ORACLE DATABASE 12c

Why use PL/SQL?

ORACLE WHITEPAPER | November 2016

ORACLE

Contents

Abstract

Large software systems must be built from modules. A module hides its implementation behind an interface that exposes its functionality. This is computer science’s most famous principle. For applications that use an Oracle Database, the database is, of course, one of the modules. The implementation details are the tables and the SQL statements that manipulate them. These are hidden behind a PL/SQL interface. This is the Thick Database paradigm: *select*, *insert*, *update*, *delete*, *merge*, *commit*, and *rollback* are issued only from database PL/SQL. Developers and end-users of applications built this way are happy with their correctness, maintainability, security, and performance. But when developers follow the NoPlsql paradigm, their applications have problems in each of these areas and end-users suffer. This paper provides PL/SQL devotees with unassailable arguments to defend their beliefs against attacks from skeptics; skeptics who read it will see the light.

Introduction

The design of an application that uses an Oracle Database for persistence needs to address these main challenges: the specification of the business functions that it must perform, and the requirements for the information model that these imply; how the end-user chooses between business functions, provides facts that drive them, and sees the effects they have; and how the effects of the business functions are persisted and retrieved. It goes without saying that the application will be valueless unless only correct and complete data is persisted and retrial exactly honors the request.

PL/SQL's purpose is to enable the correct and efficient implementation of the business functions and the persistence and retrieval of their effects. While SQL famously provides an excellent scheme to express persistence and retrieval declaratively, it is equally famously unable to express higher level business logic. To choose between possibly radically different courses of action, according to currently available facts, requires programming in a good old *if... then... else* language. PL/SQL meets its purpose by allowing *select*, *insert*, *update*, *delete*, *commit*, and *rollback* statements, using genuine SQL syntax, among the classical procedural statements that its parent, Ada, gave it. In PL/SQL, therefore, a SQL statement can be seen as a special kind of subprogram.

It's common to lead with concerns about performance; but it would be meaningless to try to improve the performance of an incorrect application. Correctness must first be established; and only then, may performance be considered.

I'll therefore take the opportunity, in this paper, to specialize software engineering's central, generic principle for maximizing the chance of application correctness to address the question *"Why use PL/SQL?"* It turns out that we can both have our cake and eat it: the architectural approach that maximizes the likelihood of application correctness is the same one that brings optimal performance – and optimal security.

And finally, an acknowledgment: There's a lot of overlap between what I write about in this paper and what Toon Koppelaars[1] writes about in the *Helsinki Declaration*.[2] For at least the last decade and a half, Toon has been a practical exponent of, and a vigorous advocate for, what I formalize here as the Thick Database paradigm. He calls it the fat database paradigm. Over the years, I've heard both terms used often by various speakers at conferences for users of Oracle Database.

Applying the principle of modular design to an application that uses an Oracle Database for persistence

Few would deny that the correct 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 that exposes this via an interface. The interface directly expresses this, and only this, functionality; and it hides all the implementation details behind this interface. 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 half century.

These days, an application that uses an Oracle Database as its persistence mechanism is decomposed at the coarsest level into the database module, the application server modules, the modules that implement the graphical user-interface, and so on. Self-evidently, the overarching requirement that the database module must meet is at, all times, to persist only complete and correct data, and to retrieve it correctly. The completeness and correctness criteria are prescribed by the specification of the business functions, and so the database

1. Toon's Twitter handle is @ToonKoppelaars: twitter.com/ToonKoppelaars

2. thehelsinkideclaration.blogspot.co.uk/2009/03/start-of-this-blog.html

module cannot be responsible for meeting its requirement without the authority brought by owning the entire implementation of the business function logic together with the persistence and retrieval of the data.

The ultimate implementation of the database module is the SQL statements that query, and make changes to, the application's tables. However, an operation upon a single table often implements just part of the persistence requirement of what, in the application's functional specification, is characterized as a particular business function. The canonical example is the transfer funds business function within the scope of all the accounts managed by an application for a particular bank. This function 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 typically also required.

This immediately suggests this interface:

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

It is specified as a function to reflect the possibility that the attempt may be denied; and the return datatype is non-scalar to reflect the fact that the reason for the denial might be characterized by several values, including, for example, the shortfall amount in the source account. These need to be presented back to the end-user.

Notably, the possibility to request that the action be done some time in the future implies all sorts of secondary persistence structures and daemons to act on the information that they store.

We can see immediately that there are many different plausible 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. There will doubtless be a table to hold requests for transfers that are due to be enacted in the future. There are very many possibilities for implementing these actions.

The point is obvious: a single interface 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 interface that models each business function according to its specification. All the procedural logic to implement each business function, and most certainly the details of the names and structures of the tables, and the SQL that manipulates them, should be securely hidden from outside-of-the-database entities.*

This is sometimes known as the *Thick Database paradigm*. It sets the context for the discussion of when to use SQL and when to use PL/SQL. The only kind of SQL statement that is allowed into the database is a *call* statement[3] that invokes exactly one of the interface's subprograms:

```
call Transfer_Funds(:s, :t, :a, ...) into :r
```

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. PL/SQL's package construct supports the coarsest granularity, and this construct explicitly supports interface exposure and implementation hiding. Then, subprograms do this at the next finer level. And subprograms within subprograms to arbitrary

3. The *call* statement is semantically equivalent to an anonymous PL/SQL block that does nothing more than invoke a single user-written procedure or function. The rule that every call to the database must be a *call* statement is terse and implies that every such call may invoke only a single PL/SQL subprogram. It would take more words to say this if an anonymous PL/SQL block were allowed as a top-level call. The *call* statement has the further advantage over the anonymous PL/SQL block that it is described in the SQL Standard. Of course, in Oracle Database, *call* statement can invoke only PL/SQL subprogram.

depth complete the picture. Because SQL statements can be seen as a special kind of subprogram, called from PL/SQL, the division of labor between the PL/SQL and SQL subsystems in the database, and how they communicate mutually, are usually the most critical determinants of overall PL/SQL performance.

The alternative to the Thick Database paradigm is to make every call to the database a SQL statement that explicitly manipulates the application's tables. I'll call this the *NoPlsql paradigm*. Of course, it makes the database tier unworthy of the term "module".

An alternative interpretation of the principle of modular design for an application that needs to persist data

These days, several frameworks exist that allow an information model, and the business functions for change and retrieval that should operate upon its instance, to be specified in a declarative fashion. Typically, they are implemented outside of the database and they allow the developer to choose the persistence mechanism; usually, the Oracle Database is just one possibility among several. *Figure 1* shows the general idea.[4]

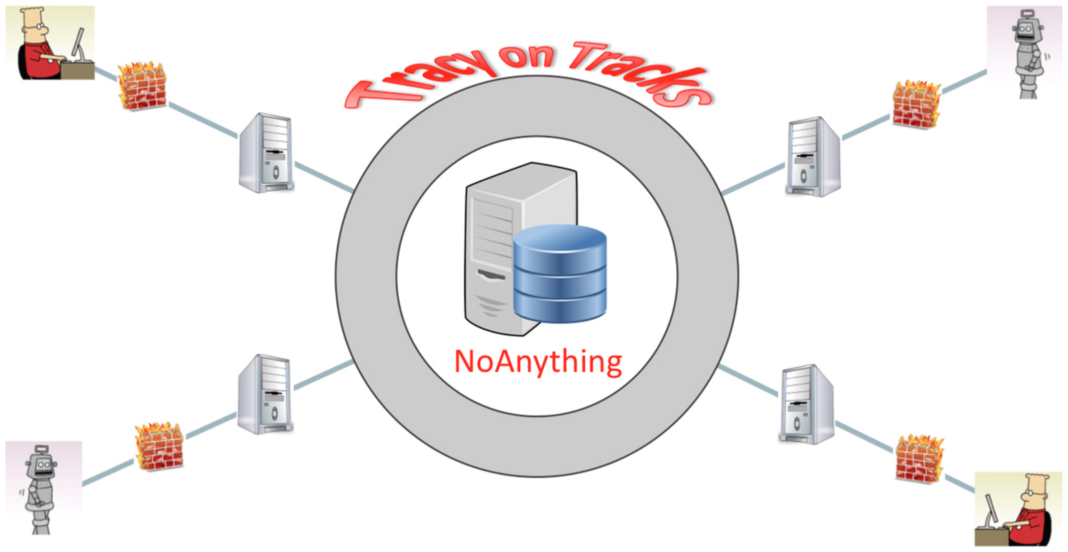


Figure 1. A sketch of the framework paradigm.

The defining paradigm is that the actual protocol, specific to the chosen persistence mechanism, that deals with the data is entirely mechanically generated by the framework. In the case that the Oracle Database is chosen as the persistence mechanism, the typical outcome is that the use of SQL is inefficient. For example, operations tend to be done row-by-row (sometimes referred to as slow-by-slow[5]) rather than in bulk. And features specific to the Oracle Database are not used. The most notable neglected feature is PL/SQL. Because the framework is distinct from the Oracle Database that it encapsulates, the performance penalty that comes from doing every individual *select*, *insert*, *update*, *delete*, *merge*, *commit*, and *rollback* in its own top-level database call becomes very noticeable. Nothing can be done about this type of problem except by human intervention to re-write some of the mechanically generated SQL statements by hand. But any such intervention immediately compromises the paradigm.

4. The names *Tina on Tracks*, *Tracy on Tacks*, and *NoAnything* are entirely fictitious. Any resemblance to actual product names is unintended.

5. This term is attributed to Tom Kyte.

Another feature of this approach is that the only top-level calls allowed to the Oracle Database must be those that come from the framework. This requires a special kind of policing that must supplement, rather than rely upon, the intrinsic security mechanisms provided by the Oracle Database. These problems, and others brought by using the framework paradigm, are popular presentation topics at Oracle User Group conferences. This paper will say no more about the framework paradigm.

> *Note:* When I first published this paper, the *Tracks* framework was at version 1 and was called *Tina on Tracks*. Now this has been replaced by version 2, called *Tracy on Tracks* – artfully implying compatibility. However, the versions aren't fully compatible, and so the shapes of the generated tables, and correspondingly the *select*, *insert*, *update*, *delete*, and *merge* statements that query and change them, differ slightly. This brings the need for a one-off *ad hoc* migration exercise[6].

The time-honored alternative to the framework paradigm uses ordinary human intellect to produce the table design that implements the application's information requirements. If a modeling tool is used, then at least the resulting table design is understood by developers, and the understanding informs the rest of the design of the application. This is the context for what this paper deals with.

What is SQL, and how to make it happen?

A SQL statement describes the effect it that it should have without specifying how to achieve that effect. This is why SQL is classified as a declarative language. Significantly, SQL has no larger building block than the single statement and therefore no terminator character is needed – and nor is one allowed. This implies that SQL cannot be used to write a program. Moreover, the compilation and execution model for SQL differs from that for many familiar programming languages. A SQL statement cannot be named, compiled to produce a persistently stored named executable, and then thereafter executed by mentioning its name. Rather, the text of a SQL statement must be submitted to an Oracle Database[7] for compilation and execution each time its effect is needed. This is done by using an *if... then... else* programming language or a tool that has been implemented using such a language that can interpret the SQL*Plus scripting language. From now on, I'll use the term *host language* for any *if... then... else* programming language that, by any means, is able to make SQL happen[8].

Some host languages handle SQL by providing only a subroutine interface for the explicit steps exemplified by these:

» *open a cursor*

» *compile a SQL statement* in that cursor

» when appropriate, *bind actual arguments to placeholders in the SQL statement*

» *execute the SQL statement*

» eventually, *close the cursor*

---

6. I was inspired to write this tongue-in-cheek note after watching the presentation by Hadi Hariri called *The Silver Bullet Syndrome*, published on 13-November-2015, posted here: *https://www.youtube.com/watch?v=3wyd6J3yjcs*.

---

7. The Oracle Database is our focus of interest in this paper. The general execution model for SQL is the same for all database systems.

---

8. I appreciate that the term *host language* is generally used more specifically. The SQL standard defines a scheme for embedding an extended form of SQL in languages like C or FORTRAN so that a precompiler can translate this into a lower level version that actually makes the real SQL happen. It calls such a language a host language. Oracle Corporation's Pro*C is an example of such a scheme. I need a less specific term, but none seems to have been established.

For a *select* statement, extra steps are needed: subroutine calls are used to establish a mapping between *select* list items and variables in the host language; and subroutine calls are used to fetch the results.

Given that a host language is needed to make SQL happen, where should it be run? There are only two choices: inside the database or outside the database.

Running the host language outside of the database implies submitting every ordinary *select*, *insert*, *update*, *delete*, *merge*, *commit*, and *rollback* to the database as a top-level call. This approach requires that the microscopic detail of the tables in every single schema and both the within-table constraints and the between-table constraints need to be understood outside of the database. inevitably, then, it will be possible to query and change every single table from outside of the database. Because famously some kinds of constraint cannot be implemented declaratively, the responsibility to enforce the data rules will lie partly outside of the database; and so will the responsibility to honor the specified integrity of the business functions. This approach means that the database cannot be responsible for meeting its charter: to persist only complete and correct data, and to retrieve it correctly and according to the various security rules that the overall application requires.

Needless to say, the communication with the database will be chatty and performance will suffer. But when correctness has already been sacrificed, this is hardly the most important concern.

The only sensible choice, then, is to run the host language inside the database. There are only two such languages that can run inside an Oracle Database: Java and PL/SQL. But Java has only subroutine support for SQL. The *DBMS_Sql* package is PL/SQL's subroutine interface for doing SQL; but it also, uniquely, has language features for doing SQL. We shall look at these in the section *"PL/SQL's unique specific features for making SQL happen"* on page 11.

The inescapable conclusion is that every ordinary *select*, *insert*, *update*, *delete*, *merge*, *commit*, and *rollback*, must be issued from PL/SQL units stored in the database.

How, then, can the PL/SQL subprograms that expose the database interface be invoked from outside of the database? The *call* statement allows exactly this. Because it is just one particular kind of SQL statement, it can therefore be executed from any outside-of-the-database SQL-aware host language. *Code_1* shows the invocation of a procedure.

```
-- Code_1
call Pkg.Bulk_Insert(:Rows)
```

We can now name, and characterize, two paradigms for the architecture of an application that uses an Oracle Database:

» *The NoPlsql paradigm:* every kind of SQL statement is allowed from outside-of-the-database entities except for two, the *call* statement and the anonymous PL/SQL block, because of a religious objection to PL/SQL. The implementation of a business function typically needs lots of round trips to and from the database. The time for these might dominate the time for executing the SQL itself.

» The *Thick Database paradigm*: the only kind of SQL statement that is allowed from outside-of-the-database entities is the *call* statement. Every business function needs just a single round trip to and from the database. Thereafter, the PL/SQL-SQL-PLSQL round trips that implement the business functions take place within the same operating system process.

Binding *in* and *out* actual arguments to placeholders when a *call* statement is invoked from outside-of-the-database code

Outside-of-the-database code whose purpose is to submit SQL statements for execution in an Oracle Database, and to receive their results, must use the Oracle Net protocol. Because Oracle Corporation does not document this protocol, outside-of-the-database code must, ultimately, use the subroutine interface to Oracle Net exposed by the Oracle Call Interface library, hereinafter the OCI, or a subroutine interface built on top of this like ODBC or JDBC[9] that is specific for a particular host language environment. The subroutines for binding, in some particular outside-of-the-database host language, have formal parameters whose datatypes are defined in that language specifically to match the Oracle Database datatypes that the SQL statement placeholders stand for.

Through Oracle Database 11*g* Release 2, it was possible to bind only to placeholders standing for formal parameters and variables with SQL datatypes but not to those standing for PL/SQL-only datatypes like *record* and *index-by-pls_integer table*. This restriction caused noticeable discomfort because an *index-by-pls_integer table* of *record* is the target for PL/SQL's *bulk select* feature and the source for its *bulk DML* feature[10]. A reasonable workaround was available; but it was rather cumbersome and had some performance drawbacks. It isn't necessary to say any more about the workaround because, starting with Oracle Database 12*c*, the OCI now allows binding to placeholders that stand for all of the PL/SQL-only datatypes except for *index-by-varchar2 table*; and ODBC and JDBC, for example, inherit this improvement.

In modern multi-tier architectures, where the relationship between outside-of-the-database entities and the database is stateless, the response to any end-user query is never more than tens of records. A large result set is delivered in successive pages; and each new page request implies a newly executed *select* statement that uses an appropriate pagination restriction. An *index-by-pls_integer table* of *record*, populated by a *bulk select* statement, is the perfect structure to pass a page of results from the database to an outside-of-the-database program. Similarly, when the end-user interacts to gather, or modify, several records that then should all be persisted, an *index-by-pls_integer table* of *record* is the perfect structure to pass these to the database from the outside-of-the-database program so that it can be used as the source for a *bulk DML* statement.[11]

In summary, staring with Oracle Database 11*g* Release 2, the *call* statement allows the invocation, from an outside-of-the-database program, of any arbitrarily parameterized PL/SQL subprogram.

The Thick Database paradigm

It is straightforward to install the set of artifacts that define an application's Oracle Database backend in such a way that the only items that outside-of-the-database code can see are the PL/SQL subprograms that implement the specified business functions. In other words, the Thick Database paradigm can be formally enforced.

The multi-schema model

This is the simplest enforcement regime:

» Install all the tables in a dedicated *Data* schema.

9. There are two flavors of JDBC. Thick JDBC is built on top of the OCI. And thin JDBC interfaces directly to Oracle Net using Java code. This distinction is of no consequence for what this paper deals with. Thin JDBC, by requirement, is indistinguishable from Thick JDBC from the viewpoint of Java code that uses it.

10. I'll refer to the *forall* statements for *insert*, *update*, or *delete* as *bulk DML*.

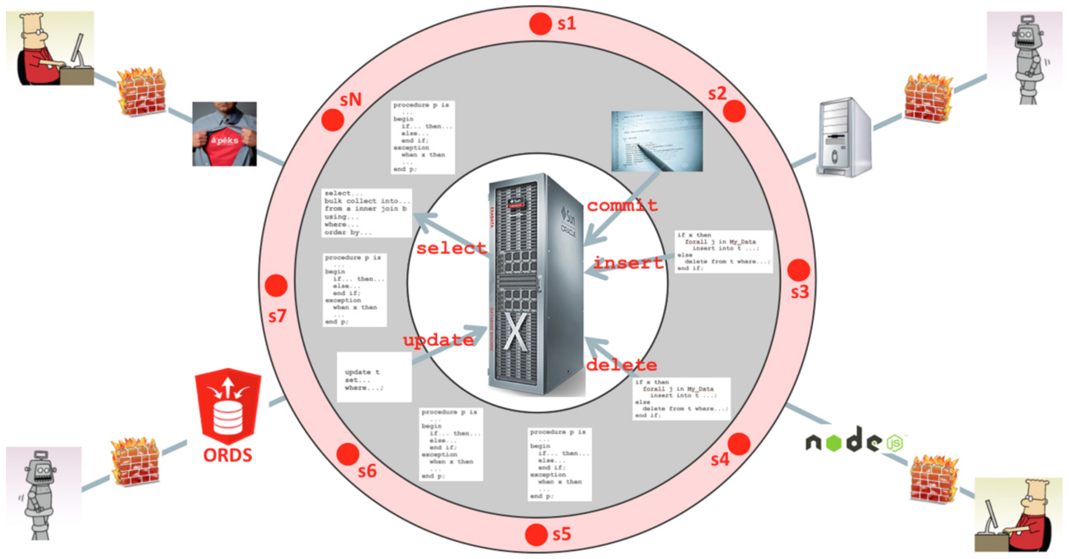
11. There seems, in these modern architectures, to be no use case for the *ref cursor*.

» Install all the PL/SQL units in a dedicated *Code* schema, using definer's rights units[12] and grant the *Select*, *Insert*, *Update*, and *Delete* object privileges on the tables in the *Data* schema to the owner of the *Code* schema.

» Install private synonyms in an otherwise empty *Access* schema for the subset of these units in the *Code* schema that defines the database interface[13]; and grant the *Execute* object privilege on only these units to the owner of the *Access* schema.

» Let only the name and password for database user that owns the *Access* schema be known to developers of outside-of-the-database code.

This basic regime can be refined by using several schemas for the tables, to reflect a classification scheme for the data, and several schemas for the PL/SQL units, to reflect a classification scheme for the functionality. A common scheme is to separate tables for usage auditing from those for the application's substantive data, and to separate the PL/SQL units that manipulate these two kinds of tables accordingly. *Figure 2* shows the general idea.



```
procedure p is
  ...
begin
  if... then...
  else...
  end if;
exception
  when x then
  ...
end p;
```

```
select...
bulk collect into...
from a inner join b
using...
where...
order by...
```

```
update t
set...
where...;
```

```
if x then
  forall j in My_Data
    insert into t ...;
else
  delete from t where...;
end if;
```

Figure 2. The Thick Database paradigm.

The inner white circle represents the *Data* schema. The surrounding gray ring represents the *Code* schema. Notably, every single *select*, *insert*, *update*, *delete*, *merge*, *commit*, and *rollback* that executes when the application is in ordinary use is issued from one of the PL/SQL units in this schema. The outer pink ring represents the *Access* schema where the red dots *s1*, *s2*, and so on represent the synonyms that denote the subset of units in the *Code* schema that define the exposed interface. The only database calls that are allowed invoke the interface subprograms.

The picture shows a few examples of different kinds of outside-of-the-database entities that use the interface that the *Access* schema exposes. Notice that Oracle Application Express is shown. The fact that it actually runs inside the database is very interesting for various practical reasons. But conceptually it is a peer to the other outside-of-the-database entities. The specific mechanism that is used to invoke PL/SQL subprograms

12. The distinction between definer's rights and invoker's rights is discussed in the section *"The authid mode"* on page 17.

13. Some developers prefer to keep the *Access* schema totally empty and to implement, instead, a logon trigger that sets the *Current_Schema* to the *Access* schema.

varies between these entities. For example, a daemon whose purpose is to communicate with, for example, a remote service for verifying the use of a credit card (depicted by the robot icon) might be written in C using the OCI to invoke *call* statements explicitly. When Oracle REST Data Services expose the PL/SQL subprograms that define the interface, its server looks after the translation of an incoming HTTP request to an appropriate *call* statement[14] without the developer who uses it having to program this. The same applies for the composition of the response from the value returned by the nominated PL/SQL subprogram.

Compare *Figure 2* with *Figure 1* on 4.

» In the framework NoPlsql paradigm, it is the boundary of the framework itself that implements the hard shell round the persisted data and the functionality to manipulate it. And the framework defines the available protocols for exercising the functionality. For example, if the main application persists only current operational data and it is decided to implement a distinct business intelligence application to accumulate operational data over time, then this data would have to be extracted by using a protocol supported by the framework. When the protocol doesn't support bulk SQL, the performance of the system-to-system transfer can be punitively slow.

» In the Thick Database paradigm, it is the bare Oracle Database itself that implements the hard shell. Many access protocols are available out of the box, as *Figure 2* indicates. But it is always possible for a developer to implement an entirely new special scheme by programming ordinarily in, for example, C using the OCI or in Java using JDBC. Such a scheme can be designed specifically for optimal performance when large volumes of data are to be transferred. The hardness of the shell, exposed as it is by PL/SQL subprograms that can, in such new programming projects, be exercised using *call* statements, is not compromised.

Recommended best-practice refinement: formalize a convention to define the exposed subprogram interface

The interface is defined by a set of subprograms. In the general case, there may be several different clients of the database—each with its own *Access* schema—and each might need to use just its own subset of the interface-defining subprograms. This implies that it must be possible to grant the *Execute* object privilege with the granularity of the individual subprogram. It seems tempting, therefore, to define the overall interface as a set of schema-level procedure and function jackets that simply call their substantive counterparts. However, this brings a disadvantage. The *Execute* object privilege brings with it the ability to read the source code of the target of the grant, when this is a schema-level procedure or function, in the *All_Source* dictionary view. (For a package, the grant brings the ability to read only the source code of the spec; the source code of a package body is always hidden from all users but its owner.) Critically, as the next section explains, each interface subprogram will implement error-logging code. Security would be compromised if users that are authorized to use an interface subprogram could read how error-logging is done. The rule must therefore be to implement each subprogram in the set that exposes the database interface as a singleton procedure or function (or conceivably as a set of overloads with a single name) in its own dedicated package. The package spec can contain comments to explain the purpose and usage of the subprogram; and the body will hide the details of the error-logging. Some developers have told me that they like to take this formalization one step further and locate the PL/SQL units that define the interface in a dedicated schema that holds no other objects.

Business function atomicity

When the top-level database call is a *call* statement, and this ends with an Oracle error, all changes made by all *insert*, *update*, *delete*, and *merge* statements that it executed are automatically rolled back. However, when every *insert*, *update*, *delete*, and *merge* statement is executed each in its own server call, and when just one

14. The implementation might use an anonymous PL/SQL block rather than a *call* statement. This is unimportant for this paper's purpose because the programmer doesn't see the implementation of the invocation of the PL/SQL subprogram.

causes an Oracle error, only *its* effect is rolled back while the changes made by the other statements are not. Thus when the Thick Database paradigm is followed, the atomicity of each business function is guaranteed by a transparent, uniform scheme. But when the NoPlsql paradigm is followed, all outside-of-the-database code must be carefully policed to ensure that *commit* and *rollback* statements are properly placed.[15]

Each interface subprogram (when it changes data), but no others, would implement *commit* as its final executable statement. And each interface subprogram would implement an *others* handler to deal with the logging of unexpected exceptions and the invocation of *Raise_Application_Error()* to notify outside-of-the-database code that a fatal error had occurred, and to provide the incident identifier for a support ticket. The *others* handler would start with *rollback*[16]. This approach is illustrated in *Code_7 on page 17*.

With the formality in place that the interface is defined by a set of packages, each of which exposes just a single subprogram, it is easy to recap these critically important rules:

» Each interface-defining jacket subprogram does no more than to invoke the subprogram that exposes the substantive implementation, and to implement error-handling.

» The *commit* and *rollback* statements must be allowed only within the interface-defining package bodies[17].

» The final executable statement, before the exception section, of the implementation of *every* such interface-defining subprogram, when its purpose is to change data, must be the *commit* statement.

» No interface-defining subprogram may be called from database PL/SQL.

Mechanical functional testing and performance testing of the database interface

A huge benefit of the Thick Database paradigm is that both the functional correctness and the performance of the database module can tested independently of any outside-of-the-database code that will use it. This is the empirical proof of design by contract. The tests are written by developers whose speciality is Oracle Database development engineering. In other words they are SQL and PL/SQL experts. This means that all sorts of bonuses are immediately available. For example, test results can be recorded easily in database tables (probably in a different database than the one under test, by using a database link) for subsequent analysis and long-term archiving. If execution plans, or other such metrics need to be gathered, this can all be done by interleaving the SQL and PL/SQL calls for these purposes with the ones for exercising the functionality under test. The old SQL*Plus warhorse is often enough for running such scripted tests. But, of course, the test actions can trivially be invoked from dedicated PL/SQL units written specially to implement the test suite.

Notice that performance testing becomes very clean in this approach. All concerns about tier-to-tier latency vanish. This means that if, later, an end-to-end test that involves noticeable amounts of outside-of-the-database code shows poor performance, it will be immediately apparent where the blame lies. If the end-to-end performance metrics are close to those for exercising the database functionality directly, then the testers know one thing. And if the end-to-end metrics are much worse than the bare database metrics, then the testers know something entirely different.

15. We shall elaborate on this in the section *"Oracle errors are magically transformed to PL/SQL exceptions"* on page 16.

16. The *rollback* is not necessary because it is done implicitly when an unhandled exception escapes the top-level call. However, it is in no way harmful to write it, and doing so makes the behavior unambiguously clear to all readers.

17. If autonomous transactions are used for error-logging in units that implement the functionality that the interface-defining subprograms express, then these must, of course, also implement a *commit*.

When mechanical testing is easy, problems are disclosed early in the development cycle. But when, as is implied by the NoPlsql paradigm, testing requires simulation of the HTTP requests emanating from the end-user's browser, and is therefore hard, problems of all sorts – functional and performance – are disclosed only late, if at all, in the cycle.

Stubs for concurrent development of outside-of-the-database code and the implementation of the database interface

The Thick Database paradigm brings another benefit. The developers of the outside-of-the-database code can make progress on their side of the interface at the same time that the developers who are implementing the interface carry out that project. The approach needs only an early agreement on the specification of the interface with an adequate, interim, "mock" implementation.

PL/SQL's unique specific features for making SQL happen

Because PL/SQL is a specialization of Ada designed explicitly to work with the Oracle Database, it naturally has many unique features for making SQL happen. They allow code to be compact and clear, which hugely increases the likelihood of correctness. And they deliver run-time efficiency. The examples in the following sections all show how PL/SQL manages the *select* statement. Similar benefits apply for *insert*, *update*, *delete*, and *merge*.

PL/SQL supports all the SQL datatypes natively

Every SQL datatype can be used to declare a variable or a formal parameter in PL/SQL. This is hugely important for correctness because any scalar expression that can be used in a SQL statement can also be used in PL/SQL with the same spelling; and it will have the same semantics. In particular, the notion of *null*, with its famous rules for combination using boolean operators like *or* and *and*, is identical in PL/SQL and SQL.

PL/SQL allows so-called anchored declarations, using *%type* and *%rowtype*. This scheme lets the PL/SQL directly express the intention that a variable or formal parameter has the same datatype as a table column. Using it ensures code clarity, reliability, and maintainability.

Ordinary scalar SQL functions, like *To_Char()*, *Sqrt()*, *Floor()*, *Ceil()*, *Least()*, *Greatest()*, and so on are all available in PL/SQL.[18] Of course, aggregate functions like *Max()* and analytic functions like *Lag()* are available only in SQL.

Static SQL

PL/SQL allows all the SQL statements that an application ordinarily needs – *select*, *insert*, *update*, *delete*, *merge*, *commit*, and *rollback*[19] – as PL/SQL statements. Such statements are called static SQL. In some cases, the spelling of the static SQL statement is identical to that of the corresponding plain SQL statement. Where appropriate, the static SQL syntax extends that of plain SQL, for example with the *select... into* static SQL statement for getting exactly one row.

18. A few scalar SQL functions are not supported in PL/SQL. *Nvl2()* is an example. Enhancement request 21781286 asks for every ordinary SQL function to be supported in PL/SQL.

19. In other words, static SQL supports the class of SQL statements that allow binding the placeholders together with those for transaction control.

The *cursor for loop* is a very telling example of the unique value that PL/SQL adds in comparison to the subroutine approach that other languages must use. Consider the example shown in *Code_2*.

```
-- Code_2
procedure p(Lo in t.c3%type, Hi in t.c3%type) is
begin
  for r in (
    select t.c1, t.c2
    from t
    where t.c3 between p.Lo and p.Hi order by 1)
  loop
    Buffer_Row_For_Display(r.c1, r.c2);
  end loop;
end p;
```

with this declaration of the buffering procedure:

```
procedure Buffer_Row_For_Display(c1 in t.c1%type, c2 in t.c2%type)
```

Notice these features:

» The static SQL statement is spelled the same as the corresponding plain SQL statement except that, where the plain SQL statement would have placeholders, the static SQL statement has identifiers that cannot resolve in SQL scope but do resolve in PL/SQL scope. The PL/SQL compiler works in cooperation with the SQL compiler to derive this canonically spelled plain SQL statement text:

```
SELECT T.C1, T.C2 FROM T WHERE T.C3 BETWEEN :B2 AND :B1 ORDER BY 1
```

» The derived SQL statement text is stored with the compiled version of the PL/SQL unit so that when it is executed, the current values of *p.Lo* and *p.Hi* are bound to the placeholders *:B2* and *:B1*. We hear constantly of the evils of encoding would-be bind values as literals into the text of the SQL statement. This is encouraged by the subroutine style of doing SQL that is all that most outside-of-the-database host languages offer. It leads to a new so-called hard parse for each new execution; and the contention that this causes can bring a system to its knees under normal concurrent multi-session use. This is an ever-popular conference theme: *"bind to placeholders, don't concatenate values as text, to avoid the cost of hard parse"*. With static SQL, it is simply impossible to avoid following this rule.

» Cursor management is implicit. The PL/SQL runtime system looks after it, and as soon as control leaves the loop – ordinarily or because of an exception – the cursor is closed. Cursor leaks simply cannot happen.

» Stated more carefully, the cursor is actually held open, but marked as semantically closed. The PL/SQL runtime system may close it if there is space pressure, but typically does not need to. This means that when control later returns to the loop it is necessary only to bind and execute the SQL statement. The *"bind to placeholders"* mantra implies that, with hard parse avoided, the search to find the already-compiled child in the cursor cache is free. But cost of this search, the so-called soft parse, can be significant. PL/SQL's soft parse avoidance scheme therefore brings a further benefit on top of the hard parse avoidance.

» Array fetch is automatically used under the covers to reduce the cost of the PL/SQL-SQL-PLSQL round trip.

» The loop implicitly declares the variable *r* as a *record* with these fields: *(t.c1%type, c2 in t.c2%type)*.

The hard parse avoidance, implicit cursor management, and soft parse avoidance benefits apply for all variants of static SQL. And the same understanding that lets us see that the *"bind to placeholders"* mantra is implicit lets us see that SQL injection is simply impossible with static SQL. This is a huge benefit; and it comes effortlessly when *select*, *insert*, *update*, *delete*, and *merge* are issued only from PL/SQL using static SQL.

Native dynamic SQL

PL/SQL also provides native language support for doing dynamic SQL. In the very rare situations where static SQL doesn't support the SQL requirement, then it's very likely that native dynamic SQL will. In the even rarer situations where native dynamic SQL doesn't support the SQL requirement, then it's guaranteed that the *DBMS_Sql* subroutine approach will. The canonical example that calls for dynamic SQL is when the name of a table isn't known until run-time[20]. *Code_3* shows a simple example of how native dynamic SQL supports single-row *select*:

```
-- Code_3
function Facts_For_PK(Table_ID in Identifier_t, PK in t.PK%type) return Row_t is
  r Row_t;
  Safe_Table_ID constant Identifier_t :=
    Sys.DBMS_Assert.Simple_Sql_Name(Table_ID);
  Stmt constant Stmt_t := '
    select t.c1, t.c2
    from '||Safe_Table_ID||' t
    where t.PK = :b';
begin
  execute immediate Stmt into r.c1, r.c2 using PK;
  return r;
exception when No_Data_Found then
  r.c1 := null; r.c2 := null;
  return r;
end Facts_For_PK;
```

Notice these features:

» The text of the plain SQL statement is assembled programatically. In general, this brings the notorious risk of SQL injection. However, we can see by a simple inspection of the source code that the text is assembled only from pieces that are fixed at compile time together with the dynamically created text of the table's identifier. Critically, the use of the *Simple_Sql_Name()*[21] function in the *DBMS_Assert* package ensures that the identifier is properly formed and can denote only a table in the current schema.

» The syntax of *execute immediate* deals with the binding and the return of the *select* list values in a compact and clear fashion.

» If not exactly one row is selected, then an exception is raised.[22] *No_Data_Found* is raised when no row is returned. This is deemed to reflect a regrettable, but unsurprising, error: a bad choice was made for the value of the actual argument for the *PK* formal parameter. A handler is therefore provided for this, and the outcome is signaled in a useful way. *Too_Many_Rows* is raised when two or more rows are returned. The spelling of the *PK* formal parameter indicates that it denotes the table's primary key. Therefore, if *Too_Many_Rows* is raised, this means that a fundamental data rule has been violated. The only sensible policy is to let this exception bubble up to subprogram invoked by the *call* statement that started the present database server call. The outside-of-the-database program would then receive a generic *ORA-20000* error and an associated text like *"Sorry, something went wrong. Call Support and quote Incident_ID 42"*.

» The static SQL implementation of this query scenario will raise the *No_Data_Found* and *Too_Many_Rows* exceptions in the same circumstances.

» Though the SQL text that is submitted using *execute immediate* can be different each time the source code location is revisited, it often happens that many successive revisits use the same text. Under these circumstances, the same soft parse avoidance scheme is enjoyed as has been described for static SQL.

---

20. One reasonable scenario that leads to knowing a table's name only at run-time is when a pool of same-shape tables is provided for storing interim results during a session, and where the number of such tables that will be needed emerges only at run-time. Such tables are checked out of the pool, used for a period, and then checked back in.

21. This function should have been called *Simple_Sql_Identifier()*.

22. We shall return to this point in the section *"Oracle errors are magically transformed to PL/SQL exceptions"* on page 16.

» The items *Identifier_t* and *Stmt_t* are subtypes. This device comes from PL/SQL's parent, Ada. In PL/SQL, it provides a convenient way to give a name to a datatype that is anchored, using *%type* or *%rowtype*, to a table.

The other class of use cases that call for dynamic SQL is for SQL statements that simply are not supported by static SQL. Obvious examples are DDL statements, *alter session*, and *alter system*. While these are uncommon in ordinary application code, they are the very essence of installation scripts. Some developers draw benefit from encapsulating practices that they want to enforce, like for example that *create index* should always use the *online* mode, in PL/SQL procedures. *Code_4* is a simple way to allow scripts that use *create table* to complete silently even when the to-be-created table already exists.

```
-- Code_4
procedure Create_Table_Force(
  Owner_ID  in Identifier_t := null,
  Table_ID  in Identifier_t,
  Stmt_Tail in Stmt_t)
is
  Safe_Qualified_ID constant Identifier_t not null :=
    case
      when Owner_ID is null then
        Sys.DBMS_Assert.Simple_Sql_Name(Table_ID)
      else
        Sys.DBMS_Assert.Simple_Sql_Name(Owner_ID)||'.'||
        Sys.DBMS_Assert.Simple_Sql_Name(Table_ID)
    end;

  Drop_DDL constant Stmt_t :=
    'drop table '||Safe_Qualified_ID;

  Cr_DDL constant Stmt_t :=
    'create table '||Safe_Qualified_ID||' '||Stmt_Tail;
begin
  declare
    Table_Doesnt_Exist exception;
    pragma Exception_Init(Table_Doesnt_Exist, -00942);
  begin
    execute immediate Drop_DDL;
  exception when Table_Doesnt_Exist then
    null;
  end;
  execute immediate Cr_DDL;
end Create_Table_Force;
```

The procedure is defined in an invoker's rights package that also declares the subtypes. Notice these features:

» The procedure allows the actual argument for the *Owner_ID* formal parameter to be omitted. Here's an example of its use:

```
begin
  Pkg.Create_Table_Force(
    Table_ID=>'x',
    Stmt_Tail=>'(n integer primary key)');
end;
```

» The *case* expression that sets the value for *Safe_Qualified_ID* has the same syntax and semantics in PL/SQL as it does in SQL. The same is true for the text values that are established by concatenating literal values with variables.

» The value of every one of the variables, *Safe_Qualified_ID*, *Drop_DDL*, and *Cr_DDL* is assigned in the declaration that establishes each as constant. This is a powerful locution for aiding clarity; the reader can be confident that the assigned values are immutable.

» The *ORA-00942* error is mapped to a user-defined PL/SQL exception so that it can be caught and handled without the burden of testing error codes and using *if... then... else* constructs.[23] This is done in an inner block statement, taking advantage of another feature inherited from Ada.

Bulk SQL

Both static SQL and native dynamic SQL have bulk variants that use compact and clear syntax. *Code_5* shows the bulk static SQL equivalent of the *cursor for loop* shown in *Code_2 on page 12* with the refinement that the pagination syntax is added and the subprogram is recast, more realistically, as a function.

```
-- Code_5
function f(
  Lo in t.c3%type, Hi in t.c3%type, Offset in integer, Page_Size in integer)
  return Rows_t
is
  Rows Rows_t;
begin
  select t.c1, t.c2
  bulk collect into Rows
  from t
  where t.c3 between f.Lo and f.Hi
  order by PK
  offset Offset rows fetch next Page_Size rows only;
  return Rows;
end f;
```

The function *f()* is designed to be invoked by outside-of-the-database code using a *call* statement. The graphical user-interface code would capture *Lo*, *Hi*, and *Page_Size*, and would compute *Offset* by counting the number of times that *“Next Page”* has so far been selected. The invocation of the *call* statement would use an output actual of an appropriate datatype to match PL/SQL’s *Rows_t index-by-pls_integer table* of *record*. Significantly, because the pagination syntax shown here was brought by Oracle Database 12*c*, an earlier implementation of *f()* would use the well-known, but more cumbersome, *“Rownum slicing”* syntax. The implementation can be modernized without changing the signature of *f()* and without, therefore, needing any changes to outside-of-the-database code[24]. This is a text-book example of the wisdom of the modular approach to software construction.

Notice these features:

» The static SQL statement is spelled the same as it would be for the corresponding *cursor for loop* except for the presence of *bulk collect into Rows*. Interestingly, the PL/SQL compiler works in cooperation with the SQL compiler to generate the identically canonically spelled plain SQL statement text that it would for the corresponding *cursor for loop*:[25]

```
SELECT T.C1, T.C2
FROM T
WHERE T.C3 BETWEEN :B2 AND :B1
ORDER BY PK
OFFSET :B4 ROWS FETCH NEXT :B3 ROWS ONLY
```

» The hard parse avoidance and soft parse avoidance scheme is identical for bulk static SQL as it is for non-bulk static SQL.

» The *bulk collect into Rows* clause can be followed by the *Limit* clause. This is used within a surrounding infinite loop when the result set is too large to be comfortably accommodated in its entirety in session memory. However, the use of the pagination syntax ensures that the result set is manageably small so that the entire result set can be fetched in a single operation.

23. We shall return to this point in the section *“Oracle errors are magically transformed to PL/SQL exceptions”* on page 16.

24. I do realize that this example embodies a white lie because the ability to bind to an *index-by-pls_integer table* of *record* is new in Oracle Database 12*c*. But the generic point is still well-made.

25. I added line breaks to make the SQL text more readable.

Code_6 shows the bulk native dynamic SQL version of the bulk static SQL shown in *Code_5*.

```
-- Code_6
function f(
  Table_ID in Identifier_t,
  Lo in t.c3%type, Hi in t.c3%type, Offset in integer, Page_Size in integer)
  return Rows_t
is
  Rows Rows_t;
  Safe_Table_ID constant Identifier_t :=
    Sys.DBMS_Assert.Simple_Sql_Name(Table_ID);
  Stmt constant Stmt_t := '
    select t.c1, t.c2
    from '||Safe_Table_ID||' t
    where t.c3 between :b1 and :b2
    order by PK
    offset :b3 rows fetch next :b4 rows only';
begin
  execute immediate Stmt
  bulk collect into Rows
  using in f.Lo, in f.Hi, in f.Offset, in f.Page_Size;
  return Rows;
end f;
```

The hard parse avoidance and soft parse avoidance schemes for bulk native dynamic SQL are identical to those for bulk static SQL in just the same way that the schemes for non-bulk native dynamic SQL and non-bulk static SQL are identical. Similarly, the *Limit* clause is optional for bulk native dynamic SQL just as it is for bulk static SQL.

Oracle errors are magically transformed to PL/SQL exceptions

The OCI subroutine library is implemented in C, and C has no formal exception notion. When outside-of-the-database programs use the OCI, they must test a return code explicitly to detect if an Oracle error has occurred, and if it has, test further to detect the particular error code. This leads to tedious and error-prone coding patterns to unwind the stack until a subroutine is reached that knows how to react to the detected error. In contrast, PL/SQL inherits a formal exception notion from Ada. The PL/SQL runtime system looks after detecting when a SQL statement that it executes terminates with an Oracle error and, when it does, it automatically presents this as an exception[26]. Such an exception can always be caught by an *others* handler, and its identity can be determined by testing the value returned by the *SqlCode()* function; but such an approach would simply turn down the benefit of native exception handling. Rather, for Oracle errors that the developer will be able to handle, the best practice is to map each one to a declared exception using the *Exception_Init()* pragma. We saw this in the section *"Native dynamic SQL"* on page 12 with the declared *ORA-00942* exception. Some exceptions, like *No_Data_Found* and *Too_Many_Rows* are already defined in the *Standard* package.

26. This section's title, *PL/SQL's unique specific features for making SQL happen*, implicitly over-sells PL/SQL's automatic transformation of Oracle errors into exceptions. Some other languages do the same. Java is one example.

This feature of how PL/SQL does SQL allows a clutter-free coding style: no code is written to handle unexpected errors except in each of the subprograms that define the database interface.[27] *Code_7* shows the boilerplate code that should be used at the end of every such subprogram.

```
-- Code_7
-- The implementation of Some_Interface_Defining_Suprogram
   ...
  commit;
exception when others then
  declare
    Incident_ID pls_integer;
  begin
    -- Ensure business function atomicity.
    rollback;
    Incident_ID := Log_Errors(...);
    Raise_Application_Error(-20000,
     'Sorry, something went wrong. '||
     'Call Support and quote Incident_ID '||Incident_ID);
  end;
end <Some_Interface_Defining_Suprogram>;
```

This coding standard ensures that when an unexpected exception occurs, no information about the internals of the database module is leaked to outside-of-the-database entities. Such leakage can help hackers design their exploits. The atomicity of the business function implemented by such a subprogram (see *"Business function atomicity"* on page 9) depends critically depends on the explicit invocation of *rollback* as the first executable statement in the *others* handler. Similarly, the subprogram's executable section must end with *commit*; and no subprogram except those that define the database interface may issue *commit*. Strictly speaking, *commit* and *rollback* are needed as shown only when the interfacing-defining subprogram makes table changes; but they are harmless in such subprograms make no table changes.

The *authid* mode

The distinction between the definer's rights and invoker's rights *authid* modes is meaningful only for code that is stored in an Oracle Database and are therefore owned by a database user. We have already decided that PL/SQL is much better suited than Java for doing SQL and so we shall consider only PL/SQL units in this section. The *authid* mode non-negotiably governs the privilege regime in which the code executes; and it governs the regime in which unqualified names are resolved. The explanation of the difference between definer's rights and invoker's rights rests on the notion of the *Current_User*. When a session is idle, waiting for the next database call, the value of the *Current_User* is identical to that of the *Session_User*.[28] The value of *Session_User* is fixed, for the entire lifetime of the session, as the database user that created that session; but the value of *Current_User* can change. When, during a database call, the point of execution moves, either on making a call or on returning from a call, into a definer's rights unit, the value of *Current_User* is set to the owner of that unit. In contrast, when the point of execution moves into an invoker's rights unit on making a call, the value of *Current_User* remains unchanged. This means that when the point of execution is in a invoker's rights unit, the value of *Current_User* might be different from the owner of the unit. The SQL statements issued from a PL/SQL unit execute in the privilege regime defined by the direct grants that have been made to the *Current_User*. Any privileges that the *Current_User* or the *Session_User* might have via a role are ignored except for those that have been granted directly to *public*.[29] When the *call* statement that

27. Some developers prefer always to implement an *others* handler for every subprogram so that it can log information that is in scope only there. Then they re-raise the unexpected exception. This allows context sensitive diagnostic information to be captured at each level of the stack as it unwinds. This pattern still brings the advantage of clutter-free coding because the *others* handlers separate diagnostic code from the mainstream implementation of the ordinary logic.

28. I spell *Current_User* and *Session_User* this way because their values are given by *Sys_Context('Userenv', 'Current_User')* and *Sys_Context('Userenv', 'Session_User')*.

implements the top-level database call invokes an invoker's rights unit, its SQL statements execute with the privileges of the session user and their name resolution is done in the session's current schema.

Having removed Java from the discussion, we see that the definer's rights notion is effectively unique to database PL/SQL. This is critical for enforcing the regime described in the section *"The Thick Database paradigm"* on page 7. Only by using a schema design like that section describes, and by using definer's rights PL/SQL, can it be guaranteed that outside-of-the-database code has no ability to see the application's tables[30] and is able only to invoke the PL/SQL subprograms that define the database's interface.

The value of invoker's rights PL/SQL is appreciated by considering subprograms like the *Create_Table_Force()* procedure shown in *Code_4*. Here the aim is provide a convenience wrapper around powerful facilities, like creating a table or creating, altering, or dropping a database user, to support doing these operations according to recommended practices in a particular development shop. The developer must, having created a session as the database user in question, be able to do the required operation directly using the primitive DDL statements in order to be able to use the invoker's rights unit for its intended purpose.

In summary, an invoker's rights unit bestows no extra privilege to the database user that is the *Current_User* at the moment it is invoked. But a definer's rights unit can do this so that it can mediate actions, in a tightly controlled way, that the database user that is the *Current_User* at the moment it is invoked is unable to achieve using ordinary SQL statements.

No *make*: the promise brought by static dependency tracking

The source code of a PL/SQL unit typically makes lots of static references to other database objects. These might be to other PL/SQL units; or they might be, for example, to tables or views mentioned in static SQL statements or anchored declarations. Other types of object come into this picture. Such a static reference creates a dependency that can be seen in the *DBA_Dependencies* view, and each immediate dependency parent typically has its own immediate dependency parents. A PL/SQL unit therefore almost always has an appreciably large closure of immediate and transitive dependency parents. A PL/SQL unit can be valid only if each of its dependency parents, in the entire transitive closure, is valid; additionally, of course, its source must not have any syntax or semantic errors. An attempt to use an invalid PL/SQL unit will cause an Oracle error.

The Oracle Database implements an invalidation mechanism. Whenever an object is changed or dropped, each of its immediate dependants stands to be invalidated. Sometimes, the fine-grained dependency tracking scheme allows some, or all, dependants to remain valid; but this outcome uses a very rigorous proof that the change to the dependency parent is innocuous with respect to a dependant that remains valid. Safety is never compromised. Whenever an object is consequentially invalidated in this way, then every object in its closure of dependants is invalidated. The total effect of changing or dropping an object is immediately seen in the *DBA_Dependencies* view.

All this implies that a PL/SQL unit can be used only when it has been compiled against the current definition of its entire closure of dependency parents. There is no visible notion of linking. When the point of execution moves from one unit to another, units are simply loaded for execution on demand.

This scheme can be understood as an automatic, and completely reliable, equivalent of the regime that the use of utilities like UNIX's *make* delivers for other kinds of programming projects. And it comes with no developer

29. Oracle Database 12*c* brings an enhancement in this space that allows a role to be granted to a PL/SQL unit. This provides additional benefit beyond what this section describes. It isn't necessary, for the purpose of this paper, to understand the details of this new scheme.

30. Nor can outside-of-the-database code change the implementation of the PL/SQL interface. It can't even view it in *All_Source*.

effort beyond just typing up the various individual object definitions as scripted DDL statements. The enormous value that this scheme brings to PL/SQL developers seems often to be overlooked.

Developer productivity

The fact that PL/SQL runs in an Oracle Database means that all sorts of formal and *ad hoc* developer productivity tools are exposed by PL/SQL subroutines, dictionary views, and performance views.

General metadata

All relevant facts about all application-specific PL/SQL units, all Oracle-supplied PL/SQL units, and all their dependency parents are available. *DBA_Objects* lists basic facts like the owner of the object, its type, when it was created and last modified, and its validity. *DBA_Source* lists the PL/SQL source text. *DBA_Procedures* and *DBA_Arguments* list further specific properties of PL/SQL units, the subprograms within them, and their formal parameters. These facts are derived by compilation. The PL/Scope feature allows the developer to request that extra metadata be derived at compilation time. This is exposed by *DBA_Identifiers* and shows the per-unit line and column location all the identifiers within the corpus of PL/SQL source code of interest, the kind of item that the identifier denotes (subprogram, variable, exception, and so on), and the kind of the usage (declaration, definition, reference, and so on). This supports reliable, mechanical impact analysis. The value of *DBA_Dependencies* has already been mentioned. Moreover, everything else about the environment in which the PL/SQL has been compiled is listed in other views. Obvious examples are *DBA_Tables*, *DBA_Indexes*, *DBA_Constraints*, *DBA_Views*, *DBA_Sequences*, and *DBA_Triggers*. The list of useful dictionary views goes on.

In addition to the complete static definition of the application's Oracle Database backend comes a wealth of data about its run-time behavior. Some is available on an always-on, sampling basis; and some is made available by turning on various tracing facilities.

Compiler warnings

Many programming languages implement compiler warnings. The set of PL/SQL warnings includes ones that are specific to its use of SQL. For example, when the datatype of a PL/SQL bind actual is different from that of the table column denoted by the placeholder to which it is bound, a warning will be drawn because this usage can lead to a suboptimal execution plan. (The suboptimality occurs because the necessary implicit datatype conversion can mean that an index that would boost performance cannot be used.) The requested warnings level, together with any warnings that are drawn, are stored persistently for each unit (and exposed by the *DBA_Plsql_Object_Settings* and *DBA_Errors* views) until the next compilation. This enables mechanical policing, implemented in PL/SQL of course, of development shop standards.

Tracing

The classical use of the "print statement" to trace program execution and diagnose problems still continues to be a very popular first approach, and it often turns out to be sufficient. It has the obvious appeal that output can be produced selectively, both with respect to source code location and by conventional testing of relevant values, for example only every *N*th loop iteration, or only when a value exceeds a threshold. PL/SQL's conditional compilation allows such tracing code to remain permanently in place. This kind of tracing is often helped by printing out the call stack. The *Utl_Call_Stack* package gives the developer considerable flexibility to compose the most appropriate *"where am I?"* description. The *DBMS_Trace* package allows non-invasive tracing at controllable levels of granularity. And the interactive debugging tool, exposed by Oracle SQL Developer, supports the conventional set of non-invasive features (set breakpoint, run to next breakpoint, step over, step into, and so on) from a graphical user-interface.

Performance analysis

The developer uses the hierarchical performance profiler (mediated by the *DBMS_Hprof* package) to find the major time-consumers. This tool sees the SQL statements that the PL/SQL issues as just another kind of subprogram and reports homogeneously on the two kinds of subprogram. Very often, of course, the SQL statements will be the major time-consumers. When the hierarchical performance profiler shows that the major time-consumer is a PL/SQL subprogram, and when ordinary inspection of the source code isn't enough to spot the problem, the statement-oriented profiler (mediated by the *DBMS_Profiler* package) provides mechanical assistance for the within-subprogram analysis. Each of these profilers outputs the data from a run to ordinary tables. The programming skills needed to orchestrate profiling runs and to analyze their results are the same skills that the developer uses to write the code under analysis: PL/SQL and SQL.

Oracle SQL Developer

Oracle SQL Developer is a modern IDE that allows the graphical definition and interrogation of everything that defines the application's entire Oracle Database backend and its run-time behavior.

Using conditional compilation to compare candidate alternative approaches in situ

During development, a second candidate approach is often generated by ordinary file copy and subsequent minor editing. Sometimes more than two candidates are created this way. It quite often happens that the exercise of comparing the candidates leads to a realization the an improvement should be made in the code that has not been differentiated. It's time-consuming and error-prone to make the same changes in several places. Conditional compilation solves this problem by allowing the sections of differentiated candidate code to remain in place in with the typically much larger volume of non-differentiated code in one single file. This has the huge advantage of making the differentiation immediately apparent and ensuring that the non-differentiated code is singly defined. In general, this scheme cannot be achieved by run-time tests because the differentiated candidates need different declarations for items that must retain their pre-determined names. Because PL/SQL's conditional compilation can test a package global constant, differentiation that needs to be done in lock-step across several PL/SQL units can be singly controlled. This scheme is particularly valuable when the choice of candidate will be determined by a performance test. The experiment can be very straightforwardly mechanized and therefore easily repeated; and only repeatability can bring confidence in the results of a performance comparison experiment.

In this scenario, when the winner has been identified, the interim expedient use of conditional compilation constructs will simply be removed. In another scenario, where code must be selected according to, say, the version of the Oracle Database, because only in the later version is new syntax supported, the conditional compilation constructs remain part of the ultimate production code.

Deploying PL/SQL code changes with zero downtime

Edition-based redefinition (hereinafter EBR)[31] allows deployed PL/SQL units to be changed, dropped, or created, in the production database, without interrupting the application's availability. You simply create a new edition – which has the semantic effect that all PL/SQL units, views, and synonyms are instantaneously copied into the new edition – and then run the same DDL statements in the new edition that you would have run to do the patching in downtime. The implementation uses a copy-on-write mechanism so that only the objects you change consume space. When you have ensured that the new edition represents the changes you intended,

31. EBR was brought by Oracle Database 11*g* Release 2; Oracle Database 12*c* brought significant enhancements to make it easier to adopt. EBR has a much broader applicability than is sketched in this paper: it allows changes to be made to any of the artifacts that jointly implement an application's database backend without interrupting availability. For example, it allows changes to table structures and changes to the representation of data held in tables. Learning resources are listed on the OTN page for EBR at *oracle.com/ebr*.

then you start to take new server calls in the new edition. EBR makes it easier to hot patch code inside the database than to do this for code that runs outside of the database.

Using PL/SQL to empower developers to provision their own databases

The Thick Database paradigm brings a clear notion of the Oracle Database Development Engineer job role. Such an engineer needs to be fluent in both SQL and PL/SQL and to take the stance that, when hierarchically decomposing a PL/SQL subprogram, SQL is one way to implement a subprogram in the decomposition. The role needs the usual software engineer skills specialized to the Oracle Database: information modeling and table design, set theory, the SQL language, execution plans and all that this implies about the use of indexes and statistics, the PL/SQL language, and the various tools that come with these two languages (dictionary views, performance views, tracing, profiling, and so on). Notably, administrator skills, like backup and recovery, Data Guard, RAC deployment, how to patch and upgrade an installed Oracle Database, and so on, are not essential.

The responsibility of the Oracle Database Development Engineer is everything that is hidden behind the database interface, and nothing that is outside of the database.

Therefore all that the Oracle Database Development Engineer needs in order to work is an appropriately populated database for the task in hand, a text editor, and a tool than can interpret the SQL*Plus scripting language.

The Mulitenant architecture, brought by Oracle Database 12*c*, lets each developer have one or several dedicated pluggable databases (hereinafter PDBs). When the container database is installed on a suitable filesystem, the sparse clone feature allows the very rapid creation of a new PDB – and with a tiny space requirement. All the primitive operations for PDB provisioning are exposed by SQL statements and these, of course, can be encapsulated by a PL/SQL subprogram to expose higher-order functionality, like cloning a so-called gold PDB, when this is authorized, in a controlled fashion with respect to degrees of freedom like filesystem location for the datafiles, and so on. It is then very straightforward to build a small developer-shop-specific application, with suitable metadata for each developer like the set of gold PDBs that the developer is allowed to clone, the total space that all the PDBs owned by a developer can use, and so on.

Oracle Database Development Engineers can then take provisioning into their own hands using an interface to these tasks exposed by PL/SQL – their stock-in-trade. They can clone the gold PDB of interest, deploy various changes within it, run tests that change the data, drop the clone, and start again. Of course, the whole process can be mechanized, using the SQL*Plus scripting language. This means that, after making small changes to the installation scripts of interest, the effect can be fully mechanically tested by a single command. Therefore the change-and-test cycle that characterizes the software engineer’s job can be executed in just minutes – even when a test might destroy the database.

This improves productivity in the same way that this transition did for programming four decades ago:

» *from* the batch submission of jobs on punched cards, hours of waiting for the compilation and test run, reams of line printer output, and a follow-up round of tedious punch card editing so that the cycle could be repeated

» *to* interactive WYSIWYG text editors and immediate compilation and testing.

The difference in degree becomes a difference in kind.

Conclusion

A serious application whose charter is to persist and retrieve critical data is valueless unless the data is persisted correctly and unless queries retrieve exactly what is requested.

PL/SQL is better suited to the task of executing SQL statements and processing their results than any other programming language that can do this. It brings these significant benefits:

» optimal expressiveness, maintainability, and therefore maximum chance of correctness

» optimal security

» optimal performance.

It's no coincidence that PL/SQL uniquely has these properties. They were defined specifically as the requirements that the language should meet at the time of its invention.

A database that is encased in a hard shell that exposes only the intended functionality using a small number of entry points better guarantees the correct persistence and retrieval of the data that it manages than one that exposes its inner blood and guts to outside-of-the-database entities.

I know of many applications whose developers adhere strictly to the Thick Database paradigm; and I know of many whose developers religiously follow the NoPlsql paradigm. End-users of the first kind of application are happy with the correctness, maintainability, security, and performance of their applications. End-users of the second kind routinely complain. There are performance problems because the execution of a single business transaction often involves many round trips from the application server module to the database module. There are long lead times for fixing problems and for bringing new functionality because even small patches imply large effort when there is no clear distinction between the concerns of the database "module" and the concerns of the outside-of-the-database "modules". Moreover, SQL injection vulnerabilities are more common when the NoPlsql paradigm is followed.

I am convinced that an application that uses an Oracle Database as its persistence mechanism has no special properties that recommend that its design, uniquely among an uncountable number of diverse software systems, should disregard the otherwise universally respected wisdom of modular software construction.

Bryn Llewellyn,
Distinguished Product Manager,
Database Server Technologies Division, Oracle Headquarters
bryn.llewellyn@oracle.com

Why use PL/SQL?
November 2016
Author: Bryn Llewellyn