AgensGraph Quick Guide

Information of technical documentation

Title : AgensGraph Quick Guide
Published date : March 22, 2018
S/W version : AgensGraph v1.3
Technical documentation version : v1.0

Contents

About technical documentation

Objective of technical documentation

This technical article is described for application program developers who wish to develop programs using various application libraries provided by AgensGraph® (hereinafter referred to as ”AgensGraph”), a database administrator (hereinafter referred to as ”DBA”) who wants to create a database and ensure the operation of AgensGraph and all database users who want to use AgensGraph to perform database operations or to refer to Hybrid SQL for application program creation.

Prerequisites for technical documentation

In order to fully understand this article, you should be familiar with the following topics:

- Graph Database
- Relational Database
- Basic Programming
- Basic API
- Basic knowledge of UNIX series (including Linux)

Limitations of technical documentation

This guide does not contain everything you need to apply or operate AgensGraph in practice. Therefore, refer to each technical manual for operation and management such as installation and environment setting.

Conventions of Technical Documentation

<table>
<tr><th>Mark</th><th>Description</th></tr>
<tr><td><AaBbCc123></td><td>The file name of the program source code, directory</td></tr>
<tr><td>[Button]</td><td>Button or menu name in GUI</td></tr>
<tr><td>Bold</td><td>Emphasis</td></tr>
<tr><td>“ ”(Quotes)</td><td>Refer to other relevant guides or other chapters and sections within the guide</td></tr>
<tr><td>‘Input field’</td><td>A description of the entries in the UI</td></tr>
<tr><td>Hyperlink</td><td>Email account, Website</td></tr>
<tr><td>></td><td>Progress of menu</td></tr>
<tr><td>+—</td><td>Has subdirectory or file</td></tr>
<tr><td>|—</td><td>No subdirectories or files</td></tr>
<tr><td>Notes</td><td>Notes or cautions</td></tr>
<tr><td>[Figure 1.1]</td><td>Figure name</td></tr>
<tr><td>[Table 1.1]</td><td>Table name</td></tr>
<tr><td>AaBbCc123</td><td>Commands, output after execution, sample code</td></tr>
<tr><td>{}</td><td>Required Argument Values</td></tr>
<tr><td>[]</td><td>Optional Argument Value</td></tr>
<tr><td>|</td><td>Selection Argument Value</td></tr>
</table>

Locations

Korea

Bitnine Inc.

A1201 GangSeo Hangang Xi Tower 401,

Yangcheon-ro, Gangseo-gu, Seoul,

South Korea

Tel : +82-70-4800-3517

Fax : +82-70-8677-2552

Email : agens@bitnine.net

Web : bitnine.net

USA

Bitnine Global Inc.

3945 Freedom Cir., Suite 260,

Santa Clara, CA 95054

U.S.A

Tel : +1 (408) 352-5165

Email : agens@bitnine.net

Web : bitnine.net

1 Release Notes

This document is for AgensGraph v1.3 community edition. See the release link.

2 Overview

2.1 Basic information

2.1.1 About the Documentation

This document is a short guide for AgensGraph developers. This guide will introduce various aspects of AgensGraph, including the data model and data definition language, Cypher query processing abilities, and Java development capabilities. Because AgensGraph is compatible with PostgreSQL, some parts of the documentation quote parts of PostgreSQL's documentation or link to it.

2.1.2 License

2.2 AgensGraph Introduction

2.2.1 What is AgensGraph?

AgensGraph is a new generation multi-model graph database for the modern complex data environment. AgensGraph is a multi-model database based on PostgreSQL RDBMS, and supports both relational and graph data models at the same time. This enables developers to integrate the legacy relational data model and the flexible graph data model in one database. AgensGraph supports Ansi-SQL and Open Cypher (http://www.opencypher.org). SQL queries and Cypher queries can be integrated into a single query in AgensGraph.
AgensGraph is very robust, fully-featured and ready for enterprise use. It is optimized for handling complex connected graph data and provides plenty of powerful database features essential to the enterprise database environment including ACID transactions, multi-version concurrency control, stored procedure, triggers, constraints, sophisticated monitoring and a flexible data model (JSON). Moreover, AgensGraph leverages the rich eco-systems of PostgreSQL and can be extended with many outstanding external modules, like PostGIS.

2.3 Features

- Multi-model support of the property graph data model, relational data model and JSON documents
- Cypher query language support
- Integrated querying using SQL and Cypher
- Graph data object management
- Hierarchical graph label organization
- Property indexes on both vertexes and edges
- Constraints: unique, mandatory and check constraints
- ACID transactions
- Hadoop connectivity
- Active-standby replication
- And many other features

2.3.1 More Information

Because AgensGraph inherits all features of PostgreSQL (http://postgresql.org), users and database administrators can refer to the documentation of PostgreSQL for more information.

3 Installation

AgensGraph runs on Linux and Windows. There are two methods available to install AgensGraph: downloading the binary package or compiling the package from source code.

3.1 Installation of pre-built packages

3.1.1 Installing AgensGraph on Linux

1. Get the pre-compiled binary:

   Visit the AgensGraph download page and download the corresponding version of AgensGraph.

   Tip: If you do not know your system environment, you can use the command:

   `uname -sm`

2. Extract the package:

   Extract the downloaded file into a directory for your use (for example, /usr/local/AgensGraph/ on Linux)

   tar xvf */path/to/your/use*

   Note : If you want AgensGraph on other operating systems, please contact Bitnine's support team.

3.2 Installation of build source code

1. Access the AgensGraphgithub and get the source code.

   $ git clone https://github.com/bitnine-oss/agensgraph.git

2. Install the following essential libraries according to each OS.

   1. CENTOS

      $ yum install gcc glibc glib-common readline readline-devel zlib zlib-devel

   2. Fedora

      $ dnf install gcc glibc bison flex readline readline-devel zlib zlib-devel

   3. Ubuntu

      $ sudo apt-get install build-essential libreadline-dev zlib1g-dev flex bison

3. Go to the clone location and run configure on the source tree. The --prefix=/path/to/intall option allows you to set the location where AgensGraph will be installed.

   $ ./congifure

4. Run the build.

   $ make install

5. Install the extension module and binary

   $ make install-world

3.3 Post-Installation Setup and Configuration

1. Environment variable setting (optional):

   You can add these commands into a shell start-up file, such as the ~/.bash_profile.

   ```
   export LD_LIBRARY_PATH=/usr/local/AgensGraph/lib:$LD_LIBRARY_PATH
   export PATH=/usr/local/AgensGraph/bin:$PATH
   export AGDATA=/path/to/make/db_cluster
   ```

2. Creating a database cluster*:

   ```
   initdb [-D /path/to/make/db_cluster]
   ```

3. Starting the server**:

   ```
   ag_ctl start [-D /path/created/by/initdb]
   ```

4. Creating a database**:

   ```
   createdb [dbname]
   ```

   If dbname is not specified, a database with the same name as the current user is created, by default.

5. Execute the interactive terminal:

   ```
   agens [dbname]
   ```

   If the db_cluster directory is not specified with `-D` option, the environment variable `AGDATA` is used.

3.4 Configuring Server Parameters

In order to attain optimal performance, it is very important to set server parameters correctly according to the size of data and machine resources. Among many server parameters, the following parameters are crucial for AgensGraph graph query performance. (You can edit `$AGDATA/postgresql.conf` file to set these parameters (restart required)).

- shared_buffers: The size of memory for caching data objects. This parameter should be increased for the production environment. It is optimal when it is as large as the data size. But, this parameter should be set carefully considering concurrent sessions and memory size allocated for each queries. The recommended setting is half of the physical memory size.
- work_mem: This shoud be also increased according to the size of physical memory and the properties of queries that will be executed carefully.
- random_page_cost: This parameter is for query optimization. For graph queries, it is recommended to reduce this value to 1 or 0.005 (in case graph data is fully cached in memory).

For more information, you can refer to PostgreSQL documentation.

4 Data Model

4.1 AgensGraph Data Model

AgensGraph is a multi-model database. AgensGraph simultaneously supports both the property graph model and the relational model.

4.1.1 Property Graph Model

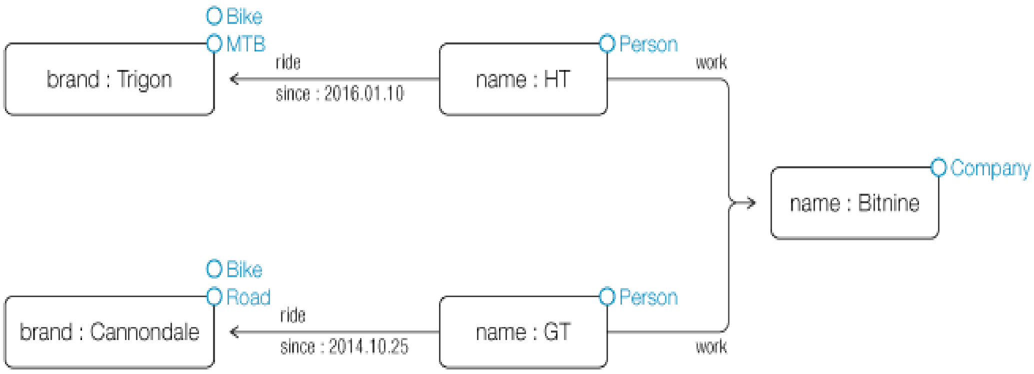

Figure 1.1: Labeled Property Graph Model

The property graph model contains connected entities, which can have any number of attributes. In AgensGraph, an entity is known as a vertex. Vertices can have an arbitrary number of attributes and can be categorized with labels. Labels are used to group vertices in order to represent some categories of vertices; i.e. representing the role of a person.

Edges are directed connections between two vertices. Edges can also have attributes and categorized labels like vertices. In AgensGraph, an edge always has a start vertex and an end vertex. If a query tries to delete a vertex, it must delete all its edges first. Broken edges cannot exist in AgensGraph.

Properties of edges and vertices are represented in the JSON format. JSON is a text format for the serialization of semi-structured data. JSONs are comprised of six data types: strings, numbers, booleans, null, objects and arrays. AgensGraph objects take full advantage of the JSON format by storing information as unordered collections of zero or more name/value pairs. A name is a string and a value can be any aforementioned type, including nested JSON types. AgensGraph specifically uses the JSONB format. Since JSONB is a decomposed binary format, it is processed much faster than regular JSON, but at the cost of a slightly slower input time.

4.1.2 Data Objects in AgensGraph

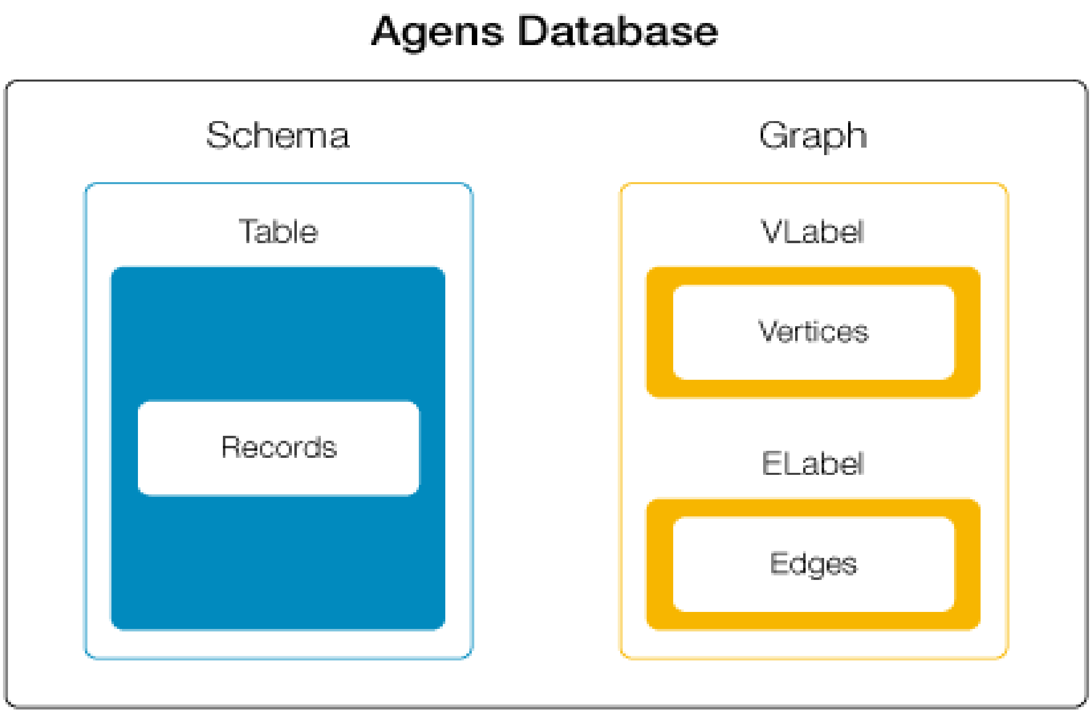

Figure 1.2: AgensGraph Simple Data model

In AgensGraph, several databases can be created and each database can contain one or more schemas and graphs. Schemas are for relational tables, and graph objects are for graph data. Schema name and graph name can not be the same. Vertices and edges are grouped into labels. There are two kinds of labels: vertex labels and edge labels. Users can create several graphs in a database but only one graph could be used at one time.

4.1.3 Labels

Labels are used to group vertices and edges. Users can create property indexes for all vertices under a given label. Labels can be used to provide access controls for different types of users, and label hierarchies can be created to add inheritance to labels. There is default labels for vertices: ag_vertex. If one creates a vertex without specifying its label, then the vertex is stored in the default label. While An edge always has one label.
We call vertex label and edge label as VLABEL and ELABEL respectively.

- VLABEL : The vertex label. Categorizes vertices to represent their roles.
  - Vertex : entities which can hold attributes.
- ELABEL : The edge label. Categorizes edges to represent their roles.
  - Edge : relationships connecting entities.

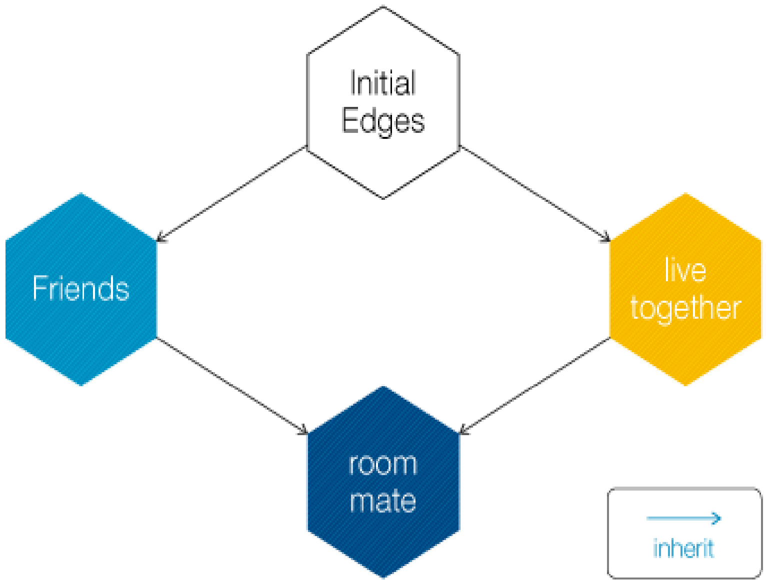

Figure 1.3: Edge Label inheritance example

Every label inherits one or more labels. The figure above shows an example hierarchy of edge labels. The label hierachy is similar to a class hierarchy in object-oriented programming. Each parent label contains child label data. For example, given the above hierarchy, if a query matches with the edge `friends' then, the results contain the data of the `roommate' label.

4.2 Data Definition Language

This section introduces DDLs for graph objects with a few examples.

4.2.1 Quick Description

Graph

To create a graph, use the CREATE GRAPH command.

CREATE

```
CREATE GRAPH graphname;
```

Several graphs can be created in a database. In order to specify which graph is to be used, the session parameter *graph_path* is used.

To show the current graph path, use the following command.

```
SHOW graph_path;
```

When a graph is created using `CREATE GRAPH`, *graph_path* is set to the created graph if *graph_path* is not set. You can create multiple graphs and change *graph_path* to another graph using the following command:

```
SET graph_path = graphname;
```

The *graph_path* is a session variable, so every client must set the *graph_path* before querying the graph. Only one graph name can be specified for *graph_path*. Querying over multiple graphs is not allowed.
If you set the *graph_path* for each user or database using the `ALTER ROLE` or `DATABASE` statement, you do not need to run the `SET graph_path` statement whenever you connect the database.

```
ALTER ROLE user1 IN DATABASE gdb SET graph_path TO graphname;
ALTER DATABASE gdb SET graph_path TO graphname;
```

DROP

```
DROP GRAPH graphname CASCADE;
```

A graph has initial labels for vertices and edges. These labels cannot be deleted. To drop the graph, users must do so with the `CASCADE` option. If current *graph_path* is the deleted graph, then *graph_path* is reset to null.

Labels

CREATE

```
CREATE VLABEL person;
CREATE VLABEL friend inherits (person);

CREATE ELABEL knows;
CREATE ELABEL live_together;
CREATE ELABEL room_mate inherits (knows, live_together);
```

The keywords VLABEL and ELABEL are used for identifying vertices and edges respectively. CREATE VLABEL will create a vertex label. VLABEL can inherit VLABEL only (in other words, VLABEL cannot inherit ELABEL). `inherits` is an option to inherit a parent label. If not specified, the system sets the initial label as a parent label. Multiple inheritance is possible for creating complex labels.
DROP

```
DROP VLABEL friend;
DROP VLABEL person;

DROP ELABEL knows CASCADE;
```

VLABEL friend inherits person, so VLABEL person cannot be dropped directly. VLABEL friend should be dropped first. In order to drop all child labels, the option CASCADE can be used to drop all dependencies.

4.2.2 Detailed Description

GRAPH

CREATE GRAPH

```
CREATE GRAPH [ IF NOT EXISTS ] graph_name [AUTHORIZATION role_name];
```

- IF NOT EXISTS
  - Do nothing if the same name already exists
- AUTHORIZATION role_name
  - The role name of the user who will own the new graph

ALTER GRAPH

```
ALTER GRAPH graph_name RENAME TO new_name;
ALTER GRAPH graph_name OWNER TO { new_owner | CURRENT_USER | SESSION_USER };
```

- graph_name
  - The name of an existing graph.
- RENAME TO new_name
  - This form changes the name of a graph to new_name.
- OWNER TO new_owner
  - This form changes the owner of the graph.

LABEL

CREATE LABEL

The synopsis of both VLABEL and ELABEL is identical.

```
CREATE [ UNLOGGED ] VLABEL [ IF NOT EXISTS ] label_name [DISABLE INDEX]
       [ INHERITS ( parent_label_name [, ...] ) ]
       [ WITH (storage_parameter)]
       [ TABLESPACE tablespace_name ];
```

- UNLOGGED
  - Data written to an unlogged label is not recorded to the write-ahead log, which makes unlogged labels considerably faster than logged labels. However, unlogged labels are not crash-safe.
- IF NOT EXISTS
  - Do nothing if the same name already exists.
- label_name
  - The name of the vertex/edge label to be created.
- DISABLE INDEX
  - Create label with invalid index. The invalid indexes can not be used for searching or inserting until reindexed.
- INHERITS (parent_label [, ⋯])
  - The optional INHERITS clause specifies a list of vertex/edge labels. If it is empty, the new label inherits the initial label. Use of INHERITS creates a persistent edge between the new child label and its parent label(s). The data of the child label is included in scans of the parent(s) by default.
- TABLESPACE tablespace_name
  - The new label will be created in the tablespace whose name is tablespace_name.

ALTER LABEL

```
ALTER [ IF EXISTS ] VLABEL label_name RENAME TO new_name;
ALTER [ IF EXISTS ] VLABEL label_name OWNER TO new_owner;
ALTER [ IF EXISTS ] VLABEL label_name SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN };
ALTER [ IF EXISTS ] VLABEL label_name SET TABLESPACE new_tablespace;
ALTER [ IF EXISTS ] VLABEL label_name CLUSTER ON idxname;
ALTER [ IF EXISTS ] VLABEL label_name SET WITHOUT CLUSTER;
ALTER [ IF EXISTS ] VLABEL label_name SET LOGGED;
ALTER [ IF EXISTS ] VLABEL label_name SET UNLOGGED;
ALTER [ IF EXISTS ] VLABEL label_name INHERIT parent_label;
ALTER [ IF EXISTS ] VLABEL label_name NO INHERIT parent_label;
ALTER [ IF EXISTS ] VLABEL label_name DISABLE INDEX;
```

- IF EXISTS
  - Do not throw an error if the label does not exist.

- label_name
  - The name of an existing vertex/edge label.
- RENAME TO new_name
  - Changes the name of a label to new_name.
- OWNER TO new_owner
  - Changes the owner of the label.
- SET STORAGE
  - This form sets the storage mode for property. This controls whether property is held inline or in a secondary TOAST table, and whether the data should be compressed or not.
  - PLAIN must be used for fixed-length, inline and uncompressed values.
  - MAIN is inline and compressed.
  - EXTERNAL is external and uncompressed.
  - EXTENDED is external and compressed.
- SET TABLESPACE
  - This form changes the label's tablespace to the specified tablespace and moves the data files to the new tablespace. Indexes on the label are not moved.
- CLUSTER/WITHOUT CLUSTER
  - This form select/remove the default index for future cluster operation. See this.
- SET LOGGED/UNLOGGED
  - This form changes the label from unlogged to logged or vice-versa.
- INHERIT/NO INHERIT
  - This form adds/removes the target label to/from the parent label's children list.
- DISABLE INDEX
  - This form changes all indexes of the label to invalid index. The invalid indexes can not be used for searching or inserting until reindexed.

Property Index

The property index provides a method to build indexes on the property values. Users can also create an index for an expression.

CREATE INDEX

```
CREATE [UNIQUE] PROPERTY INDEX [CONCURRENTLY] [IF NOT EXIST] indexname
ON labelname [USING method]
( attribute_expr | (expr) [COLLATE collation] [opclass] [ASC | DESC]
[NULLS {FIRST | LAST}]  )
[WITH] (storage_parameter = value [,...])
[TABLESPACE tablespacename]
[WHERE predicate];
```

- UNIQUE
  - Causes the system to check for duplicate values in the table when the index is created if the data already exists and each time data is added. Attempts to insert or update data which would result in duplicate entries will generate an error.
- CONCURRENTLY
  - With this option, the index will be built without taking any locks that prevent concurrent CREATE, DELETE(DETACH), SET on the label. There are several caveats to be aware of when using this option. See this.
- IF NOT EXIST
  - Do nothing if the same name already exists
- indexname
  - The name of the index to be created. If it is omitted, AgensGraph chooses an adequate name.
- labelname
  - The name of the label to be indexed.
- USING METHOD
  - The name of the index method to be used. They are btree, hash, gist, spgist, gin and brin. The default method is btree.
- attribute_expr
  - An expression which is pointing the attribute of property.
- COLLATE collation
  - The name of the collation to use for the index. By default, the index uses the collation declared for the column to be indexed or the result collation of the expression to be indexed. Indexes with non-default collations can be useful for queries that involve expressions using non-default collations.

- opclass
    - The name of an operator class.
- ASC | DESC
    - Specified ascending/descending sort order. ASC is the default.
- NULLS { FIRST | LAST }
    - Specifies that nulls sort before/after non-nulls. LAST is default when ASC is specified.
- WITH storage_parameter
    - The name of an index-method-specific storage parameter. See Index Storage Parameters for details.
- TABLESPACE tablespacename
    - The tablespace in which to create the index.
- WHERE predicate
    - The constraint expression for a partial index.

DROP INDEX

```
DROP PROPERTY INDEX indexname [CASCADE | RESTRICT];
```

- indexname
    - The indexname can be checked with `ag_property_indexes` or `\dGi`.
- CASCADE
    - Automatically drop objects that depend on the graph.
- RESTRICT
    - Refuse to drop the index if any objects depend on it. This is the default.

Constraints

This section provides information about how to use constraints on properties. Users can create two types of constraints; UNIQUE constraint and CHECK constraint.

```
CREATE CONSTRAINT constraint_name ON label_name ASSERT field_expr IS UNIQUE;
CREATE CONSTRAINT constraint_name ON label_name ASSERT check_expr;
DROP CONSTRAINT constraint_name ON label_name;
```

- field_expr

    - This form represents a JSON expression. This JSON can be a nested JSON.
- UNIQUE
    - This form specified that json_expression can contain only unique values.
- check_expr
    - The check expression returns a boolean result for new or updated properties. The result must be TRUE or UNKNOWN for an insert or update to succeed. Should any property of an insert or update operation produce a FALSE result, an error exception is raised and the insert or update will rolled back.

5 AgensGraph Query

5.1 Graph Query

5.1.1 Introduction

To retrieve and manipulate graph data, AgensGraph supports the Cypher query language. Cypher is a declarative language similar to SQL. Cypher is easy to learn since its syntax visually describes the patterns found in graphs.
This guide briefly explains how to write Cypher queries using an example graph.

5.1.2 Creating an Example Graph

AgensGraph can store multiple graphs in a single database. However, Cypher has no way of discerning multiple graphs. Therefore, AgensGraph supports additional Data Definition Languages and variables to create and manage graphs using Cypher.
The following statements create a graph called *network* and set it as a current graph.

```
CREATE GRAPH network;
SET graph_path = network;
```

In this example, the *graph_path* variable is explicitly set to *network*. However, if *graph_path* is not set before creating the graph, it will be set automatically after creating the graph.

Creating the Labels

Before creating graph data, generating a label is basic. Although this is the default, label is generated automatically when you specify label in the CREATE statement in cypher(VLABEL/ELABEL can both be created).

> note: Be careful not to confuse the label with the lable. Unintentional new labels can be created.

All graph elements have one label. For vertex, if no label is specified, have ag_vertex as a default label. For edge, the label can not be omitted. ag_edge label also exists but is used for other purposes.
AgensGraph supports DDL's to create such labels.
The following statements create a vertex label *person* and a edge label *knows*.

```
CREATE VLABEL person;
CREATE ELABEL knows;
CREATE (n:movie {title:'Matrix'});
```

Creating the Vertices and Edges

Now, we can create vertices for *person* and edges for *knows* by using Cypher's `CREATE` clause. The `CREATE` clause creates a pattern that consists of vertices and edges. A vertex has the form: `(variable:label {property: value, ...})`, and a edge has: `-[variable:label {property: value, ...}]-`. An additional < on the leftmost side or > on the rightmost side is used to denote the direction of the edge. `variable` can be omitted if created vertices and edges are not to be referenced.

> note: AgensGraph does not support `--` grammar for an edge in a pattern because `--` means a comment to the end of the line.

The following statements create three simple patterns: ``Tom knows Summer'', ``Pat knows Nikki'' and ``Olive knows Todd''.

```
CREATE (:person {name: 'Tom'})-[:knows {fromdate:'2011-11-24'}]->(:person {name: 'Summer'});
CREATE (:person {name: 'Pat'})-[:knows {fromdate:'2013-12-25'}]->(:person {name: 'Nikki'});
CREATE (:person {name: 'Olive'})-[:knows {fromdate:'2015-01-26'}]->(:person {name: 'Todd'});
MATCH (p:Person {name: 'Tom'}),(k:Person{name: 'Pat'})
CREATE (p)-[:KNOWS {fromdate:'2017-02-27'} ]->(k);
```

To store properties of vertices and edges, AgensGraph uses PostgreSQL's *jsonb* type. Properties can have nested JSON objects as their values. Since AgensGraph uses PostgreSQL's type system, any data type supported by PostgreSQL can be stored into the properties of vertices and edges.

5.1.3 Querying the Graph

Let's retrieve the pattern we created above. Cypher has the `MATCH` clause to find a pattern in a graph.
The following statement finds the pattern, ``A person called Tom knows a person''.

```
MATCH (n:person {name: 'Tom'})-[:knows]->(m:person) RETURN n.name AS n, m.name AS m;
   n   |    m
-------+----------
 "Tom" | "Summer"
 "Tom" | "Pat"
(2 rows)
```

Since properties are of the *jsonb* type, we need methods to access their property values. PostgreSQL supports those methods through operators such as `->`, `->>`, `#>`, and `#>>`. If a user wants to access to the property *name* of vertex *m*, one can write `(m)->>name`. AgensGraph offers an alternate way to access these elements. AgensGraph uses the dot operator . and bracket operators [] on vertices and edges to access property values in JSON objects and elements in JSON arrays as shown above.

The RETURN clause returns variables and its properties as a result of the query. The result is a table which has multiple matched patterns in its rows.

Variable Length Edges

Let's consider a query that finds knows of `Tom' and knows of knows. We can use the UNION clause:

```
MATCH (p:person {name: 'Tom'})-[:knows]->(f:person)
RETURN f.name
UNION ALL
MATCH (p:person {name: 'Tom'})-[:knows]->()-[:knows]->(f:person)
RETURN f.name;
```

It can also be written as:

```
MATCH (p:person {name: 'Tom'})-[r:knows*1..2]->(f:person)
RETURN f.name, r[1].fromdate;
```

A query looking for vertices located after a variable length of edge-vertex paths is typical in graph databases. *1..2 used in the edge represents such a variable-length edge. Where 1 is the minimum length of the edge and 2 is the maximum length. If you do not specify a value, the default range values are 1 and infinity.

You can also use update clauses such as CREATE, SET, REMOVE and DELETE after MATCH clauses. In the next section, we will see how data in a graph can be modified and deleted.

5.1.4 Manipulating the Graph

You can set properties on vertices and edges using the SET clause. If you set a null value to a property, the property will be removed.

The following statement finds the given pattern and updates the property in the matched edge.

```
MATCH (:person {name: 'Tom'})-[r:knows]->(:person {name: 'Summer'})
SET r.since = '2009-01-08';
```

To delete vertices and edges in a graph, we can use the DELETE clause.

The following statement finds a vertex and deletes it.

```
MATCH (n:person {name: 'Pat'}) DETACH DELETE (n);
```

The above example actually uses the DETACH DELETE clause to delete vertices and edges attached to the vertices all at once.

The final shape of the graph *network* is as follows:

```
MATCH (n)-[r]->(m) RETURN n.name AS n, properties(r) AS r, m.name AS m;
    n    |                         r                         |    m
---------+---------------------------------------------------+----------
 "Tom"   | {"since": "2009-01-08", "fromdate": "2011-11-24"} | "Summer"
 "Olive" | {"fromdate": "2015-01-26"}                        | "Todd"
(2 rows)
```

5.1.5 MERGE

If you need to ensure that a pattern exists in the graph, you can use MERGE. It will try to find the pattern if the pattern exists in the graph, or else it creates the pattern if it does not exist. If the pattern exists, it is treated like a MATCH clause, otherwise it is treated like a CREATE clause. MERGE is a MATCH or CREATE of the entire pattern. This means that if any element of the pattern does NOT exist, AgensGraph will create the entire pattern.

The following statement guarantees that everyone's city exists in the graph.

```
CREATE VLABEL customer;
CREATE VLABEL city;

CREATE (:customer {name:'Tom', city:'santa clara'}),
       (:customer {name:'Summer ', city:'san jose'}),
       (:customer {name:'Pat', city:'santa clara'}),
       (:customer {name:'Nikki', city:'san jose'}),
       (:customer {name:'Olive', city:'san francisco'});

MATCH (a:customer)
MERGE (c:city {name:a.city});

MATCH (c:city) RETURN properties(c);
       properties
-------------------------
 {"name": "santa clara"}
 {"name": "san jose"}
 {"name": "san francisco"}
(3 rows)
```

MERGE can perform SET depending on whether the pattern is MATCHed or CREATed. If it is MATCHed, it will execute ON MATCH SET clause. If it is CREATE-ed, it will execute ON CREATE SET clause.

```
CREATE (:customer {name:'Todd', city:'palo alto'});

MATCH (a:customer)
MERGE (c:city {name:a.city})
    ON MATCH SET c.matched = 'true'
    ON CREATE SET c.created = 'true';

MATCH (c:city) RETURN properties(c);
                  properties
----------------------------------------------
 {"name": "santa clara", "matched": "true"}
 {"name": "san jose", "matched": "true"}
 {"name": "san francisco", "matched": "true"}
 {"name": "palo alto", "created": "true"}
(4 rows)
```

The default isolation level of the transaction is Read committed on AgensGraph. Therefore, if another transaction executes MERGE for the same pattern at the same time, two (or more) identical patterns can be created at the same time. To prevent this, you can execute a transaction at the serializable isolation level as in the example below. If an attempt is made to create the same pattern concurrently, one transaction fails with an error. If you retry a failed transaction, it will behave like MATCH instead of CREATE because of the pattern created by the succeeded transaction already exists.

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
MATCH (a:customer)
MERGE (c:city {name:a.city});
COMMIT;
```

5.1.6 Finding the Shortest Path

The `shortestpath` function can be used to find the shortest path between two vertices. If you want to find all paths, you can use the `allshortestpaths` function.

```
MATCH (p:person {name:'Tom'}), (f:person {name:'Olive'}) CREATE (p)-[:knows]->(f);
MATCH (p1:person {name: 'Tom'}), (p2:person {name: 'Todd'}),
       path=shortestpath((p1)-[:knows*1..5]->(p2)) RETURN path;
```

In this example, we create a `knows' edge path from `Tom' to `Pat'. To find the `knows' path from `Tom' to `Nikki', we can use the `shortestpath` function. The `shortestpath` function takes a pattern consisting of a start vertex, a edge and an end vertex. We can use a variable length edge expression in the edge to specify if we are looking for a certain degree of connection.

The query results are as follows.

```
[person[3.1]{"name": "Tom"},knows[4.5][3.1,3.5]{},person[3.5]{"name": "Olive"},knows[4.3][3.5,3.6]
{"fromdate": "2015-01-26"},person[3.6]{"name": "Todd"}]
```

5.2 Hybrid Query

5.2.1 Introduction

In this section, we will see how to use SQL and Cypher together in AgensGraph using the following example graph. Hybrid Query performs aggregation and statistical processing on table and column by using SQL query used in RDB,and the Cypher query used by GDB supports better data query than RDB's Join operation.

```
CREATE GRAPH bitnine;
CREATE VLABEL dev;
CREATE (:dev {name: 'someone', year: 2015});
CREATE (:dev {name: 'somebody', year: 2016});

CREATE TABLE history (year, event)
AS VALUES (1996, 'PostgreSQL'), (2016, 'AgensGraph');
```

Cypher in SQL

Since the result of a Cypher query is a relation, you can use a Cypher query in the FROM clause of SQL as if it is a sub-query. It is possible to use Cypher syntax inside the FROM clause to utilize dataset of vertex or edge stored in graph DB as data in SQL statement.

Syntex :

```
SELECT [column_name]
FROM ({table_name|SQLquery|CYPHERquery})
WHERE [column_name operator value];
```

Example :

```
SELECT n->>'name' as name FROM history, (MATCH (n:dev) RETURN n) as dev
where history.year > (n->>'year')::int;
  name
---------
 someone
(1 row)
```

SQL in Cypher

When querying the content of the graph DB with the cypher queries, you can use the match and where clause when you want to search using specific data of the RDB. However, the resulting dataset in the SQL queries must be configured to return a single row of results.

Syntex :

```
MATCH [table_name]
WHERE (column_name operator {value|SQLquery|CYPHERquery})
RETURN [column_name];
```

Example :

```
MATCH (n:dev)
WHERE n.year < to_jsonb((SELECT year FROM history WHERE event = 'AgensGraph'))
RETURN properties(n) AS n;

                 n
-----------------------------------
 {"name": "someone", "year": 2015}
(1 row)
```

6 Graph Data Import

This section introduces an example to import graph data from external foreign files.

1. Install the extension `file_fdw' necessary to use the AgensGraph foreign-data wrapper to interface with files on the server's filesystem.

```
CREATE EXTENSION file_fdw;
```

2. Create the data import server.

```
CREATE SERVER import_server FOREIGN DATA WRAPPER file_fdw;
```

3. Create the foreign table to receive data from a foreign file.

```
CREATE FOREIGN TABLE vlabel_profile (id graphid, properties text)
SERVER import_server
OPTIONS( FORMAT 'csv', HEADER 'false', FILENAME '/path/file.csv', delimiter E'\t');
```

```
CREATE FOREIGN TABLE elabel_profile (id graphid, start graphid, "end" graphid, properties text)
SERVER import_server
OPTIONS( FORMAT 'csv', HEADER 'false', FILENAME '/path/file.csv', delimiter E'\t');
```

This `properties text' item should be written differently depending on the situation. See the following example.

```
comments.csv


id|creationDate|locationIP|browserUsed|content|length       // This is the HEADER of CSV
2473901162497|1316237219961|77.240.75.197|Firefox|yes|3
2473901162498|1316225421570|213.180.31.8|Firefox|thanks|6
4123168604165|1345407771283|41.203.141.129|Firefox|LOL|3
```

In this case, the schema of foreign table should be as follows.

```
create foreign table comments_profile
(
   id int8,
   creationDate int8,
   locationIP varchar(80),
```

```
   browserUsed varchar(80),
   content varchar(2000),
   length int4
)
server import_server
options
(
 FORMAT 'csv',
 HEADER 'true',     //Indicates the presence of a csv header
 DELIMITER '|',
 NULL '',
 FILENAME '/path/to/comments.csv'
 );
```

4. Execute the import. The data must be cast as type JSONB, since vertices and edges in AgensGraph are stored in the JSONB format.

```
CREATE VLABEL test_vlabel;

LOAD FROM vlabel_profile AS profile_name
CREATE (a:test_vlabel =to_jsonb(row_to_json(profile_name)));
```

```
CREATE ELABEL test_elabel;

LOAD FROM elabel_profile AS profile_name
MATCH (a:test_vlabel), (b:test_vlabel)
WHERE (a).id = to_jsonb(profile_name).start AND (b).id = to_jsonb(profile_name).end
CREATE (a)-[:test_elabel]->(b);
```

Following the example above, you can do the following:

```
CREATE VLABEL comments;

LOAD FROM comments_profile AS ROW
CREATE (:comments =to_jsonb(row_to_json(row)));
```

6.0.1 Indexes with Data Import

The cost of maintaining indexes during bulk insertion is very expensive. AgensGraph provides grammars to toggle indexes by disabling them temporarily and reindexing them later. The disabled indexes do not interfere with bulk inserts. After bulk data import, the option REINDEX LABEL can be used. It will take some time but it is much faster than bulk insert with valid indexes.

```
CREATE VLABEL test_vlabel DISABLE INDEX;
OR
CREATE VLABEL test_vlabel;

ALTER VLABEL test_vlabel DISABLE INDEX;

-- DATA IMPORT
REINDEX VLABEL test_vlabel;
```

7 Tools

7.1 Command Line Interface Tool

AgensGraph has a command line interface tool called *agens*. Users can query and maintain AgensGraph using *agens* efficiently.

> Note: *agens* is based on *psql* (the PostgreSQL interactive command line tool) so that all features of *psql* can be used in AgensGraph too.

7.1.1 Meta Command

```
\dG[+]  [PATTERN]      list graphs
\dGe[+]  [PATTERN]     list graph edge labels
\dGl[+]  [PATTERN]     list graph labels
\dGv[+]  [PATTERN]     list graph vertex labels
\dGi[+]  [PATTERN]     list graph property indexes
```

agens supports the above additional meta-commands for administration and scripting. You can obtain a list of graph or labels. If + is appended to the command, each object is listed with its associated permissions and description, if any. If *pattern* is specified, only objects whose name match the pattern are listed.

8 Client Drivers

8.1 Java Driver

8.1.1 Introduction

This section briefly explains how to process graph data through AgensGraph to Java application developers. AgensGraph's JDBC driver is based on the PostgreSQL JDBC Driver and offers a way for the Java application developer to access the Agensgraph database in their applications. The API of the AgensGraph Java Driver and Postgres JDBC Driver are very similar. The only difference is that AgensGraph uses the Cypher query language instead of SQL and utilizes graph data (vertices, edges and paths) as data types.

8.1.2 Usage of the Java Driver

Get the Driver

You can download the precompiled driver(jar) from Download link or use maven as follows:

```
<dependencies>
  ...
  <dependency>
    <groupId>net.bitnine</groupId>
    <artifactId>agensgraph-jdbc</artifactId>
    <version>1.4.1</version>
  </dependency>
  ...
</dependencies>
```

You can search the latest version on The Central Repository with GroupId and ArtifactId.

Connection

To connect to AgensGraph using the Java Driver, we need two things: the name of class to be loaded into the Java Driver and the connection string.

- The name of class is `net.bitnine.agensgraph.Driver`.
- The connection string consists of the sub-protocol, server, port, database.

- `jdbc:agensgraph://`is the sub-protocol to use a particular Driver and a hold value.
- It is written as `jdbc:agensgraph://server:port/database`, including the sub-protocol.

The following code is an example of how to connect AgensGraph. It connects to AgensGraph through the `Connection` object and is ready to be queried.

```
import java.sql.DriverManager;
import java.sql.Connection;

public class AgensGraphTest {
  public static void main(String[] args) {
  try{
    Class.forName("net.bitnine.agensgraph.Driver");
    String connectionString = "jdbc:agensgraph://127.0.0.1:5432/agens";
    String username = "test";
    String password = "test";
    Connection conn = DriverManager.getConnection(connectionString, username, password);
    ...
  } catch (Exception e) {
          e.printStackTrace();
  }
  }
}
```

Retrieving Data

This example illustrates how to export Graph data using MATCH.

A Cypher query is executed using `executeQuery()`. The output is a `ResultSet` object, which is the same output format found in the JDBC driver. The output of the query is the `vertex' type of AgensGraph. Java Driver returns this output as a Vertex instance. Because the Vertex class is a sub-class of `Jsonb`, users can obtain information from the property fields.

```
import java.sql.DriverManager;
import java.sql.Connection;
import java.sql.Statement;
import java.sql.ResultSet;
```

```
import net.bitnine.agensgraph.graph.Vertex;

public class AgensGraphTest {
  public static void main(String[] args) {
  try{
    ...
    Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery(
        "MATCH (:Person {name: 'John'})-[:knows]-(friend:Person)" +
        "RETURN friend");
    while (rs.next()) {
      Vertex friend = (Vertex)rs.getObject(1);
      System.out.println(friend.getString("name"));
      System.out.println(friend.getInt("age"));
    } catch (Exception e) {
            e.printStackTrace();
    }
  }
  }
}
```

Creating Data

The following example illustrates how to insert a vertex with a label Person into AgensGraph. Users can input a property of a vertex using strings in Cypher queries. They can also be bound after making a `Jsonb`.

```
import java.sql.DriverManager;
import java.sql.Connection;
import java.sql.PreparedStatement;

import net.bitnine.agensgraph.util.Jsonb;
import net.bitnine.agensgraph.util.JsonbUtil;

public class AgensGraphTest {
  public static void main(String[] args) {
```

```
    ...
    PreparedStatement pstmt = conn.prepareStatement("CREATE (:person ?)");
        Jsonb john = JsonbUtil.createObjectBuilder()
       .add("name", "John")
       .add("from", "USA")
       .add("age", 17)
       .build();
    pstmt.setObject(1, john);
    pstmt.execute();
  }
}
```

The following is generated as a string:

```
"CREATE (:Person {name: 'John', from: 'USA', age: 17})"
```

[Reference]

In JDBC, the question mark (?) is the placeholder for the positional parameters of a PreparedStatement. There are, however, a number of SQL operators that contain a question mark. To avoid confusion, when used in a prepared statement, it must be used with spaces.

9 External Modules

9.1 External Modules

9.1.1 Introduction

This section briefly explains external modules which are contained in AgensGraph. One of great things of AgensGraph is that there are plenty of great external modules. Any PostgreSQL external modules can be used in AgensGraph too. The external modules which are provided by AgensGraph will be maintained with version upgrades. Usually, these modules have to be set correctly. You can refer to the documentations provided by each external module.

9.1.2 Server Monitoring: pg_stats_info

Users can monitor AgensGraph using pg_stats_info. You can also use Tadpole for AgensGraph (http://bitnine.net/downloads/tadpole-agensgraph/) for real-time monitoring.

9.1.3 Plan Optimization: pg_hint_plan

Using pg_hint_plan, users can modify query plans to optimize the performance. pg_hint_plan can be used to Cypher plans.

9.1.4 Warm Cache: pg_prewarm

By loading graph objects (vertex, edge and indexes), one can obtain the best performace. Using pg_prewarm, users can specify graph objects to be loaded in memory.

9.1.5 Statistics: hyperloglog

Hyperloglog can be used to estimate the count of distinct values in tables.

9.1.6 Big Data: hadoopfdw

Using Hadoopfdw and `LOAD` clause, users can load data from external hadoop systems to AgensGraph graph object.