An Oracle White Paper
September 2013

With Oracle Database 12*c*, there is all the more reason to use database PL/SQL

Abstract

Fifty-year-old wisdom instructs us to expose the database to client-side code as a PL/SQL API and to securely hide the implementation details — the names and structures of the tables, and the SQL statements that manipulate them — from the client. Yet a very large number of customers ignore, or refuse to follow, this wisdom. Some of the reasons for refusal have a reasonable justification. It is claimed that PL/SQL in the database cannot be patched without causing unacceptable downtime; that the datatypes for passing row sets between the client-side code and the database PL/SQL are unnatural, cumbersome to use, and bring performance problems; and that it is impractical to give each of a large number of developers a private sandbox within which to work of the intended changes to database PL/SQL.

Oracle Database 12*c*, hereinafter 12.1, brings changes that demolish each of these objections.

There is now no excuse to follow the best practice principle that is already universally followed in all software projects except those that use a relational database as the persistence mechanism.

Disclaimer

With Oracle Database 12*c*, there is all the more reason to use database PL/SQL

Introduction

The general wisdom is that the successful implementation of a large software system depends upon good modular design. A module is a unit of code organization that implements a coherent subset of the system's functionality and exposes this via an API that directly expresses this, and only this, functionality and that hides all the implementation details behind this API. Of course, this principle of modular decomposition is applied recursively: the system is broken into a small number of major modules, each of these is further broken down, and so on. This principle is arguably the most important one among the legion best practice principles for software engineering — and it has been commonly regarded as such for at least the past fifty years.

These days, an application that uses Oracle Database to implement its database of record is decomposed at the coarsest level into the database tier, the middle tier, and the browser tier. The focus of this paper is limited to the database tier.

The ultimate implementation of the database tier is the SQL statements that query from, and make changes to, the content of, the application's tables. However, very commonly, an operation upon a single table implements just part of what, in the application's functional specification, is characterized as a business transaction. The canonical example is the *transfer funds* transaction within the scope of all the accounts managed by an application for a particular bank. The *transfer funds* transaction is parameterized primarily by identifying the source account, the target account, and the cash amount; other parameters, like the date on which the transaction is to be made, and a transaction memo, are sometimes required. This immediately suggests the API:

```
function Transfer_Funds(Source in ..., Target in..., Amount in..., ...)
  return Outcome_t is ...
```

The point is made without cluttering the example by showing the datatypes and all the possible parameters. The API is specified as a function to reflect the possibility that the attempt may be disallowed, and the return datatype is nonscalar to reflect the fact that the reason that the attempt is disallowed might be parameterized, for example by the shortfall amount in the source account.

We can see immediately that there are several different design choices. For example, there might be a separate table for each kind of account, reflecting the fact that different kinds of account have different account details; or there might be a single table, with an account kind column, together with a family of per-account-kind details tables. There will similarly be a representation of account holders, and again these might have various kinds, like personal and corporate, with different details.

The point is obvious: a single API design that exactly reflects the functional specification may be implemented in many different ways. The conclusion is equally obvious:

The database module should be exposed by a PL/SQL API.
And the details of the names and structures of the tables, and the SQL that manipulates them, should be securely hidden from the middle tier module.

I know of many customers who strictly adhere to this rule; and I know of many who do not — to the extent that all calls to the database module are implemented as SQL statements that explicitly manipulate the application's tables. Customers in the first group seem generally to be very happy with the performance and maintainability of their applications. Ironically, customers in the second group routinely complain of performance problems (because the execution of a single business transaction often involves many round trips from the middle tier module to the database module). And they complain of maintenance problems (because even small patches to the application imply changes both to the implementation of the database module and to the implementation of the middle tier module).

Why then do some customers refuse to acknowledge fifty-year-old wisdom, and choose rather to bring themselves problems? Here, briefly, are the reasons I hear:

- My religion prevents me from using PL/SQL in the database.
- changing PL/SQL in the database requires shutting down the application; and the downtime might be very long
- the datatypes for passing row sets between the client-side code and the database PL/SQL are unnatural, cumbersome to use, and bring performance problems
- it is impractical to give each of a large number of developers a private sandbox within which to work of the intended changes to database PL/SQL

Sadly, no release of Oracle Database will be able to address the first objection. However, 12.1 brings new features that demolish each of the remaining objections.

Edition-based redefinition enhancements

Edition-based redefinition, hereinafter EBR, was introduced in Oracle Database 11*g* Release 2, hereinafter 11.2. In that release, it had some properties that made its adoption, in general, possible only after making changes to the disposition of the application's objects among their owning schemas — sometimes to the extent of creating new schemas and repositioning objects into those. 12.1 brings enhancements that remove all barriers to adopting EBR.

EBR allows database PL/SQL, views, and synonyms to be changed in the privacy of a new edition, without at all interrupting the database's availability, using exactly the same DDL statements that would be used to patch the application's server-side code in a downtime exercise. The changes made in the new edition are invisible to the running application. Only when the code in the new edition is fully installed and quality controlled, are the application's database sessions simply failed over to the new edition, session by session, until no sessions any longer are using the old edition. Then the new edition is established as the run edition; and the old edition is retired.

EBR allows table changes, too. But to use it for this purpose requires some thought, some changes to the application's schema objects, and a new way of writing the patch scripts. However, the use of EBR, in 12.1, to allow online changes to only PL/SQL, views, and synonyms is fruit that is hanging so low that it is literally on the ground.

Those who know something about EBR will know that the adoption barrier was a consequence of the rule that a non-editioned object may not depend upon an editioned object[1]. This barrier is removed by two distinct enhancements, thus:

- *The granularity of the editioned state of the name of a PL/SQL unit, a view, or a synonym is now the single occurring name.* New DDL statements control this new degree of freedom.

  This means that, for example, *Scott.Name_1* can be chosen to be editioned and can denote a procedure in one edition and a function (obviously with a different implementation) in another edition. On the other hand *Scott.Name_2* can be chosen to be non-editioned, and will, therefore, have the same implementation in all editions. In 11.2, the granularity of the editioned state was the entire schema: every name of every object in that schema, whose type is editionable, was either editioned or non-editioned.

  The new per-object granularity of the editioned state solves the adoption problem met when a table has a column whose datatype is defined by a user-defined-type, hereinafter UDT, in the same schema as the table. Because tables are *not* editionable, and UDTs *are* editionable, a table is not allowed to depend on UDT. Therefore if, in 11.2, an attempt was made to editions-enable a schema owning both a table and a UDT dependency parent, using the *alter user enable editions* statement, then the attempt caused an error, to prevent violation of the rule. The attempt would succeed only after re-establishing the UDT in a different schema, that would not be editions-enabled. This, of course, implied moving all the data from a column defined by the UDT in its old location to one defined by the UDT in its new location.

- *Materialized views and virtual columns have new metadata in 12.1 to specify the edition to be used to resolve the name of an editioned dependency parent.* They also have new metadata to specify the range of editions within which the optimizer may consider the object. New DDL statements control setting this new metadata.

  The subquery that defines a materialized view may refer to a PL/SQL function. And a table may have a virtual column whose defining expression includes a reference to a PL/SQL function. (A virtual column is the modern way to implement the functionality of a function-based index.) But materialized views and tables are non-editionable. Therefore, in 11.2, a schema with either a materialized view, or a table, with a PL/SQL dependency parent could not be editions-enabled because of the rule that a non-editioned object may not depend upon an editioned object. While the new per-object granularity for the editioned state might be used in 12.1 to overcome this problem, this would lead to discomfort in the bigger picture of the application design. A major benefit of EBR is that exactly that it allows changes to PL/SQL units while they are in use — so this benefit must not be sacrificed.

  The solution is to address the rule that a non-editioned object may not depend upon an editioned object in a different way. The editioned dependency parent is found by looking in the edition that the non-editioned dependent object specifies explicitly in its metadata.

  The semantic model for EBR specifies that creating a new edition causes a copy of every editioned object in the parent edition into the new child edition. But the customer's mental model of the implementation specifies that the *copy on change* paradigm will be used to create the new edition. Only when an editioned object is first changed, will there be two distinct objects in the old and the new editions. According to the semantic model, a materialized

1. In 11.2, the rule for the name resolution of an editioned dependency parent is to look for it in the same edition as the dependent object. So when the dependent object is non-editioned, the rule cannot be followed. Therefore any attempt to introduce a dependency that violates this rule causes an error. The attempt may be made explicitly, by a DDL statement on an individual object. Or it may be made implicitly by using the *alter user enable editions* statement.

view or an index on a virtual column would be usable only in the edition in which the PL/SQL dependency parent exists. This would imply creating a new materialized view or table, or course with a new name, for each new edition. To avoid this expense, a materialized view and a virtual column have new metadata to specify the range of editions within which the optimizer may consider the object when computing the execution plan for a SQL statement. This accommodates the fact that the PL/SQL might indeed have different copies in the parent and child editions because of a change to the unit that does not change the semantics of the function defined within the unit. This might occur, for example, if the function is defined in a package specification and a change is made to a different subprogram defined in the same unit.

NOTE: If the schema, prior to EBR adoption, has an occurrence of an index defined on an expression that includes a reference to a PL/SQL function, and if the post-adoption plan needs this function to be editioned, then the index must, in the editions-enabled schema, be dropped and re-created on a virtual column defined using the expression that used to define the index.

The general availability of Oracle E-Business Suite Release 12.2 was announced in October 2013, certified on Oracle Database version 11.2.0.3. It has been enhanced to use EBR, by making whatever changes that were needed to overcome the adoption barriers described above. From now on, every patch and upgrade will be done as an EBR exercise. The downtime that patching caused, in earlier releases, was of somewhat unpredictable duration on the order of hours, or even days, By using EBR, the downtime will be slashed to a predictable number of minutes.

Improved binding from client code to database PL/SQL

New in 12.1, the invocation of PL/SQL subprograms from the client can now bind values of non-SQL datatypes: records and index-by-PL/SQL-tables. This is important because, in a PL/SQL subprogram, an index-by-PL/SQL-table of records is the natural target for *select... bulk collect into...* and the natural source for *forall j in 1..My_Collection.Count(*)...*

Through 11.2, the invocation of PL/SQL subprograms from the client could bind only variables of the SQL datatypes. This meant that the contents of a record, or an index-by-PL/SQL-table with any kind of payload, had to be copied to, or from, a variable of a corresponding SQL datatype to be passed to, or from, the client. For example, a record had to be copied to a UDT defined using *create type t as object...*, hereinafter ADT (for abstract datatype); and an index-by-PL/SQL-table of records had to be copied to a nested table of ADTs. This led to cumbersome code and a performance penalty.

Now in 12.1, when using, for example, a JDBC client, JPublisher can be used to create the definitions of Java classes, to be used in the client for binding, by deriving them from PL/SQL datatypes defined in a package specification. The OCI, similarly, has mechanisms to bind values of non-SQL datatypes.

Recall that a PL/SQL subprogram must be invoked from a client by using an anonymous PL/SQL block — which is a kind of SQL statement[2]. Therefore, the ability to bind non-SQL datatypes from the client implies the ability to bind values of these datatypes into an anonymous block, and into other kinds of SQL statement. For example, a PL/SQL

2. The *call* statement is syntax sugar: it is transformed into a an anonymous PL/SQL block under the covers.

subprogram with a formal parameter declared using an index-by-PL/SQL-table of records can be invoked using dynamic SQL from another PL/SQL subprogram. Similarly, when a PL/SQL function is used in an expression in a DML statement, it can now have formals of non-SQL datatypes. Here is a particularly interesting example:

```
select * bulk collect into IBBI_1 from IBBI_2 where...
```

Here, *IBBI_1* and *IBBI_2* are variables of the same index-by-PL/SQL-table of records declared in the package specification of the package body that contains this code. This brings the declarative, set manipulating power of SQL into the implementation of ordinary PL/SQL code. Compare this with the complexity of implementation the functionality of restriction on a collection using looping and *if* tests.

The multitenant architecture and pluggable databases

Serious software development uses a source control system to record the canonical definitions of a system as it evolves and branches. The database module is represented in a source control system by the set of SQL statements that establishes the database artifacts that implement this module. There may also be sets of SQL statements that patch or upgrade one version of the database module to another version. The most interesting example of such a SQL statement, for the present discussion, is the *create or replace* DML statement for a PL/SQL unit, because typically, during a development cycle, procedural code is changed far more frequently than is the definitions of the shapes of tables, and so on. In a particular release cycle, several of the SQL statements that define the database module are changed.

When, as is common, the changes to the defining SQL statements are made jointly by the several members of a development team, then a discipline must be followed. Individual developers check out, change, and check back in individual SQL statements Of course, in order to make and test the intended change to a SQL statement, it is necessary to instantiate the database module using the checked-in set of SQL statements as of some label, and to work within this environment. This suggests that every developer must have his own private database for this purpose. I know of several customers who, in recognition of this, have developed fairly sophisticated internal applications to allow a developer to self-provision a database as of a source control system label. The most successful of these internal applications make explicit use of thin provisioning schemes provided by filesystem vendors because otherwise the space consumption by a database for each of many tens of developers would be unacceptable.

Of course, writing such an internal application is a non-trivial task; and, sadly, I know of other customers who have found the ideal approach too daunting. Instead, they use an *ad hoc* scheme where many developers make the intended changes to their checked-out SQL statements in a single, shared database. Obviously, this leads to no end of difficulty; and these difficulties increase exponentially as the size of the code base increases. This effect is exacerbated when the number of PL/SQL is large.

This has led some customers to limit the use of database PL/SQL — for example, by allowing it only for triggers. And it has led others avoid it altogether.

The advent of the multitenant architecture in 12.1 changes this. A pluggable database, hereinafter PDB, can be cloned from a read-only master PDB that represents a particular source control system label using SQL statements. And it can later be dropped using SQL statements. When the underlying filesystem supports thin provisioning, then the *snapshot* keyword can be used in the *clone PDB* SQL statement so that the cloned PDB is provisioned instantly and with no storage expense. Only when changes are made, will blocks be written to represent the deltas.

The fact that the *clone PDB* and *drop PDB* operations are exposed as SQL statements makes it very straightforward to write a PL/SQL application to allow developers to self-provision their PDBs.

Conclusion

The following enhancements, brought by Oracle Database 12*c*, have been discussed in this paper:

- *Edition-based redefinition:* The granularity of the editioned state of the name of a PL/SQL unit, a view, or a synonym is now the single occurring name. Materialized views and virtual columns have new metadata to specify the edition to be used to resolve the name of an editioned dependency parent. They also have new metadata to specify the range of editions within which the optimizer may consider the object.

- *PL/SQL:* Values of non-SQL datatypes can be bound to the formal parameters of database PL/SQL subprograms invoked from the client. In particular, row sets can now be passed between the client and the database using the natural datatype: an index-by-PL/SQL-table of records.

- *The multitenant architecture:* The *clone PDB* operation, taking advantage of the snapshot facility in the underlying filesystem, and the *drop PDB* operation are exposed as SQL statements. This makes it very straightforward to write a PL/SQL application to allow developers to rapidly, and thinly, self-provision a private database environment in which to change and test their checked out code.

There is now no excuse for violating the best practice principle that is already universally followed in the majority of software projects. Now you can confidently expose the Oracle Database using a PL/SQL API — hiding all the details of tables and SQL statements behind this API — knowing that this approach brings only benefits.

With Oracle Database 12c, there is all the more reason to use database PL/SQL
September 2013
Author: Bryn Llewellyn

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200

oracle.com

Oracle is committed to developing practices and products that help protect the environment

Hardware and Software, Engineered to Work Together