  
Specifies the transaction access mode (read/write or read only).

## Examples

```
-- Start a transaction in default mode:
START TRANSACTION;
SELECT * FROM tpcds.reason;
END;

-- Start a transaction in default mode:
BEGIN;
SELECT * FROM tpcds.reason;
END;

-- Start a transaction with the isolation level being READ COMMITTED and the access mode being READ WRITE:
START TRANSACTION ISOLATION LEVEL READ COMMITTED READ WRITE;
SELECT * FROM tpcds.reason;
COMMIT;
```

## Helpful Links

[COMMIT | END, ROLLBACK, SET TRANSACTION](#)

# 15.11 ROLLBACK

## Function

Rolls back the current transaction and backs out all updates in the transaction.

**ROLLBACK** backs out of all changes that a transaction makes to a database if the transaction fails to be executed due to a fault.

## Precautions

If a **ROLLBACK** statement is executed out of a transaction, no error occurs, but a warning information is displayed.

## Syntax

```
ROLLBACK [ WORK | TRANSACTION ] ;
```

## Parameter Description

**WORK | TRANSACTION**

Optional keyword that more clearly illustrates the syntax.

## Examples

```
-- Start a transaction:
START TRANSACTION;

-- Back out all changes:
ROLLBACK;
```



## Helpful Links

[COMMIT | END](#)

# 15.12 RELEASE SAVEPOINT

## Function

**RELEASE SAVEPOINT** destroys a savepoint previously defined in the current transaction.

Destroying a savepoint makes it unavailable as a rollback point, but it has no other user visible behavior. It does not undo the effects of commands executed after the savepoint was established. To do that, use **ROLLBACK TO SAVEPOINT**. Destroying a savepoint when it is no longer needed allows the system to reclaim some resources earlier than transaction end.

**RELEASE SAVEPOINT** also destroys all savepoints that were established after the named savepoint was established.

## Precautions

- Specifying a savepoint name that was not previously defined causes an error.
- It is not possible to release a savepoint when the transaction is in an aborted state.
- If multiple savepoints have the same name, only the one that was most recently defined is released.

## Syntax

```
RELEASE [ SAVEPOINT ] savepoint_name;
```

## Parameter Description

### **savepoint\_name**

Specifies the name of the savepoint you want to destroy.

## Examples

```
-- Create a table:  
CREATE TABLE tpcds.table1(a int);  
  
-- Start a transaction:  
START TRANSACTION;  
  
-- Insert data:  
INSERT INTO tpcds.table1 VALUES (3);  
  
-- Establish a savepoint:  
SAVEPOINT my_savepoint;  
  
-- Insert data:  
INSERT INTO tpcds.table1 VALUES (4);  
  
-- Delete the savepoint:  
RELEASE SAVEPOINT my_savepoint;
```

```
-- Submit the transaction:  
COMMIT;  
  
-- Query the table content, which should contain both 3 and 4.  
SELECT * FROM tpods.table1;  
  
-- Delete the tables:  
DROP TABLE tpods.table1;
```

## Helpful Links

[SAVEPOINT](#), [ROLLBACK TO SAVEPOINT](#)

# 15.13 ROLLBACK PREPARED

## Function

**ROLLBACK PREPARED** cancels a transaction ready for two-phase committing.

## Precautions

- The function is only available in maintenance mode (when GUC parameter **xc\_maintenance\_mode** is **on**). Exercise caution when enabling the mode. It is used by maintenance engineers for troubleshooting. Common users should not use the mode.
- Only the user that initiates a transaction or the system administrator can roll back the transaction.
- The transaction function is maintained automatically by the database, and should be not visible to users.

## Syntax

```
ROLLBACK PREPARED transaction_id ;
```

## Parameter Description

**transaction\_id**

Specifies the identifier of the transaction to be submitted. The identifier must be different from those for current prepared transactions.

## Helpful Links

[COMMIT PREPARED](#), [PREPARE TRANSACTION](#)

# 15.14 ROLLBACK TO SAVEPOINT

## Function

**ROLLBACK TO SAVEPOINT** rolls back to a savepoint. It implicitly destroys all savepoints that were established after the named savepoint.

Rolls back all commands that were executed after the savepoint was established. The savepoint remains valid and can be rolled back to again later, if needed.

## Precautions

- Specifying a savepoint name that has not been established is an error.
- Cursors have somewhat non-transactional behavior with respect to savepoints. Any cursor that is opened inside a savepoint will be closed when the savepoint is rolled back. If a previously opened cursor is affected by a **FETCH** or **MOVE** command inside a savepoint that is later rolled back, the cursor remains at the position that **FETCH** left it pointing to (that is, the cursor motion caused by **FETCH** is not rolled back). Closing a cursor is not undone by rolling back, either. A cursor whose execution causes a transaction to abort is put in a cannot-execute state, so while the transaction can be restored using **ROLLBACK TO SAVEPOINT**, the cursor can no longer be used.
- Use **ROLLBACK TO SAVEPOINT** to roll back to a savepoint. Use **RELEASE SAVEPOINT** to destroy a savepoint but keep the effects of the commands executed after the savepoint was established.

## Syntax

```
ROLLBACK [ WORK | TRANSACTION ] TO [ SAVEPOINT ] savepoint_name;
```

## Parameter Description

*savepoint\_name*

Rolls back to a savepoint.

## Examples

```
-- Undo the effects of the commands executed after my_savepoint was established:
START TRANSACTION;
SAVEPOINT my_savepoint;
ROLLBACK TO SAVEPOINT my_savepoint;
-- Cursor positions are not affected by savepoint rollback.
DECLARE foo CURSOR FOR SELECT 1 UNION SELECT 2;
SAVEPOINT foo;
FETCH 1 FROM foo;
?column?
-----
1
ROLLBACK TO SAVEPOINT foo;
FETCH 1 FROM foo;
?column?
-----
2
RELEASE SAVEPOINT my_savepoint;
COMMIT;
```

## Helpful Links

[SAVEPOINT, RELEASE SAVEPOINT](#)

# 16 Appendix

---

## 16.1 GIN Indexes

### 16.1.1 Introduction

Generalized Inverted Index (GIN) is designed for handling cases where the items to be indexed are composite values, and the queries to be handled by the index need to search for element values that appear within the composite items. For example, the items could be documents, and the queries could be searches for documents containing specific words.

We use the word "item" to refer to a composite value that is to be indexed, and the word "key" to refer to an element value. GIN stores and searches for keys, not item values.

A GIN index stores a set of (key, posting list) key-value pairs, where a posting list is a set of row IDs in which the key occurs. The same row ID can appear in multiple posting lists, since an item can contain more than one key. Each key value is stored only once, so a GIN index is very compact for cases where the same key appears many times.

GIN is generalized in the sense that the GIN access method code does not need to know the specific operations that it accelerates. Instead, it uses custom strategies defined for particular data types. The strategy defines how keys are extracted from indexed items and query conditions, and how to determine whether a row that contains some of the key values in a query actually satisfies the query.

### 16.1.2 Scalability

The GIN interface has a high level of abstraction, requiring the access method implementer only to implement the semantics of the data type being accessed. The GIN layer itself takes care of concurrency, logging and searching the tree structure.

All it takes to get a GIN access method working is to implement multiple user-defined methods, which define the behavior of keys in the tree and the relationships between keys, indexed items, and indexable queries. In short, GIN combines extensibility with generality, code reuse, and a clean interface.

There are four methods that an operator class for GIN must provide:

- `int compare(Datum a, Datum b)`  
Compares two keys (not indexed items) and returns an integer less than zero, zero, or greater than zero, indicating whether the first key is less than, equal to, or greater than the second. Null keys are never passed to this function.
- `Datum *extractValue(Datum itemValue, int32 *nkeys, bool **nullFlags)`  
Returns a palloc'd array of keys given an item to be indexed. The number of returned keys must be stored into `*nkeys`. If any of the keys can be null, also palloc an array of `*nkeys` bool fields, store its address at `*nullFlags`, and set these null flags as needed. `*nullFlags` can be left NULL (its initial value) if all keys are non-null. The returned value can be NULL if the item contains no keys.
- `Datum *extractQuery(Datum query, int32 *nkeys, StrategyNumber n, bool **pmatch, Pointer **extra_data, bool **nullFlags, int32 *searchMode)`

Returns a palloc'd array of keys given a value to be queried; that is, `query` is the value on the right-hand side of an indexable operator whose left-hand side is the indexed column. `n` is the strategy number of the operator within the operator class. Often, `extractQuery` will need to consult `n` to determine the data type of `query` and the method it should use to extract key values. The number of returned keys must be stored into `*nkeys`. If any of the keys can be null, also palloc an array of `*nkeys` bool fields, store its address at `*nullFlags`, and set these null flags as needed. `*nullFlags` can be left NULL (its initial value) if all keys are non-null. The returned value can be NULL if the query contains no keys.

`searchMode` is an output argument that allows `extractQuery` to specify details about how the search will be done. If `*searchMode` is set to `GIN_SEARCH_MODE_DEFAULT` (which is the value it is initialized to before call), only items that match at least one of the returned keys are considered candidate matches. If `*searchMode` is set to `GIN_SEARCH_MODE_INCLUDE_EMPTY`, then in addition to items containing at least one matching key, items that contain no keys at all are considered candidate matches. (This mode is useful for implementing is-subset-of operators, for example.) If `*searchMode` is set to `GIN_SEARCH_MODE_ALL`, then all non-null items in the index are considered candidate matches, whether they match any of the returned keys or not.

`pmatch` is an output argument for use when partial match is supported. To use it, `extractQuery` must allocate an array of `*nkeys` Booleans and store its address at `*pmatch`. Each element of the array should be set to `TRUE` if the corresponding key requires partial match, `FALSE` if not. If `*pmatch` is set to `NULL` then GIN assumes partial match is not required. The variable is initialized to NULL before call, so this argument can simply be ignored by operator classes that do not support partial match.

`extra_data` is an output argument that allows `extractQuery` to pass additional data to the `consistent` and `comparePartial` methods. To use it, `extractQuery` must allocate an array of `*nkeys` pointers and store its address at `*extra_data`, then store whatever it wants to into the individual pointers. The variable is initialized to `NULL` before call, so this argument can simply be ignored by operator classes that do not require extra data. If `*extra_data` is set, the whole array is passed to the `consistent` method, and the appropriate element to the `comparePartial` method.

- `bool consistent(bool check[], StrategyNumber n, Datum query, int32 nkeys, Pointer extra_data[], bool *recheck, Datum queryKeys[], bool nullFlags[])`  
Returns **TRUE** if an indexed item satisfies the query operator with StrategyNumber **n** (or might satisfy it, if the `recheck` indication is returned). This function does not have direct access to the indexed item's value, since GIN does not store items explicitly. Rather, what is available is knowledge about which key values extracted from the query appear in a given indexed item. The check array has length **nkeys**, which is the same as the number of keys previously returned by **extractQuery** for this query datum. Each element of the check array is **TRUE** if the indexed item contains the corresponding query key, for example, if `(check[i] == TRUE)`, the *i*-th key of the `extractQuery` result array is present in the indexed item. The original query datum is passed in case the **consistent** method needs to consult it, and so are the **queryKeys[]** and **nullFlags[]** arrays previously returned by **extractQuery**. **extra\_data** is the extra-data array returned by **extractQuery**, or **NULL** if none.

When **extractQuery** returns a null key in **queryKeys[]**, the corresponding **check[]** element is **TRUE** if the indexed item contains a null key; that is, the semantics of **check[]** are like **IS NOT DISTINCT FROM**. The **consistent** function can examine the corresponding **nullFlags[]** element if it needs to tell the difference between a regular value match and a null match.

On success, **\*recheck** should be set to **TRUE** if the heap tuple needs to be rechecked against the query operator, or **FALSE** if the index test is exact. That is, a **FALSE** return value guarantees that the heap tuple does not match the query; a **TRUE** return value with **\*recheck** set to **FALSE** guarantees that the heap tuple does match the query; and a **TRUE** return value with **\*recheck** set to **TRUE** means that the heap tuple might match the query, so it needs to be fetched and rechecked by evaluating the query operator directly against the originally indexed item.

Optionally, an operator class for GIN can supply the following method:

- `int comparePartial(Datum partial_key, Datum key, StrategyNumber n, Pointer extra_data)`  
Compares a partial-match query key to an index key. Returns an integer whose sign indicates the result: less than zero means the index key does not match the query, but the index scan should continue; zero means that the index key matches the query; greater than zero indicates that the index scan should stop because no more matches are possible. The strategy number **n** of the operator that generated the partial match query is provided, in case its semantics are needed to determine when to end the scan. Also, **extra\_data** is the corresponding element of the extra-data array made by **extractQuery**, or **NULL** if none. Null keys are never passed to this function.

To support "partial match" queries, an operator class must provide the **comparePartial** method, and its **extractQuery** method must set the **pmatch** parameter when a partial-match query is encountered. For details, see [Partial Match Algorithm](#).

The actual data types of the various Datum values mentioned in this section vary depending on the operator class. The item values passed to **extractValue** are always of the operator class's input type, and all key values must be of the class's **STORAGE** type. The type of the query argument passed to **extractQuery**, **consistent** and **triConsistent** is whatever is specified as the right-hand input type

of the class member operator identified by the strategy number. This need not be the same as the item type, so long as key values of the correct type can be extracted from it.

### 16.1.3 Implementation

Internally, a GIN index contains a B-tree index constructed over keys, where each key is an element of one or more indexed items (a member of an array, for example) and where each tuple in a leaf page contains either a pointer to a B-tree of heap pointers (a "posting tree"), or a simple list of heap pointers (a "posting list") when the list is small enough to fit into a single index tuple along with the key value.

Multi-column GIN indexes are implemented by building a single B-tree over composite values (column number, key value). The key values for different columns can be of different types.

#### GIN Fast Update Technique

Updating a GIN index tends to be slow because of the intrinsic nature of inverted indexes: inserting or updating one heap row can cause many inserts into the index. After the table is vacuumed or if the pending list becomes larger than **work\_mem**, the entries are moved to the main GIN data structure using the same bulk insert techniques used during initial index creation. This greatly increases the GIN index update speed, even counting the additional vacuum overhead. Moreover the overhead work can be done by a background process instead of in foreground query processing.

The main disadvantage of this approach is that searches must scan the list of pending entries in addition to searching the regular index, and so a large list of pending entries will slow searches significantly. Another disadvantage is that, while most updates are fast, an update that causes the pending list to become "too large" will incur an immediate cleanup cycle and be much slower than other updates. Proper use of autovacuum can minimize both of these problems.

If consistent response time (of entity cleanup and of update) is more important than update speed, use of pending entries can be disabled by turning off the **fastupdate** storage parameter for a GIN index. For details, see the [CREATE INDEX](#).

#### Partial Match Algorithm

GIN can support "partial match" queries, in which the query does not determine an exact match for one or more keys, but the possible matches fall within a narrow range of key values (within the key sorting order determined by the **compare** support method). The **extractQuery** method, instead of returning a key value to be matched exactly, returns a key value that is the lower bound of the range to be searched, and sets the **pmatch** flag true. The key range is then scanned using the **comparePartial** method. **comparePartial** must return zero for a matching index key, less than zero for a non-match that is still within the range to be searched, or greater than zero if the index key is past the range that could match.

## 16.1.4 GIN Tips and Tricks

### Create vs. Insert

Insertion into a GIN index can be slow due to the likelihood of many keys being inserted for each item. So, for bulk insertions into a table, it is advisable to drop the GIN index and recreate it after finishing the bulk insertions. GUC parameters related to GIN index creation and query performance as follows:

- `maintenance_work_mem`  
Build time for a GIN index is very sensitive to the **`maintenance_work_mem`** setting.
- `work_mem`  
During a series of insertions into an existing GIN index that has **`fastupdate`** enabled, the system will clean up the pending-entry list whenever the list grows larger than **`work_mem`**. To avoid fluctuations in observed response time, it is desirable to have pending-list cleanup occur in the background (that is, via `autovacuum`). Foreground cleanup operations can be avoided by increasing **`work_mem`** or making **`autovacuum`** more aggressive. However, if **`work_mem`** is increased, a foreground cleanup (if any) will take a longer time.
- `gin_fuzzy_search_limit`  
The primary goal of developing GIN indexes is to create support for highly scalable full-text search in GaussDB(DWS). However, a very large set of results may be returned by a full-text query for words that frequently occur. In addition, reading many tuples from the disk and sorting them will consume large numbers of resources, which is unacceptable for production.  
To facilitate controlled execution of such queries, GIN has a configurable soft upper limit on the number of rows returned: the **`gin_fuzzy_search_limit`** configuration parameter. It is set to 0 (meaning no limit) by default. If a non-zero limit is set, then the returned set is a subset of the whole result set, chosen at random.

## 16.2 Extended Functions

For the extended data, GaussDB(DWS) provides corresponding operation functions. The operation function depends on the extension data type. Currently, in the Beta stage, functions may not stable or versions may later change. Use these functions with caution in commercial service scenarios. Otherwise, the service upgrade and scale-out may be affected.

The following table lists the extended functions supported by GaussDB(DWS) and they are for reference only.

Type	Name	Description
Access privilege inquiry function	<code>has_sequence_privilege(user, sequence, privilege)</code>	Queries whether a specified user has privilege for sequences.
	<code>has_sequence_privilege(sequence, privilege)</code>	Queries whether the current user has privilege for sequences.



Type	Name	Description
Trigger function	pg_get_triggerdef(trigger_oid)	Gets <b>CREATE [ CONSTRAINT ] TRIGGER</b> command for triggers.
	pg_get_triggerdef(trigger_oid, pretty_bool)	Gets <b>CREATE [ CONSTRAINT ] TRIGGER</b> command for triggers.

## 16.3 Extended Syntax

GaussDB(DWS) provides some extended syntax. Currently, they are in the Beta phase and are for internal use only.

The following table lists the extended syntax supported by GaussDB(DWS) and they are for reference only.

**Table 16-1** Extended SQL syntax

Category	Keywords	Description
Creating a table (CREATE TABLE)	[ WITH ( {storage_parameter = value} [, ... ] )   <b>WITH OIDS</b>   <b>WITHOUT OIDS</b> ]	Specifies whether the attribute OIDS can be specified when you create a table.
	<b>INHERITS ( parent_table [, ... ] )</b>	Specifies whether an inherited table is supported.
	<b>DISTRIBUTE BY { REPLICATION   <b>ROUNDROBIN</b>   { [HASH]   <b>MODULO</b> } ( column_name ) }</b>	When this switch is turned on, the local table supports roundrobin and modulo distribution modes.
	<b>TO { <b>GROUP</b> groupname   <b>NODE</b> ( nodename [, ... ] ) }</b>	Specifies whether users can specify table data to the list of DNs to be distributed using TO NODE/ GROUP.
	column_constraint: <b>REFERENCES reftable [ ( refcolumn ) ] [ <b>MATCH FULL</b>   <b>MATCH PARTIAL</b>   <b>MATCH SIMPLE</b> ] [ <b>ON DELETE</b> action ] [ <b>ON UPDATE</b> action ]</b>	Specifies whether users can use REFERENCES reftable [ ( refcolumn ) ] [ <b>MATCH FULL</b>   <b>MATCH PARTIAL</b>   <b>MATCH SIMPLE</b> ] [ <b>ON DELETE</b> action ] [ <b>ON UPDATE</b> action ] to create a foreign key constraint for a table.

Category	Keywords	Description
	table_constraint: <b>EXCLUDE [ USING index_method ] ( exclude_element WITH operator [, ... ] )</b> index_parameters [ WHERE ( predicate ) ]   <b>FOREIGN KEY ( column_name [, ... ] ) REFERENCES reftable [ ( refcolumn [, ... ] ) ] [ MATCH FULL   MATCH PARTIAL   MATCH SIMPLE ] [ ON DELETE action ] [ ON UPDATE action ]</b>	Specifies that users cannot use <b>EXCLUDE [ USING index_method ] (exclude_element WITH operator [,... ])</b> to create an excluding constraint for a table.
Altering a table (ALTER TABLE)	<b>TO { GROUP groupname   NODE ( nodename [, ... ] ) }</b>	Modifies the list of DNs where table data is distributed.
	<b>DELETE NODE ( nodename [, ... ] )</b>	Deletes the DNs on which table data is distributed.
Loading a module	<b>CREATE EXTENSION</b>	Loads a new module (such as DBLINK) to the current database.
	<b>ALTER EXTENSION</b>	Modifies the loaded module.
	<b>DROP EXTENSION</b>	Deletes the loaded module.
Aggregate functions	<b>CREATE AGGREGATE</b>	Defines a new aggregation function.
	<b>ALTER AGGREGATE</b>	Modifies the definition of an aggregate function.
	<b>DROP AGGREGATE</b>	Drops an existing function.
Operators	<b>CREATE OPERATOR</b>	Defines a new operator.
	<b>ALTER OPERATOR</b>	Modifies the definition of the operator.
	<b>DROP OPERATOR</b>	Drops an existing operator from the database.
Operator classes	<b>CREATE OPERATOR CLASS</b>	Defines a new operator class.
	<b>ALTER OPERATOR CLASS</b>	Modifies the definition of an operator class.
	<b>DROP OPERATOR CLASS</b>	Drops an existing operator family.
Operator families	<b>CREATE OPERATOR FAMILY</b>	Defines a new operation family.

Category	Keywords	Description
	<b>ALTER OPERATOR FAMILY</b>	Modifies the definition of an operator family.
	<b>DROP OPERATOR FAMILY</b>	Deletes an existing operator family.
Text search parsers	<b>CREATE TEXT SEARCH PARSER</b>	Creates a text retrieval parser.
	<b>ALTER TEXT SEARCH PARSER</b>	Modifies a text retrieval parser.
	<b>DROP TEXT SEARCH PARSER</b>	Deletes the existing text search parser.
Text search templates	<b>CREATE TEXT SEARCH TEMPLATE</b>	Creates a text search template.
	<b>ALTER TEXT SEARCH TEMPLATE</b>	Modifies the text search template.
	<b>DROP TEXT SEARCH TEMPLATE</b>	Deletes the existing text search template.
Collation rules	<b>CREATE COLLATION</b>	Creates a collation rule. The collation rule allows users to define data in the column-level, or even the collation rule and character class behaviors at the operation level.
	<b>ALTER COLLATION</b>	Modifies the collation rule.
	<b>DROP COLLATION</b>	Deletes the collation rule.
Requirement	<b>CREATE RULE</b>	Creates a rule. A rule indicates that you are to execute some other actions when performing operations on a specified table.
	<b>DROP RULE</b>	Deletes a rule.
Generating a notification	<b>NOTIFY</b>	The <b>NOTIFY</b> command sends a notification together with an optional "payload" string to each client that has previously executed <b>LISTEN</b> for a specified channel in the current database.
Listening for a notification	<b>LISTEN</b>	Registers a listener for the current session.

Category	Keywords	Description
Stopping listening for a notification	<b>UNLISTEN</b>	Clears all listeners of this session registration.
Loading or reloading a shared library file	<b>LOAD</b>	Loads a shared library file to the address space of the database server.
Releasing session resources in a database	<b>DISCARD</b>	Releases session resources in a database.
Procedural languages	<b>CREATE LANGUAGE</b>	Registers a new language.
	<b>ALTER LANGUAGE</b>	Modifies the definition of a procedural language.
	<b>DROP LANGUAGE</b>	Deletes a procedural language.
Domain	<b>CREATE DOMAIN</b>	Create a domain.
	<b>ALTER DOMAIN</b>	Modifies the definition of the existing domain.
	<b>DROP DOMAIN</b>	Deletes a domain.
Coding conversion	<b>CREATE CONVERSION</b>	Defines the character set conversion.
	<b>ALTER CONVERSION</b>	Modifies the definition of code conversion.
	<b>DROP CONVERSION</b>	Deletes a previously defined code conversion.
Type conversion	<b>CREATE CAST</b>	Defines a new type conversion. This conversion is used as an example to describe how to convert between two types.
	<b>DROP CAST</b>	Deletes a previously defined type conversion.

Category	Keywords	Description
Creating a cursor	CURSOR name [ BINARY ] [ <b>INSENSITIVE</b> ] [ [ NO ] <b>SCROLL</b> ] [ <b>WITH HOLD</b> ] FOR query	<p><b>INSENSITIVE</b>:                      The keyword is used only for being compatible with the SQL standard.</p> <p><b>SCROLL</b>: declares that the cursor can be used for reverse search.</p> <p><b>WITH HOLD</b> indicates that a cursor can still be used after the transaction creating the cursor is successfully submitted.</p>
Moving a cursor	<b>MOVE BACKWARD</b>	A reverse mobile cursor can be used only when it is used together with <b>SCROLL</b> .