PL/SQL STARTER FRAMEWORK

Schema Design Guidelines

Table of Contents

Overview

This document covers conventions and guidelines surrounding the design of a physical data model and independent objects within it. It gives guidance on the selection and use of certain Oracle features, helping the reader avoid pitfalls and traps of using these features, or using them to solve the wrong problem.

Standards and guidelines for physical database installation and configuration, security, database object naming and PL/SQL programming are covered elsewhere.

It is assumed the reader has a basic understanding of relational database design, and Oracle Database features.

Database Design Guiding Principles

Start with a Solid Foundation

Start by attempting to capture all the entities, attributes, relationships and business rules.

Proceed to the physical model, adding indexes, surrogate keys, constraints, and audit columns. Strive for third normal form as the bedrock layer of your physical data model. If the bedrock model is too difficult to navigate and query, create views that hide all the inner joins. If the bedrock model can't be coaxed to perform, **then** add denormalized tables and materialized views.

If you start with a solid, clean data model, it is easy to build on top of it later when needed. But if you start at a higher, less granular level, with denormalized data, natural keys, repeating attributes, etc., it is really hard and costly to clean it up and simplify.

Do it right the first time.

Protect Your Data

One of the primary reasons that databases exist is to protect the integrity and accuracy of critical data.

Use integrity constraints. Eliminate redundancy. Hide natural keys behind surrogate keys. Secure your data.

Allow Your Database to Sing

Too many treat their database like a hole in the ground, only useful for dumping data. This is akin to purchasing a fine sports car, but never leaving 1st gear. Use the full capabilities of the database. Let it do what it does best. There is no need for an application to write its own mechanisms to store, sort, cache, search, validate, or protect the data. This is the job of the database. It has been coded and optimized by hundreds of software engineers. There is no need to attempt to replicate their efforts.

Of course, this is not carte blanche to use every new feature the database touts. Put new features through appropriate testing before adopting them.

Design to End Goals from Day One

Determine the performance, functionality, flexibility and data quality goals of your information system before beginning. These are not things that can be stapled onto the data model right before release. They must be the driving forces behind your design sessions from day one. Automated, round-trip, and agile development tools and processes are making late-stage changes easier, but it will still cost less to design properly the first time around.

If you could choose one phase of your project on which to expend extra time, it should be during information model design. Infuse the model with features that meet system goals from the start, not after trouble comes along in production.

Database Design Tools and Environment

In an ideal world, all database changes would begin with the data architect (usually the DB Engineer) for your project. These changes would be entered in the modeling tool, validated against existing standards, and then forward engineered into DDL scripts. The DDL scripts can then be checked into the source code control system, tested in development and moved into the next build.

When this method of changing the data model is achieved, we will be able to build data dictionary documentation from the modeling tool, compare data models from prior releases, generate data model change release notes, maintain a much cleaner information model, have up-to-date model diagrams, etc.

The author has been through many data modeling and database programming IDE tool evaluations. In terms of cost, robustness and functionality, every time Embarcadero's ER/Studio was the modeling tool of choice, and Allround Automation's PL/SQL Developer was the programming tool of choice. The software version tool has changed with every project. Currently the author is using Subversion and is quite happy with it.

Your shops may have different needs. So shop around and choose well.

Tables

This section assumes the reader is already experienced in relational database design. It aims to provide some guidance and best practices regarding table design and the use of certain Oracle-specific features.

No Generic Models

Generic models attempt to stick everything in one of three tables: the object table, the attribute table, and the associative table that ties them together. These models never perform in production, except on the most trivial of systems. Start with 3rd Normal Form (see Solid Foundation above). Avoid generic models.

Repeating Attributes

If you are designing a table that has repeating attributes, break the attributes off into a child table, otherwise known as an attributive table. Again, go for a solid foundation. Don't compromise or get lazy because you are pressed for time.

Don't create tables with NESTED TABLE or VARRAY columns. There are too many tools, including the JDBC and ODBC drivers, which don't yet support tables constructed in this manner. It also causes problems when doing remote operations over db links. If you have a need for a nested table column, create a regular attributive child table instead.

Reference Codes

Every system has scores of code sets, lists of fairly static codes, literals, names, translations and reference values. They are often referred to as lookup, type, or reference tables. They can be implemented as one table per code set, or as a single table. There are advantages and drawbacks to each.

With the former, you have many more "moving parts" to manage, most of them quite similar in appearance. This creates additional development, testing and maintenance work. However, this approach is more flexible in that you can add additional attributes to certain code sets as the system matures, without significant rework.

With the latter approach, you would create a surrogate key to uniquely identify each entry in the main table. Then create a unique key, perhaps on the code set name, and the code itself. In internationalized applications, this single table is further decomposed into at least one additional table to contain textual representations of the code in the various languages. This approach is less flexible, but is far easier to manage. Most database professionals do not recommend this approach.

Associations

In our products, you will frequently need tables that store allowed combinations of certain reference values. These are known as associative, intersection or cross-reference entities, tables that resolve many-to-many relationships. Create a surrogate key for the association table's PK. Use this number as the FK in child tables. Create a unique key between the IDs of the items in the association. Add typical audit columns (see below) if there is a need to track changes.

Audit Columns

There are many methods of auditing data changes. At its most basic level, each table should have a set of four , non-nullable metadata columns which record who originally created the row of data, and when; and another pair which record who last modified the row of data, and when. These are usually placed at the end of the table's column list, since they are of lesser importance and infrequently used. Unless your application's architecture does not support them, use Oracle's TIMESTAMP as the datatype for your "created" and "modified" columns. The author has traditionally named this columns: CRT_BY, CRT_DT, MOD_BY, MOD_DT

If you need more granular audit data, like a record of old and new values – changes over time – speak with your DB Engineer about incorporating the common Audit Tables, or making use of Oracle's built-in auditing features.

Partitioning

The three primary reasons for partitioning a table involve easier administration, increased availability and sometimes increased query performance.

Partitioning is most advantageous for the management and availability of really big tables (over 100M rows is a pretty good indication it is going to be a big table).

So, when designing a table that you suspect is in need of partitioning, ask yourself these questions before going down that road:

- Is the table monstrous and are there frequent, necessary full scans of the entire table?
- Is there a concurrency issue around full-scanning the table? (large user base using the same table at the same time)
- Is there a need to update, repair, or drop certain portions of the table, AND a requirement for high uptime?

If you answer Yes to any of these questions, then partitioning will probably help you. Otherwise, it is not needed.

Try to ensure that your indexes on the table are all locally partitioned. This makes certain maintenance operations go smoother with much less downtime, and no index rebuilding at all (which is not the case with global indexes on partitioned tables).

You should primarily use only range-based partitioning. List, hash and composite partitioning are needed only rarely.

Do not end your partitioning scheme on a table with the MAXVALUE keyword. Add new partitions as necessary. Never create a "catch-all" partition like MAXVALUE.

If you are on 11g or higher, use reference partitioning if partitioning parent-child tables and interval partitioning to get automatic partition extension.

Parallel Processing Capabilities

Oracle's built-in parallel processing capabilities are a wonderful thing when applied to the right problem. It is horrible and will kill your scalability if applied to the wrong problem. The worst performance problems in the author's career were traced back to the parallel engine being accidentally activated.

Parallel Query

The physical DBA shoudl ensure that every table and index has DEGREE and INSTANCES set to 1 (this means PARALLEL is off by default). The parallel query feature should not be used unless a query and table is specifically designed for it. Do not design for parallel query unless all other tuning options have been exhausted AND the query naturally lends itself to being split up AND your users can do without all or most of the host CPUs for a while. You must understand that parallel query was mainly intended for data warehouse-like environments, where there are powerful servers with lots of spare CPUs, massive tables, really long queries, and very few users. If you are on an OLTP system, with a long-running query that needs optimized, don't use PQ; instead, use AQ or a scheduled job to run your parallel query at a time when the system is relatively idle.

Parallel DDL

Parallel DDL operations, on the other hand, are generally wonderful. When DDL scripts are being run, it is usually during downtime for upgrades anyway. No users are on the box, and getting back online is critical. When performing massive table moves or index rebuilds, for example, the parallel engine can be just the trick to speed up required operations. Use this feature liberally for maintenance operations and data migrations.

Parallel DML

There are also parallel DML operations. But the need to speed up massive INSERTs, DELETEs or UPDATEs is typically infrequent. If you have a need to change a value for every row in a huge table, a parallel UPDATE is a good idea. A parallel DELETE could be the answer to your prayers if you ever have to delete several years' worth of data from a non-partitioned table. If you choose the alternate route, creating a new copy of that table, and only inserting the data you want to keep (then dropping the old table), the parallel engine would still be useful for the large INSERT.

Clustered Tables

We have no negative opinions about using clusters, we simply haven't had the business need. If any of your products ever become bound by concurrency issues (100's to 1000's of concurrent users), or you are looking to eke out a few microseconds in response time, you might consider clusters or single table hash clusters for frequently read, rarely modified tables. Clusters require great care in their sizing and loading of sorted data. If the size of the rows, or amount of the rows can change much, don't use clusters, as clusters have to be rebuilt when they are altered significantly.

Index-Organized Tables

Use index-organized tables (IOTs) for most small, static, lookup tables that are mainly composed of columns that make up the primary or unique key. This is especially true of associative entities that join the primary keys of two or more tables to form a list of allowable combinations. An exception would be a table that had many other columns, not part of the key, which are also frequently read or queried. Rather than dealing with the design challenges of the threshold, block size, leaf splitting, etc. just make it a regular heap table.

IOTs are generally better at delivering the performance benefits promised by clusters because they don't suffer from the same limitations, including adjusting themselves to accommodate new rows so that related rows are physically collocated together (and therefore faster to access and read).

IOTs can also save significant space if the majority of the columns in a large table are also found in the large unique or composite index on the table.

If optimal space and speed is a concern, determine the appropriate PCTTHRESHOLD using tests and ANALYZE INDEX VALIDATE STRUCTURE as a starting point with its recommended optimal settings.

External Tables

External tables and directories are the preferred approach to loading incoming delimited files. Oracle Streams should also be evaluated as an approach for data loads. External tables work best when the file that defines the external table retains the same name for every load. Otherwise you have to create code dynamically to recreate the external table every time you get a new file. Perhaps soft links on the Unix side could be used to abstract the changing file names.

Compression

Do not use table compression except where a specific business need requires it. In our tests, table compression slowed performance down unacceptably. Table compression may be acceptable to save space consumed by old, rarely accessed data that needs to remain online.

However, index compression seems to work like a champ from our experience. In one scenario on a really large table, compressing a 17 column composite UK, used much less space and actually sped queries a bit. If you have an index that is composed of many repeating values with a couple columns at the end differentiating each entry, consider using index compression to factor out the common columns and save a great deal of space.

Effective Dating

Certain applications require that records in a table be versioned, such as prices for a company's products. The prices can change based on season, special promotions, etc. A given price would only be effective for a certain date range for product X, after which another price would take effect. Data versioning is usually accomplished by means of adding First Effective Date and Last Effective Date columns to the table. Then it becomes a simple matter of updating any DML on the table to touch only those records that match a given date.

Effective dating seems harmless to the uninitiated. And it is not too bad if the effective dating schemes remain fixed and never change. But it quickly becomes a complex tangle when the business needs to rearrange past or future records, inserting new records into the middle of an existing range scheme, adjusting the end date of the preceding record, and the first date of the following record. Some operations have to be completely redone to account for the new range scheme.

Take this long distance pricing scheme as an example.

RATE_ID	RATE	FEFF_DT	LEFF_DT
1000	.049	2005Jan01	2006Jul31
1001	.039	2006Aug01	999Dec31

Today the company was sued by the PUC, and they have to rebill their customers for the month of Feb 2006. With classic effective dating, you'd have to touch everything underlined.

RATE_ID	RATE	FEFF_DT	LEFF_DT
1000	.049	2005Jan01	2006Jan31
1001	.039	2006Aug01	999Dec31
7000	.025	2006Feb01	2006Feb28
7001	.049	2006Mar01	2006Jul31

This is just the tip of the iceberg. Having survived 3 massive projects where the majority of our effort and bugs stemmed from effective dating I can assure you effective dating should be avoided at all costs if it is not absolutely necessary.

There is hope though. There is a way to have your cake and eat it too: Add only a First Effective Date to the table requiring data versioning.

When new records are introduced into the effective range scheme, no other record needs to be touched. The data is automatically adjusted. Retroactive operations all automatically adjust as well. Taking our example above, we have

RATE_ID	RATE	FEFF_DT
1000	.049	2005Jan01
1001	.039	2006Aug01
7000	.025	2006Feb01

Queries are a bit more involved, as they utilize analytics to get the correct record. But this can be hidden by a view, packaged cursors, etc. This schema can only accommodate versioning that has no gaps. If your data can allow gaps and overlap, then traditional last effective/first effective date columns, and their attendant headaches, will be required.

<Example query to be added>

Indexes

As you may already know, as you create the PK and UK constraints for your table, the supporting unique indexes will be automatically created by Oracle, named after the constraints. For this reason, when adding the constraint, you must include the USING INDEX clause so that you can specify the TABLESPACE of the supporting index.

You should create indexes to support each FK column on child tables, if that column is not already included in another index. This prevents locking issues in systems where the application allows end users to alter or delete parent table data.

Although it is less true with 10g than it used to be, it is still a good idea to place the most selective columns at the top of the column list in your composite indexes. I've not tested this rule of thumb with index compression.

Ensure that each and every index is assigned to the appropriate index-related TABLESPACE. Separating data from indexes is becoming less and less of a concern with each new version of Oracle though, so feel free to test this guideline for truth on the current release.

Function-Based Index

Use function-based indexes (FBIs) only where it makes sense and the function being encased in the index can't be made a permanent part of the data. Imagine you have a query that is slow because it wraps the predicate column in UPPER(), forcing a full table scan. You might consider creating a function-based index for the UPPER(), but what if you stored the data in uppercase to begin with? Wouldn't that be the smartest thing to do if the business rules allowed it? If the rules won't allow it, by all means, use the FBI, or the new-to-11g virtual column.

Domain Index

I have never used domain indexes and have no experience or recommendations regarding them. If you can develop a technical case for it, and it doesn't negatively affect performance, submit your idea back to the author for inclusion in the next version of the framework.

Descending Index

The one experience this author had with creating the index in descending order was negative. It slowed down a query by 200%. Instead, I had to go back to what was used before, using the INDEX_DESC hint, which was much faster. If you insist on creating your index with DESC turned on, ensure you thoroughly test performance against a baseline.

Reverse-Key Index

Try as I might, I have never been able to understand why anyone would need this structure. If you can develop a technical case for it, and it doesn't negatively affect performance, submit your idea back to the author for inclusion in the next version of the framework.

Bitmapped Index

Bitmapped indexes are wonderful…but only in a data warehouse (DW) environment where the data is fairly static or in a system where all the indexes are rebuilt frequently. Bitmapped indexes can grow exponentially larger than the base table they are indexing (even with infrequent row addition/modification). This removes all the benefits that bitmapped indexes are supposed to provide (space and speed). If you are working on a DW-like system, please use bitmapped indexes liberally.

Views

I currently have no special advice on regular views. The only exception is that hints should never be placed within a view unless absolutely necessary. And you may need the FORCE clause to avoid invalidation issues.

Please see the PL/SQL Standards document for how a view is to be constructed and where to place the comment block.

Views can have column comments. If a column in your view is particularly complex, please add a column comment explaining it.

Updateable Views

The use of updateable views (UV) is acceptable. Unless your UV is rather simple, you will probably need to build an instead-of trigger to control DML directed at the UV.

Materialized Views

This subject could occupy an entire paper on its own. Since Oracle 9.2, materialized views (MVs) have been more reliable than they were in previous releases. I recommend them for performance-intensive environments where denormalization or caching of joins is required to meet response time goals.

Object Views

I currently have no experience with object views. Tom Kyte highly recommends them to overcome much of the object-relational impedance mismatch of which so much has been spoken and so little done.

Constraints

Use integrity constraints. Use them liberally. This is what the database lives for. Next to reading and writing data, protecting data is what a database does best. As compared to protecting data by hand-written code in the presentation or application layer, integrity constraints are much better in terms of declarative ease, centralized storage and implementation of business rules, development productivity, and superior performance.

As you finish laying out the data elements in each table, your next step should be ensuring that no dirty data would be allowed in your table. Enter integrity constraints…

Primary Key

Ideally, each table will have a sequence-generated primary key. This is called a surrogate key. It is small. It is unique. It queries very quickly. It has no inherent meaning and will never need to change. It is easy to use when joining related tables.

Natural keys, on the other hand, are often multi-column and not small. They have inherent meaning and may need to change in the future, causing large update and data conversion operations (translation: big downtime). They are more difficult to use when joining tables, and cause related indexes and foreign keys to be larger than they need to be. They simply make your data model more complex and harder to work with than necessary, and produce a more fragile system than one built with surrogate keys.

Please do not consider using natural keys just because they seem like a little less work to maintain than a sequence per table. Using natural keys as your primary key **will** come back to haunt you.

Ensure you use USING INDEX in your constraint statement, and specify the appropriate index-related TABLESPACE.

Unique Key

When one uses surrogate keys as the primary key for a table, that still leaves the original natural key which must be constrained to prevent unwanted duplicates. The natural key columns of a table should be placed into a unique key. In some systems, these are called alternate keys.

Once you have created the surrogate PK and natural UK, you should be done with unique indexes. If your table still needs an additional unique index, something is probably wrong with your model. Examine the data requirements closely to find the misunderstanding.

Ensure you use USING INDEX in your constraint statement, and specify the appropriate index-related TABLESPACE.

Referential Constraint

If there is a reference table (lookup table) that contains the valid values for a column in your table, include a foreign key to it. There are only two reasons where it might be acceptable to not include the foreign key:

1. (Every millisecond of performance counts OR every byte of space must be preserved) AND the table is modified only by automated systems that get their data from clean in-house sources.
2. Historical/audit trail tables where you wish to preserve a snapshot of all values as they were on the day the delta was recorded. This is known as an educated choice to allow orphaned data.

DELETE CASCADE & SET NULL Rule

Automated cascading of updated parent values, or the emptying of the child values, is a decision best left up to each application's DB architect. Some applications can benefit from having automated cascades. Others require a high degree of control over the sequence of events when parent data changes.

In general, if your tables contain critical data, do not use DELETE CASCADE or SET NULL.

The SET NULL rule is particularly problematic as it leaves orphaned child data with no indication as to where it came from or how it was related. I can’t foresee a good reason why this rule would ever be used.

Check Constraint

Reference tables and referential constraints maintain most valid-values integrity. In rare situations, such as a two-value column that will never be used anywhere else and never changed, a check constraint makes more sense. Check constraints should remain relatively simple. If you have to jump through hoops and work SQL magic to make the check constraint work, something is probably wrong with the business rule or the data model. Approach the designer about this. In this case, the business rule probably needs to be validated by application logic in the front-end or middle tier, not a check constraint.

NOT NULL Constraint

Place a NOT NULL constraint on every column in each primary key.

Place a NOT NULL constraint on every applicable column in each unique key. Some columns in unique keys may, of a necessity, be nullable, but this is rare.

It is preferable to place all NOT NULL columns at the top of the table, leaving the optional columns at the bottom.

Each flag column, which is meant to be binary (Y or N, 1 or 0), should be NOT NULL to prevent tertiary logic and unnecessarily complex SQL.

Triggers

In most circumstances, you will find that the problem being solved does not require a trigger. Triggers are appropriate in the right circumstances, often for enforcing complex data integrity rules or data replication. But they tend to slow a system's DML response time. Evaluate the use of triggers in the context of your system's data volume and performance goals. Choose wisely. Prototype with representative data samples if you are unsure.

Usually, there is a better solution to the problem than throwing a trigger at it, including questioning the previous business decisions that created the problem itself. Why is the problem this difficult? How did it get this way? Does the model need to be simplified? Is the requirement reasonable? Should it be approached on the front end, or in the application server, rather than the database? And so on.

If you must write a trigger, keep it short and sweet. If it is more code than a page of code, put the code in a packaged procedure, and call it from within the trigger. Pass to your routine two records containing the :old and :new values along with the triggering DML action (INSERTING, UPDATING, DELETING).

Be aware of instead-of triggers, mutating tables (and their trigger/package solution), and system triggers, which are all very useful for the right circumstances. Please see the PL/SQL Standards document for further guidance in trigger construction.

Sequences

Generally, the default syntax for sequence creation yields a sequence sufficient for most.

```
CREATE SEQUENCE <sequence name>;
```

This creates a sequence that starts at 1, has no maximum, increments by 1, has a cache of 20 and doesn't cycle. If your application will be using a sequence hundreds or thousands of times a second, use a larger cache, generally 1000 or more. Other than that, we can think of no good reason to fiddle with other sequence creation attributes. For your information, the only other attribute is ORDER. This attribute is useful for those with RAC systems.

Each table with a surrogate PK column will have its own sequence.

Sequences should be called directly using single or bulk DML. For example,

```
INSERT INTO … SELECT my_table_seq.NEXTVAL…
```

Code that requires the sequence numbers for later processing would use the single or bulk RETURNING clause within the INSERT to store the sequence numbers into variables.

An acceptable alternative for populating sequence-based columns is to use a before insert row-level trigger on each table. The trigger fires for each inserted record, determines if the sequence-based column is populated, and if not, populates it automatically with a call to NEXTVAL. This is not a good approach in systems with intensive data loading requirements.

Synonyms

Abstraction

See the naming standard. Synonyms should be named after the object they are meant to abstract or serve as proxy. Don't include tokens in the synonym name that reference schema names or database instances/hosts.

If you ever need to use database links, use views and synonyms to abstract the link. That way if the link must be changed, it only needs to be changed in one place. This also prevents a bunch of code from going invalid when you drop and recreate the link.

Security Considerations

Private synonyms are favored over public due to security concerns. However, there are legitimate reasons for public synonyms. Please speak with your physical DBA if you feel a public synonym is required.

A best practice is to create two accounts, one to own the actual database application objects, and one to serve as the "gatekeeper" or pass-thru account. The gatekeeper schema is the account used by the 2-tier clients and application server connection pools. All access to data and PL/SQL routines funnel through this gatekeeper schema, thus providing a single point of access that can be tightly controlled. The gatekeeper schema owns nothing, except for a bunch of private synonyms pointing to the objects in the owning schema to which it has been granted access. An alternative to the large mass of private synonyms is an AFTER LOGON trigger for the gatekeeper schema that switches its current schema using

```
EXECUTE IMMEDIATE 'ALTER SESSION SET CURRENT_SCHEMA = <app owner schema>;
```

User-defined Types

The use of user-defined types and aggregates is the only way to solve certain difficult problems. I have used them for three generic nested table types of string, number and date, as well as some application-specific complex structures.

If you need to construct a new collection or object type, please have your design reviewed by the physical DBA/architect prior to implementation.

Remember that scripts to create user-defined types must end in the "/" character. The usual statement-ending semi-colon will not work to run the statement.

Queues

Design and implementation standards have yet to be defined for using Oracle Advance Queuing. Until then, refer to the Oracle documentation and the physical DBAs for information regarding their use.

Jobs

Oracle jobs should be reviewed and approved by the technical lead, database architect or physical DBA.

Do not place extensive PL/SQL code within an Oracle job (the "what" parameter). It is difficult to debug and manage code stored within a single column. Instead, place the job's main body of code within a packaged function or procedure and call the routine from within the job's anonymous block.

I have yet to define and implement a a common framework to monitor and manage Oracle jobs. I may do so starting with 10g since there is a new built-in feature for job scheduling.