Doing SQL from PL/SQL: Best and Worst Practices

An Oracle White Paper
September 2008

ORACLE

NOTE

The following is intended to outline our general product direction. it is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Doing SQL from PL/SQL: Best and Worst Practices

CONTENTS

Doing SQL from PL/SQL: Best and Worst Practices

ABSTRACT

The PL/SQL developer has many constructs for executing SQL statements, and the space of possibilities has several dimensions: embedded SQL versus native dynamic SQL versus the *DBMS_Sql* API; bulk versus non-bulk; implicit cursor versus parameterized explicit cursor versus ref cursor; and so on. Deciding which to use might seem daunting. Moreover, as new variants have been introduced, older ones sometimes have ceased to be the optimal choice. Oracle Database 11*g* has kept up the tradition by bringing some improvements in the area of dynamic SQL.

This paper examines and categorizes the use cases for doing SQL from PL/SQL, takes the Oracle Database 11*g* viewpoint, and explains the optimal approach for the task at hand.

INTRODUCTION

Make sure that you're reading the latest copy of this paper. Check the URL given at the top of each page.

This paper is written for the Oracle Database developer who has reasonable familiarity with programming database PL/SQL units and who, in particular, has had some experience with all of PL/SQL's methods for processing SQL statements. Therefore, it doesn't attempt to teach, or even to review, every variant of each of these methods; rather, by assuming some prior knowledge, it is able to make points that often go unmade in accounts that teach these methods linearly. This way, it is able to give the reader the sound conceptual understanding that is the basis of any and all best practice principles.

An analogy might help. Many people, as adults, pick up a foreign language by osmosis and, eventually, end up where they can express themselves fairly clearly but, nevertheless, use idioms that indicate that they have no deep understanding of how the language works. Sometimes, these idioms are so awkward that what they want to say is misunderstood. The remedy is active study to learn the grammar rules and the meanings that sentences that adhere to these rules convey.

The first section, *"Embedded SQL, native dynamic SQL and the DBMS_Sql API"* on page 5, gives an overview of PL/SQL's three methods for processing SQL statements.

The *select* statement is by far the most frequent kind of SQL statement issued by application code[1]. The second section, *"Approaches for select statements"* on page 30, classifies the use cases along these dimensions:

- Selecting many rows where the result set size might be arbitrarily large; selecting many rows where the result set size can be assumed not to exceed a reasonable limit; and selecting exactly one row.
- Being able to fix the SQL statement at compile time; being able to fix its template at compile time, but needing to defer specifying the table name(s) until run time; needing to construct the *select list*, *where* clause, or *order by* clause at run time.
- Being able to encapsulate the specification of the SQL statement, fetching of the results, and the subsequent processing that is applied to the results in a single PL/SQL unit, or needing to implement to processing of the results in a different PL/SQL unit.

After the *select* statement, SQL statements that change table data are the next most common. The third section, *"Approaches for insert, update, delete, and merge statements"* on page 47, discusses these.

The *lock table* statement, the transaction control statements, and all the kinds of SQL statement that embedded SQL does not support are trivial to program and need no discussion.

The fourth section, *"Some use cases"* on page 59, examines some commonly occurring scenarios and discusses the best approach to implement the requirements.

---

1. Our definition of application code excludes the scripts that install and upgrade it.

Several best practice principles will be stated in the context of the discussions in this paper. They are reproduced[2] for quick reference, in *"Appendix B: Summary of best practice principles"* on page 65.

The paper aims to teach the reader the optimal approaches to use when writing *de novo* code. It makes no attempt to justify code renovation projects.

Caveat

Prescribing best practice principles for programming any 3GL is phenomenally difficult. One of the hardest challenges is the safety of the assumption that the reader starts out with these qualities:

- Has chosen the right parents[3].
- Has natural common sense coupled with well-developed verbal reasoning skills.
- Has an ability to visualize mechanical systems.
- Requires excellence from self and others.
- Has first class negotiating skills. (Good code takes longer to write and test than bad code; managers want code delivered in aggressive timeframes.)
- Has received a first class education.
- Can write excellent technical prose. (How else can you write the requirements for your code, write the test specifications, and discuss problems that arise along the way?)

Then, the reader would be fortunate enough to work in an environment which provides intellectual succor:

- Has easy access to one or several excellent mentors.

Finally, the reader would accept that, with respect to the subject of this paper, the internalization and instinctive application of best practice principles depends, ultimately, on acquiring and maintaining these qualities:

- Knows Oracle Database inside out.
- Knows PL/SQL inside out.

Periodic revision of this paper

Sadly, but realistically, this paper is likely to have minor spelling and grammar errors. For that reason alone, it is bound to be revised periodically[4]. Ongoing

---

2. This paper was prepared using Adobe Framemaker 8.0. Its cross-reference feature allows the text of a source paragraph to be included by reference at the destination. The reader can be certain, therefore, that the wording of each best practice principle in the quick-reference summary is identical to the wording where is stated. (Sadly, the mechanism does not preserve font nuances.)

3. The author's mother-tongue is British English. Readers with other mother-tongues sometimes need reminding about the tendency, in the author's native culture, towards dead pan, tongue in cheek humor as a device to make a serious point in a dramatic fashion.

4. This document's change history is listed at the end. See *Appendix A: Change History* on page 64.

discussion of use cases with customers might lead to the formulation of new best practice principles. Therefore, before settling down to study the paper, readers should ensure that they have the latest copy — for which the URL is given in the page's header.

URLs sometimes change. But this one will always take you to the Oracle Technical Network's PL/SQL Technology Center:

www.oracle.com/technology/tech/pl_sql

Even in the unlikely event that the paper is moved, it will still be easy to find from that page.

EMBEDDED SQL, NATIVE DYNAMIC SQL AND THE *DBMS_SQL* API

PL/SQL supports three methods for issuing SQL. This section gives an overview and makes some points that are best appreciated by the reader who has some experience with each of the methods. A good understanding of all three methods is necessary for choosing the optimal method for a particular requirement.

Embedded SQL

PL/SQL's embedded SQL[5] allows SQL syntax directly within a PL/SQL statement and is therefore very easy to use[6]. It supports only the following kinds of SQL statement: *select*, *insert*, *update*, *delete*, *merge*, *lock table*, *commit*, *rollback*, *savepoint*, and *set transaction*.

The syntax of a PL/SQL embedded SQL statement is usually identical to that of the corresponding SQL statement[7]. However, the *select... into* statement, the *update... set row...* statement, and the *insert... values Some_Record* statement (see *Code_3*) have PL/SQL-specific syntax[8].

An embedded SQL statement has a significant semantic bonus with respect to its regular SQL counterpart: it allows a PL/SQL identifier at a spot where the regular SQL counterpart allows a placeholder. This is explained more carefully in the section *"Resolution of names in embedded SQL statements."* on page 6.

Code_1 shows a simple example.

```
-- Code_1
for j in 1..10 loop
  v1 := f(j);
  insert into t(PK, v1) values(j, b.v1);
end loop;
commit;
```

Here, *b* is the name of the block in which the variable *v1* is declared.

---

5. We prefer the term *embedded SQL* to the more usual *static SQL* in this document for reasons that will soon become clear.

6. It is this property that led to the aphorism "PL/SQL is Oracle Corporation's procedural extension to SQL". But this is too brief to be accurate. (It reflects history: the anonymous block, which is a particular kind of SQL statement, was introduced before the stored PL/SQL unit.) It is better to say this: PL/SQL is an imperative 3GL that was designed specifically for the seamless processing of SQL commands. It provides specific syntax for this purpose and supports exactly the same datatypes as SQL.

7. SQL famously has no notions to sequence statements; therefore a SQL statement does not need (and may not have) a special terminating character. In contrast, a PL/SQL unit consists of many statements and each must end with a semicolon. PL/SQL's embedded SQL statements therefore differ from their SQL counterparts by requiring a final semicolon. Beginners are sometimes confused because the SQL*Plus scripting language supports sequences of SQL statements intermixed with SQL*Plus commands. The scripting language therefore needs a special character to terminate each SQL statement. The default is the semicolon, but this may be overwritten with the `SET SQLTERMINATOR` command. All SQL*Plus script examples in this paper were prepared to run after issuing `SET SQLTERMINATOR OFF`. A notorious beginner's mistake is to write a semicolon at the end of a would-be SQL statement in the text that is to be the argument of *execute immediate*.

Code_2 shows[9] the canonical illustration of how PL/SQLs' language features make processing SQL statements simple and self-evidently correct, the so-called *implicit cursor for loop*[10].

```
-- Code_2
for r in (
  select    a.PK, a.v1
  from      t a
  where     a.PK > Some_Value
  order by  a.PK)
loop
  Process_One_Record(r);
end loop;
```

Code_3, shows some of the kinds of embedded SQL statement that have PL/SQL-specific syntax.

```
-- Code_3
<<b>>declare
  Some_Value t.PK%type := 42;
  The_Result t%rowtype;
begin
  select    a.*
  into      b.The_Result
  from      t a
  where     a.PK = b.Some_Value;

  The_Result.v1 := 'New text';

  update  t a
  set     row = b.The_Result
  where   a.PK = b.The_Result.PK;

  The_Result.PK := -Some_Value;
  insert  into t
  values  The_Result;
end;
```

Resolution of names in embedded SQL statements.

When the PL/SQL compiler discovers an embedded SQL statement, it processes it as follows:

- It hands it over to the SQL subsystem for analysis[11]. The SQL subsystem establishes that the statement is syntactically correct (else the compilation of the PL/SQL unit fails) and it discovers the names of the *from list* items and attempts to resolve all other identifiers within their scope.
- When an identifier cannot be resolved in the scope of the SQL statement, it "escapes" and the PL/SQL compiler attempts to resolve it. It tries first within

---

8. The *where current of Cur* syntax (where *Cur* is an *explicit cursor*) is also not found in regular SQL. However, we shall see that this construct is never needed if the best practice principles that this paper recommends are followed.

9. The code examples in this paper use a test *table t*. Code to create it is shown in *"Appendix D: Creating the test user Usr, and the test table Usr.t(PK number, v1 varchar2(30), ...)"* on page 72

10. We will see in *"Selecting many rows — unbounded result set"* on page 30 that this simple construct is never preferred for production-quality code.

11. For a *select... into* statement, the PL/SQL compiler removes the *into* clause before handing the statement over to the SQL subsystem. It does the same with other PL/SQL-specific syntax in other kinds of embedded SQL statement like, for example, *set row*.

the scope of the current PL/SQL unit; if that attempt fails, then it tries in schema scope; and if that fails, the compilation of the PL/SQL unit fails.

- In the absence of a compilation error, the PL/SQL compiler generates the text of an equivalent regular SQL statement and stores this with the generated machine code. This statement will use placeholders where the embedded SQL statement used identifiers that were resolved in the scope of the PL/SQL unit.
- Then, at run time, appropriate calls are made to parse, bind, and execute the regular SQL statement. The bind arguments are provided by the escaping PL/SQL identifiers, which may be combined into expressions. For a *select* statement, the results are fetched into the designated PL/SQL targets.

Code_4 shows the generated regular SQL statement[12] for *Code_1*.

```
-- Code_4
INSERT INTO T(PK, V1) VALUES(:B2 , :B1 )
```

And *Code_5* shows the generated regular SQL statements for *Code_3*.

```
-- Code_5
SELECT A.* FROM T A WHERE A.PK = :B1

UPDATE T SET "PK" = :B1 ,
             "N1" = :B2 ,
             "N2" = :B3 ,
             "V1" = :B4 ,
             "V2" = :B5 WHERE PK = :B1

INSERT INTO T VALUES (:B1 ,:B2 ,:B3 ,:B4 ,:B5 )
```

Notice how much effort the PL/SQL compiler saves you. It discovers the shape of the *record The_Result* and the column format of the table and generates the regular SQL statement text accordingly. It canonicalizes case and whitespace to increase the probability that embedded SQL statements at different call sites will share the same structure in the shared pool. And comments in the embedded SQL are preserved in the generated SQL only if they use the /*+ ... */ hint syntax.

The PL/SQL identifier that will act as the bind argument for the generated SQL statement must denote a variable or a formal parameter; it may not denote a function that is visible only in the PL/SQL unit that includes the embedded SQL. This follows from the defined semantics of function invocation in a SQL statement: the function must be evaluated, by the SQL execution subsystem, for every row. Therefore an embedded SQL statement may use only functions that are accessible in schema scope. *Code_1* shows how to accommodate this fact.

---

12. *Code_4* and *Code_5* are discovered using a query like this:

```
select   Sql_Text
from     v$Sql
where    Lower(Sql_Text) not like '%v$sql%'
and     (Lower(Sql_Text) like 'select%a.*%from%t%' or
         Lower(Sql_Text) like 'update%t%set%' or
         Lower(Sql_Text) like 'insert%into%t%')
```

Code_5 has been formatted by hand to make it easier to read.

Name capture, fine grained dependency tracking, and defensive programming

The identifiers in *Code_3* might seem to be excessively decorated with qualifiers. It stands in contrast to a common way, shown in *Code_6*, to implement the same functionality.

```
-- Code_6
select  v1
into    l_v1
from    t where PK = p_PK;
```

Some programmers have become attached to a style where they name local variables and formal parameters using a prefix or suffix convention that denotes this status. (Some go further and use names that distinguish between in different parameter modes — *in*, *out*, and *in out*.) They usually claim that the style inoculates against the risk of name capture. The scenario they fear is that, in *Code_6*, table *t* will be altered to add a column called, say, *c1*. Because the SQL compiler attempts first to resolve *c1*, and allows it to escape to the PL/SQL compiler only when it isn't resolved, then the addition of the new *c1* column might change the meaning of *Code_6* by stopping that escape. This would depend on whether the identifier *c1* earlier had been resolved after escaping — in other words, if it was already used in the embedded SQL statement. Had it been so used, then the name *c1* is now captured by SQL. When *c1* is not qualified, then the PL/SQL unit must be recompiled to establish correctness.

Notice that, if each identifier that is intended to mean a table column is qualified with a table alias, and if each identifier that is intended to be resolved in the scope of the PL/SQL unit is qualified with the name of the block where it is declared, then no room is left for uncertainty.

When explained this way, it is obvious that the programming style cannot guarantee to deliver its intended benefit. In the present example, the table *t* might be altered to add a column called *p_PK*. The style can work only if a development shop insists on a wider naming convention that, for example, bans columns in schema-level tables with names starting with *p_* or *l_*. However, the PL/SQL compiler cannot trust such a humanly policed rule, and this fact becomes more significant in Oracle Database 11*g* which brings fine grained dependency tracking.

Earlier, dependency information was recorded only with the granularity of the whole object. In the example in *Code_6*, it would be recorded just that the current PL/SQL unit depends on the table *t*. Now, in Oracle Database 11*g*, it is recorded that the PL/SQL unit depends on the columns *t.v* and *t.PK* within table *t*. The new approach aims to reduce unnecessary invalidation by avoiding it when a referenced object is changed in a way which is immaterial for the dependant. In this example, the beginner might think at first that the addition of a new column to *t*, when the dependant PL/SQL unit refers to only certain named columns in *t*, would be immaterial. But the discussion of name capture shows that this is not always the case: the name of the new column might collide with what used to be an escaping identifier that was resolved in PL/SQL scope. The only way to guarantee that the PL/SQL has the correct meaning in the regime of the altered table is to invalidate it in response to the addition of the new column so that it will be recompiled and the name resolution will be done afresh.

The use of qualified names, as used in *Code_3*, changes the analysis. The qualified identifier *b.PK*, cannot possibly mean a column (existing or new) in the table whose alias in the query is *a*. This is easily confirmed by experiment. Create table *t* with columns *PK* and *n*; procedure *p1* containing *Code_3* and procedure *p2* containing *Code_6*. Confirm, with a *User_Objects* query, that both are valid. Then alter *t* to add a column (for example c*1* of datatype *number*) and repeat the *User_Objects* query; *p1* remains valid but *p2* becomes invalid[13].

Finally, consider the counter-example shown in *Code_7*.

```
-- Code_7
<<b>>declare
  Some_Value t.PK%type := 42;
  The_Result t%rowtype;
begin
  select    b.*
  into      b.The_Result
  from      t b
  where     b.PK = b.Some_Value;

  DBMS_Output.Put_Line(The_Result.n1);
end;
```

Here, foolishly, the alias for table *t* collides with the name of the PL/SQL block. This code might well behave correctly while *t* has no column called *Some_Value*. But if such a column were introduced, then the meaning of the query would change to what is almost certainly not intended. In other words, the code is not immune to name capture and neither is it able to take advantage of fine-grained dependency tracking.

The understanding of how the PL/SQL compiler processes embedded SQL statements, and in particular of how the risk of name capture arises, provides the rationale for this best practice principle:

In embedded SQL, dot-qualify each column name with the *from list* item alias. Dot-qualify each PL/SQL identifier with the name of the name of the block that declares it.

Principle_1

When writing an embedded SQL statement, always establish an alias for each *from list* item and always qualify each column with the appropriate alias. Always qualify the name of every identifier that you intend the be resolved in the current PL/SQL unit with the name of the block in which it is declared. If this is a block statement, give it a name using a label. The names of the aliases and the PL/SQL bocks must all be different. This inoculates against name capture when the referenced tables are changed and, as a consequence, increases the likelihood that the fine-grained dependency analysis will conclude that the PL/SQL unit need not be invalidated.

Ultimately, all SQL issued by a PL/SQL program is dynamic SQL

At run time, the SQL statement that has been generated from each embedded SQL statement is executed in the only way that the SQL subsystem in Oracle Database supports: a *session cursor*[14] is opened; the SQL statement is presented as text and is parsed; for a *select* statement, targets for the *select list* elements are defined; if the SQL statement has placeholders, then a bind argument is bound to each; the *session cursor* is executed; for a *select* statement, the

13. Of course, *p2* is easily revalidated without any code changes. But, in the general case, its meaning might change on revalidation while that of *p1* cannot.

14. This term of art is defined on page 19.

results are fetched; and the *session cursor* is closed[15]. When a PL/SQL program uses dynamic SQL, the run-time processing of the SQL statement is identical. The only difference is in who did the "thinking" needed to produces the run-time code: for native dynamic SQL, and especially for the *DBMS_Sql* API, the programmer does the work; for embedded SQL, the PL/SQL compiler does the work.

Beginners, and especially programmers who are meeting dynamic SQL for the first time and who have had some experience with embedded SQL, do not always realize this. For example, it might appear (to a beginner) that if an embedded SQL statement compiled successfully, then the SQL statement that is generated from it is bound not to fail[16] at run time. A moment's thought reveals the fallacy: when an invoker's rights PL/SQL unit[17] runs, the *Current_Schema* is set either to the *Session_User* (when only invoker's rights PL/SQL units are on the call stack) or to the *Owner* of the definer's rights PL/SQL unit[18] or view that is closest on the stack. This means, for example, that an unqualified identifier in the generated SQL statement might be resolved differently at run time than at compile time[19]. In the worst case, this can result in the *ORA-00942: table or view does not exist* run-time error[20].

Embedded SQL is more expressive than some programmers realize

We occasionally see code (especially when it has been written by a beginner and has not been reviewed by an experienced PL/SQL programmer) where native dynamic SQL or the *DBMS_Sql* API has been used to implement a SQL requirement that could have been met with embedded SQL. There is almost never any convincing reason to avoid embedded SQL when it its expressivity is sufficient; such avoidance is undoubtedly a worst practice. (We will return to this point in the next section.)

---

15. When a *select list* is not known until run time, and is not constructed directly by the PL/SQL program, then an extra step is needed. The PL/SQL program asks the SQL system to describe the *select list*. This requires the use of the *DBMS_Sql* API. However, this use case is exceedingly rare in production PL/SQL programs; an implementation design that proposes this approach should be viewed with extreme suspicion.

16. Here, the notion of failure is defined to exclude data-driven conditions like the *Dup_Val_On_Index* exception for *insert* and the *No_Data_Found* and *Too_Many_Rows* exceptions for *select... into*.

17. An invoker's rights PL/SQL unit is one where the *authid* property is equal to *Current_User*.

18. An definer's rights PL/SQL unit is one where the *authid* property is equal to *Definer*.

19. This is the reason that successful compilation of an invoker's rights PL/SQL unit sometimes requires so-called template objects in the schema owned by the PL/SQL unit's *Owner*. The rules for name resolution and for privilege checking that are used at PL/SQL compile time are identical for invoker's rights and definer's rights PL/SQL units.

Native dynamic SQL

Code_8 shows a simple example of native dynamic SQL. The use of the word *native* in the name of the method denotes the fact that it is implemented as a PL/SQL language feature.

```
-- Code_8
procedure p authid Current_User is
  SQL_Statement constant varchar2(80) := q'[
    alter session
       set NLS_Date_Format = 'dd-Mon-yyyy hh24:mi:ss'
    ]';
begin
  execute immediate SQL_Statement;
  DBMS_Output.Put_Line(Sysdate());
end p;
```

Notice that the *alter session* SQL statement is not supported by embedded SQL and so the use of a method where the PL/SQL compiler does not analyze the SQL statement (as *Code_9* dramatically demonstrates) is mandated. The term *dynamic SQL* is universally used to denote such a method, but the word "dynamic" is arguably misleading. It was chosen because the text of the SQL statement that is executed this way *may* be constructed at run time; but, of course, it need not be. In *Code_8*, the SQL statement is fixed at compile time; this is emphasized by the use of the *constant* keyword.

A generic PL/SQL best practice principle urges this:

Declare every PL/SQL variable with the *constant* keyword unless the block intends to change it.

Principle_2

Use the *constant* keyword in the declaration of any variable that is not changed after its initialization[21]. Following this principle has no penalty because the worst that can happen is that code that attempts to change a *constant* will fail to compile — and this error will sharpen the programmer's thinking. The principle has a clear advantage for readability and correctness[22].

In the context of dynamic SQL, it is especially valuable to declare the text of the SQL statement as a *constant* when this is possible because doing so reduces the surface area of attack for SQL injection[23].

Notice, in this connection, that *p*'s *authid* property is explicitly set to *Current_User*. If the *authid* clause is omitted, then the default value for the property, *Definer*, is

---

20. A definer's rights PL/SQL unit always sees just those privileges that have been granted explicitly to its *Owner* together with those that have been granted explicitly to *public*. An invoker's rights PL/SQL unit sees privileges that depend on the state of the call stack. When a definer's rights PL/SQL unit is on the stack, then the invoker's rights PL/SQL unit sees exactly the same privileges as the definer's rights PL/SQL unit, or view, that is closest on the stack. When only invoker's rights PL/SQL units are on the call stack, each sees those privileges that the *Current_User* has directly and those that the *Current_User* has via *public* together with those that the *Current_User* has via all currently enabled *roles*. Therefore, even when an invoker's rights PL/SQL unit identifies an object using a schema-qualified name, *ORA-00942* might still occur at run time.

21. It is possible for the PL/SQL compiler to detect this case (except when the variable is declared at global level in a package spec). Enhancement request 6621216 asks for a compiler warning for this case.

22. Using the *constant* keyword can, under some circumstances, tell the PL/SQL compiler that particular optimizations are safe where, without this information, they would have to be assumed to be unsafe.

used. This behavior is determined by history and cannot be changed for compatibility reasons. However, programmers should ignore the fact that the property has a default value and adopt this best practice principle:

Always specify the *authid* property explicitly. Decide carefully between *Current_User* and *Definer*.

Principle_3

Always specify the *authid* property explicitly in every PL/SQL unit; choose between definer's rights and invoker's rights after a careful analysis of the purpose of the unit[24].

Code_9 shows a counter-example of native dynamic SQL.

```
-- Code_9
procedure p(Input in varchar2) authid Current_User is
  SQL_Statement constant varchar2(80) := 'Mary had... ';
begin
  execute immediate SQL_Statement||Input;
end p;
```

This version of procedure *p* compiles without error; but at run time (in this example, irrespectively of what actual argument is used) it fails with *ORA-00900: invalid SQL statement*. This is a dramatic way to make an obvious observation and to provide the context to discuss a less obvious one; the discussion motivates some comments on best practice principles.

Obviously, the SQL statement is not analyzed until run time; it is this property of the method that allows it to support the kinds of SQL statement that embedded SQL does not. In the general case, the text of the SQL statement is not known until run time. As a consequence, a SQL statement that is executed may access objects that do not exist at compile time but that, rather, are created between compile time and when the SQL statement is executed. Before the introduction of the global temporary table,[25] PL/SQL programs sometimes used scratch tables to accommodate the overflow of large volumes of transient data. Naïve implementations would create, and later drop, the scratch table using dynamic SQL. More sophisticated implementations would reserve, and later relinquish, the name of an available such table from a managed pool. Either way, the program could not know the name of the table until run time and so

---

23. A detailed discussion of SQL injection is beyond the scope of this paper. It suffices here to say that a PL/SQL program that issues only SQL statements whose text is fixed at compile time is proof against the threat — and to point out that both embedded SQL and dynamic SQL that uses *constant* SQL statement text have this property. Of course, it is the responsibility of a PL/SQL programmer who writes code that issues SQL to ensure that this is not vulnerable to SQL injection. A companion paper, *How to write injection-proof PL/SQL*, is posted on the Oracle Technology Network here:
www.oracle.com/technology/tech/pl_sql/how_to_write_injection_proof_plsql.pdf
Its careful study is very strongly encouraged.

24. Enhancement request 6522196 asks for a compiler warning when the *authid* property is not explicitly specified. We expect this to be implemented in Oracle Database 11*g* Release 2.

25. The global temporary table is now supported in all supported versions of Oracle Database.

dynamic SQL was required even for those ordinary SQL statements that are supported in embedded SQL. *Code_10* shows a stylized typical example.

```
-- Code_10
procedure b(The_Table in varchar2, PK t.PK%type)
  authid Current_User
is
  Template constant varchar2(200) := '
    select a.v1 from &&t a where a.PK = :b1';
  Stmt constant varchar2(200) := Replace(
    Template, '&&t',
    Sys.DBMS_Assert.Simple_Sql_Name(The_Table));
  v1 t.v1%type;
begin
  execute immediate Stmt into v1 using PK;
  ...
end b;
```

The use of a *constant* template[26], and the derivation from it of the *constant* intended SQL statement, are devices to make the code self-evidently correct.

Because this paper's focus is best and worst practices, it cannot afford to show (except as clearly advertised counter-examples) code that is anything other than exemplary. This explains the use of *Sys.DBMS_Assert.Simple_Sql_Name()*. The name is qualified in accordance with a generic SQL and PL/SQL best practice principle:

Use the *Owner* to dot-qualify the names of objects that ship with Oracle Database.

Principle_4

References to objects that Oracle Corporation ships with Oracle Database should be dot-qualified with the *Owner*. (This is frequently, but not always, *Sys*.) This preserves the intended meaning even if a local object, whose name collides with that of the intended object, is created in the schema which will be current when name resolution is done.

When the actual argument is a legal, unqualified SQL identifier, spelled just as it would be spelled in a SQL statement, then the function simply returns its input; otherwise, it raises the *ORA-44003: invalid SQL name* error. Each of the invocations of *b()* shown in *Code_11* runs without error and produces the expected result.

```
-- Code_11
b('t',   42);
b('T',   42);
b('"T"', 42);
```

The invocation shown in *Code_12* fails with *ORA-44003*.

```
-- Code_12
b('"USR"."T"', 42);
```

Code_13 shows the SQL statement it requests.

```
-- Code_13
select a.v1 from "USR"."T" a where a.PK = :b1
```

Though this statement is legal, the identifier is qualified and so the assertion that it is unqualified fails.

---

26. The notation *&&t* in the SQL statement template has no formal significance. It is used in the companion paper *How to write injection-proof PL/SQL* which elaborates at length on the template notion. Oracle Database's scheme for SQL execution allows a value to be bound to a placeholder; but it has no corresponding facility for an identifier in the statement.

The invocation shown in *Code_14* also fails with *ORA-44003*.

```
-- Code_14
b(
  't a
    where 1=0
    union
    select Username v1
    from   All_Users where User_ID = :b1 --',
  42);
```

Again, the identifier is illegal and so, again, the assertion fails. *Code_15* shows the SQL statement it requests.

```
-- Code_15
select a.v1 from t a
where 1=0
union
select Username v1
from   All_Users where User_ID = :b1 -- a where a.PK = :b1
```

This is the canonical SQL injection example. The requested SQL statement is legal. Significantly, the SQL statement is an instance of a different syntax template than the programmer intended and the dynamically constructed SQL statement has been changed in a dramatic and dangerous fashion[27]. Scrupulous use of the *DBMS_Assert* functions inoculates code against such vulnerability.

While the availability of the global temporary table allows a simpler and better approach in some use cases, there are others where the claim that one or more identifiers in a SQL statement are unknown until run time is unassailable. This is use case is (almost) the only one where a *select*, *insert*, *update*, *delete*, *merge*, or *lock table* statement[28] is optimally supported with native dynamic SQL.[29]

We sometimes hear a different supposed justification for using dynamic SQL where embedded SQL is functionally adequate: that doing this avoids the creation of dependencies and therefore allows the structure of tables and views that a PL/SQL unit relies on to be changed without invalidating that PL/SQL unit. The suggestion is that the programmer knows what the PL/SQL compiler cannot: that no changes to these essential tables and views will affect the validity of the access that the PL/SQL unit makes to them. The

---

27. It helps to imagine that the attacker has seen the source code, maybe in a test implementation, and has understood its weakness. However, many vulnerabilities to SQL injection can be discovered by black-box testing.

28. We will avoid the term *DML* in this paper because its formal definition in the Oracle Database SQL Language Reference book differs from common usage. Common usage excludes *select* and sets this in contrast to *insert*, *update*, *delete*, and *merge* — calling only the latter four DML (and forgetting *lock table* altogether). Moreover, the SQL Language Reference book includes *call* and *explain plan* in its definition of DML — but neither of these is supported by PL/SQL's embedded SQL.

29. There is one other — but it is esoteric. In, for example, a data warehouse application with huge tables, the optimal execution plan depends on using literal values for restriction predicates so that proper use may be made of the statistics that record the actual distribution of the values in the referenced columns. This implies that the predicates be directly encoded into the SQL statement which in turn mandates dynamic SQL.

    For this use case, *Sys.DBMS_Assert.Enquote_Literal()* should be used to prevent the threat of SQL injection.

need for this risky analysis vanishes in Oracle Database 11*g* because of the new fine-grained dependency tracking model that it brings. The system now establishes the safety of changes made to objects that are referenced in embedded SQL. If the change is safe, then the referencing PL/SQL unit remains valid; if it is potentially unsafe, then the PL/SQL unit is invalidated.

The discussion in this section is summarized in this best practice principle:

Strive to use SQL statements whose text is fixed at compile time. When you cannot, use a fixed template. Bind to placeholders. Use *DBMS_Assert* to make concatenated SQL identifiers safe.

Principle_5

Strive always to use only SQL statements whose text is fixed at compile time. For *select*, *insert*, *update*, *delete*, *merge*, or *lock table* statements, use embedded SQL. For other kinds of statement, use native dynamic SQL. When the SQL statement text cannot be fixed at compile time, strive to use a fixed syntax template and limit the run-time variation to the provision of names. (This implies using placeholders and making the small effort to program the binding.) For the names of schema objects and within-object identifiers like column names, use *Sys.DBMS_Assert.Simple_Sql_Name()*. If exceptional requirements mandate the use of a literal value rather than a placeholder, use *Sys.DBMS_Assert.Enquote_Literal()*. For other values (like, for example, the value for *NLS_Date_Format* in *Code_8*) construct it programatically in response to parameterized user input.

Finally in this section, recall that *execute immediate* is not the only construct that implements native dynamic SQL; the the *open Cur for* statement, where *Cur* is a *cursor variable*, will be examined in a later section.

The *DBMS_Sql* API

The *DBMS_Sql* API supports a procedural method for doing dynamic SQL. In historical versions of Oracle Database, it was the *only* way to do dynamic SQL, but events have moved on and native dynamic SQL is now supported in all supported versions of Oracle Database.

Native dynamic SQL was introduced as an improvement on the *DBMS_Sql* API (it is easier to write and executes faster). This point is made convincingly by *Code_17* and *Code_18*. Each executes the SQL statement set up by *Code_16* in a contrived test that successively selects each uniquely identified single row in a large table.

```
-- Code_16
Stmt constant varchar2(80) := '
  select t.n1 from t where t.PK = :b1';
```

Code_17 uses the *DBMS_Sql* API.

```
-- Code_17
declare
  Cur    integer := DBMS_Sql.Open_Cursor(Security_Level=>2);
  Dummy  integer;
begin
  DBMS_Sql.Parse(Cur, Stmt, DBMS_Sql.Native);
  DBMS_Sql.Define_Column(Cur, 1, n1);

  for j in 1..No_Of_Rows loop
    DBMS_Sql.Bind_Variable(Cur, ':b1', j);
    Dummy := DBMS_Sql.Execute_And_Fetch(Cur, true);
    DBMS_Sql.Column_Value(Cur, 1, n1);
    ...
  end loop;

  DBMS_Sql.Close_Cursor(Cur);
end;
```

Notice that the calls to *Open_Cursor()*, *Parse()*, *Define_Column()*, and *Close_Cursor()* are done outside the loop while the calls to *Bind_Variable()* and to *Execute_And_Fetch()* are done inside the loop. This saves the cost of repeatedly parsing the same SQL statement[30].

Code_18 uses native dynamic SQL.

```
-- Code_18
for j in 1..No_Of_Rows loop
  execute immediate Stmt
    into n1 using j;
  ...
end loop;
```

Notice how much shorter and more transparent *Code_18* is than *Code_17*; this, of course, improves that probability that correctly expresses the programmer's intention. Moreover, *Code_18* runs about twice as fast as *Code_17* on a 11,000 row test table, and at about the same speed as the equivalent embedded SQL approach.

However, the *DBMS_Sql* API supports some requirements for executing a SQL statement that *cannot* be met by native dynamic SQL[31]. These are they:

---

30. The call to *Parse()* attempts to find a *sharable SQL structure* in the shared pool with the same statement text that has the same meaning. If none is found, a so-called *hard parse* occurs. This is famously expensive. But even the task of establishing that there does already exist a suitable shareable structure, the so-called *soft parse*, incurs a cost. A re-work of *Code_17* that moves the *Open_Cursor()*, *Parse()*, *Define_Column()*, and *Close_Cursor()* calls into the loop runs about three times as slowly as does *Code_17* as presented on the 11,000 row test table used for the experiment. (Examination of appropriate statistics shows that the repeated parsing is indeed soft.)

31. Conversely, almost every requirement for executing a SQL statement that can be met by native dynamic SQL can also be met by the *DBMS_Sql* API. Oracle Database 11*g* brought a number of enhancements to the *DBMS_Sql* API: *Parse()* has new overload with a clob formal for the SQL statement; the *select list* may include columns of user-defined types; bind arguments of user-defined types are supported; and a *DBMS_Sql numeric cursor* may by transformed to a *ref cursor*. (A *ref cursor* may also be transformed to a *DBMS_Sql numeric cursor*.)

    There is one exception: the *select list* cannot be bulk fetched into a collection whose datatype is user-defined; rather, one of the collection types defined in the *DBMS_Sql* package spec must be used.

- The requirement for binding to placeholders is not known until run time. It is easy so see why by looking at *Code_18*. Native dynamic SQL supports binding with the *using* clause; and this clause is fixed at compile time. The *DBMS_Sql* API, in contrast, allows as many calls to *Bind_Variable()* as are needed to be made in response to run-time tests.
- The requirement for returning values is not known until run time. This arises most obviously with a *select* statement whose *select list* is not known until run time. *Code_18* shows how native dynamic SQL specifies the PL/SQL targets for the *select list* with the *into* clause; and this clause too is fixed at compile time. The *DBMS_Sql* API, in contrast, allows as many calls to *Define_Column()* as are needed to be made in response to run-time tests.

  If an *insert*, *update*, *delete*, or *merge* statement has a *returning* clause, the PL/SQL targets for these values are specified with the *using* clause[32] as shown in *Code_19*.

```
-- Code_19
  ...
  Stmt constant varchar2(200) := q'[
    update t
      set t.v1 = 'New '||t.v1
      where t.PK = :i1
      returning t.v1 into :o1]';
begin
  ...
  execute immediate Stmt using in PK, out v1;
  ...
```

The obvious question is this: when is which method for doing dynamic SQL preferred? And it answered obviously with this best practice principle:

For dynamic SQL, aim to use native dynamic SQL. Only when you cannot, use the *DBMS_Sql* API.

Principle_6

For dynamic SQL, always use native dynamic SQL except when its functionality is insufficient; only then, use the *DBMS_Sql* API. For *select*, *insert*, *update*, *delete*, and *merge* statements, native dynamic SQL is insufficient when the SQL statement has placeholders or *select list* items that are not known at compile time[33]. For other kinds of SQL statement, native dynamic SQL is insufficient when the operation is to be done in a remote database.

Cursor taxonomy

Only when the PL/SQL programmer has a reasonable experience of using each of three methods for issuing SQL is the need for carefully defined terms of art[34] to characterize the various kinds of cursor appreciated; and only then is it possible to understand the definitions.

Questions addressed by the cursor taxonomy

These are the key questions:

---

32. This apparent asymmetry is a consequence of the syntax of SQL's *returning* clause where the target for a returned value is given as a placeholder.

33. Notice that the function *DBMS_Sql.To_Refcursor()* can be used to transform a *DBMS_Sql numeric cursor* that has been executed to *cursor variable*; and the function *DBMS_Sql.To_Cursor_Number()* can be used to transform a *cursor variable* to a *DBMS_Sql numeric cursor*.

Question 0: In what domains of discourse is the term *cursor* used, and does it mean the same thing in each domain?

Question 1: Who manages the cursor (opening it, parsing the SQL statement, and so on through to closing it) — the programmer or the PL/SQL system?

Question 2: Does the cursor have a programmer-defined identifier, and if it has, how can this be used?

Question 3: Is the cursor opened using embedded SQL, native dynamic SQL, or the *DBMS_Sql* API?

With respect to *Question 0*, there are two different domains of discourse: the overt and the covert. The overt one is PL/SQL source text and the discussion of the behavior it specifies given the definition of PL/SQL's syntax and semantics. And the covert one is the implementation of the PL/SQL system — of which, as theoretically undesirable as this might be, the professional Oracle Database developer eventually needs some understanding. The PL/SQL run-time system manages the processing of the SQL statements that the source text specifies by making appropriate calls to a server-side functional equivalent of the OCI[35]. This API can be pictured as that which implements the familiar client-side OCI at the receiving end of the Oracle Net protocol. It therefore supports an equivalent set of operations. PL/SQL code which uses any of its three methods for issuing SQL is implemented in roughly the same way with calls at run time to the server-side OCI. This means that the same notion of *cursor* with which the programmer who uses client-side OCI is familiar supports the discussion of the run-time processing of the SQL statements issued by database PL/SQL. In particular, the initialization parameters like *Open_Cursors* and *Session_Cached_Cursors* have the same significance for the SQL issued by database PL/SQL as they do for SQL issued by client programs using the OCI directly, using ODBC, or using the JDBC driver (in either its thick or thin versions).

A further part of the covert domain of discourse concerns the shared pool. Sometimes the session-independent structures that are characterized by facts exposed in the *v$SqlArea* and *v$Sql* views are referred to, albeit inaccurately, as cursors[36].

The terms of art

Here, then, are the terms of art[37]:

- *sharable SQL structure*

  This is the object that lives in the shared pool and whose metadata is exposed in the *v$SqlArea* and *v$Sql* views. A *sharable SQL structure* lives on beyond the

---

34. A Google search for "term of art" turns up the following nice definition and discussion from Everything2.com: *A word or phrase used by practitioners in a field of endeavour which has a precise and typically quite technical meaning within the context of the field of endeavour. Terms of art allow practitioners in a field to communicate with each other concisely and unambiguously. Inventing suitable yet totally new words to be used as terms of art is often quite difficult. Consequently, the words which become terms of art often also have non-field-specific meanings. This can create and/or reinforce communication barriers between a field's practitioners and non-practitioners...*

35. *OCI* stands for the *Oracle Call Interface.*

36. You hear this, for example, in turns of phrase like "cursor sharing" and "child cursor".

lifetime of the session that created it and can be used by other sessions concurrently.

The reuse of a *sharable SQL structure* is possible only when the SQL statement text is identical and when other so-called sharing criteria (most notably that an identifier in the SQL statement denotes the same object) are satisfied[38]. The term belongs, therefore, in the covert domain: the implementation of PL/SQL and, in fact, the implementation of any environment that supports the processing of SQL statements. Of course, reusing a *sharable SQL structure* improves performance; this fact is the reason behind one of the most famous best practice principles for SQL processing from all environments:

When using dynamic SQL, avoid literals in the SQL statement. Instead, bind the intended values to placeholders.

Principle_7
Avoid using concatenated literals in a dynamically created SQL statement; rather, use placeholders in place of literals, and then at run time bind the values that would have been literals. This maximizes the reuse of *sharable SQL structures.*

- *session cursor*

This is the object that lives in a session's memory[39], that dies, therefore, with the session, and whose metadata is exposed in the *v$Open_Cursor* view; it supports an individual session's SQL processing.

This term, too, belongs, in the covert domain; and, again, any client (of which PL/SQL is just one example) that issues a SQL statement uses a *session cursor*. A *session cursor* is associated with a exactly one *sharable SQL structure*; but a *sharable SQL structure* may have several *session cursors* associated with it. A *session cursor* is also an object of potential re-use. When a client finishes processing a particular SQL statement, the *session cursor* that supported this processing is not destroyed; rather it is marked as *soft closed* and is retained in a least-recently-used cache[40]. A client's call to parse a SQL statement is implemented by searching first in the cache of *soft closed session cursors*. The search uses the same criteria as the search for a *sharable SQL structure* reuse candidate in the shared pool: identity of the SQL statement text and identity of its meaning[41]. Only if no match is found is the shared pool searched for a matching *sharable SQL structure* to be used as the basis for a new *session cursor*. The search in the shared pool is

37. It helps to think of *implicit*, *explicit*, and *cursor* as words in a foreign language, and to forget any associations that their English meanings might have. Especially, it helps to think of phrases like *implicit cursor* and *explicit cursor* as foreign idioms with no direct translation into English; the concern must be only with the meaning and the correct usage of these phrases. The discipline is not, after all, unusual. The US English word *freeway* suggests that there might be no charge for using one; but freeways where a toll is levied are not uncommon.

38. The reuse is limited to these kinds of SQL statement: *select*, *insert*, *update*, *delete*, *merge*, and *anonymous PL/SQL block*. It is in these, and only these, kinds that a placeholder is legal.

39. This is usually referred to as the PGA but should, for accuracy, be called the UGA.

40. *session cursors* are cached only when the *Cursor_Space_For_Time* is set to *true*.

41. PL/SQL uses on optimized version of this approach, based on the source text location of the PL/SQL statement that occasions the parse call, that narrows the search space to just one item.

the so-called *soft parse*, and the re-usability of *soft closed session cursors* is an optimization that avoids the cost of the *soft parse*. In the worst case, no match will be found in the shared pool — and in this case the so-called *hard parse* creates an appropriate new *sharable SQL structure* that in turn becomes the basis for a new *session cursor*.

The *Open_Cursors* initialization parameter sets the maximum number of *session cursors* that can exist in the open state concurrently in one session. The *Session_Cached_Cursors* initialization parameter sets the maximum number of *session cursors* that can be in the *soft closed* state concurrently in one session[42]. Under pressure, a *soft closed session cursor* may be destroyed to make space for a newly *soft closed* one or for a newly opened *session cursor*.

- *implicit cursor*

  This term denotes a *session cursor* that supports the SQL processing that implements the family of embedded SQL constructs and native dynamic SQL constructs where, in terms of overt PL/SQL language constructs and concepts, there is, quite simply, no cursor to be seen. It is the PL/SQL runtime system, reflecting the analysis done by the PL/SQL compiler, that manages the *session cursor* without the help of explicit language constructs that specify operations like *open*, *parse*, *bind*, *execute*, *fetch*, and *close*. The term belongs, therefore, in the covert domain.

  We hear the term used informally to denote any example in the family of cursor-less PL/SQL constructs for executing SQL, but we encourage caution with this use. Paradoxically, another similar sounding term of art, *implicit cursor attribute*, belongs in the overt domain of PL/SQL's syntax and semantics. We will defer its definition until we define its cousin, the *explicit cursor attribute*. *Code_1*, *Code_2*, and *Code_3* show examples of cursor-less PL/SQL constructs for embedded SQL; and *Code_8*, *Code_10*, *Code_18*, and *Code_19* show examples of cursor-less PL/SQL constructs for native dynamic SQL, which always implies the use of *execute immediate*.

- *explicit cursor*

  While the words suggest that this might be the natural opposite of an *implicit cursor* this is not the case[43]. An *explicit cursor* is a specific PL/SQL language feature — and the term belongs, therefore, firmly in the overt domain. The identifier *Cur_Proc*, declared in the spec of package *Pkg1* in *Code_20*, denotes an *explicit cursor*.

```
-- Code_20
package Pkg1 is
  type Result_t is record(PK t.PK%type, v1 t.v1%type);
  cursor Cur_Proc(PK in t.PK%type) return Result_t;
  ...
end Pkg1;
```

---

42. *Open_Cursors* sets a functionality limit; it must acknowledge the number of *session cursors* that an application might need to have concurrently active. *Session_Cached_Cursors*, in contrast, governs a classic space against performance trade-off.

43. You have, earlier, been warned to expect this!

Cur_Proc is defined in the body of package *Pkg1* in *Code_21*.

```
-- Code_21
package body Pkg1 is
  cursor Cur_Proc(PK in t.PK%type) return Result_t is
    select    a.PK, a.v1
    from      t a
    where     a.PK > Cur_Proc.PK
    order by  a.PK;
  ...
end Pkg1;
```

Code_22 shows how the *explicit cursor Cur_Proc* might be used. The construct is called a *explicit cursor for loop*.

```
-- Code_22
for r in Pkg1.Cur_Proc(PK=>Some_Value)loop
  Process_Record(r);
end loop;
```

An *explicit cursor* cannot be defined using dynamic SQL; embedded SQL is the only possibility.

Critically, though the programmer invents the name of an *explicit cursor*, this is not a variable: it cannot be used as an actual argument in a subprogram invocation; nor can it be returned by a function. In this way, it is very much like a procedure[44] — and it shares other similarities: it can be forward declared, and the declaration and the definition can be split between a package and its body; and it can have formal parameters. We shall see, however, that there is never a good reason to write code that takes advantage of this possibility. (A function whose return a *ref cursor* can always be used as an alternative and, as shall be seen, has some advantages.)

- *ref cursor*

  This is a PL/SQL-only datatype[45] declared, for example, as is *Cur_t* in *Code_23* or in *Code_24*. A *ref cursor* may be used to declare a variable, a formal parameter for a subprogram, or a function's return value. It may not be used to declare the datatype of the element of a collection or the field of a *record*. There are exactly two kinds of *ref cursor*: a *weak ref cursor* and a *strong ref cursor*.

- *cursor variable*

  This is a variable whose datatype is based on a *ref cursor*. All of the terms *ref cursor*, *weak ref cursor*, *strong ref cursor*, and *cursor variable* describe PL/SQL language features and so they belong in the overt domain. When *Cur* is a *cursor variable*, it can be used in the *open Cur for* PL/SQL statement which is used to associate a *select* statement with the *cursor variable*. The association can

---

44. It might have been better had the construct been named *cursor subprogram*.

45. Strictly speaking, the keyword *cursor* here denotes a datatype constructor — just as does, for example, *record(...)*, or *table of boolean index by pls_integer*. The keyword *ref* is distinct and denotes the fact that quantities whose datatype is *ref cursor* obey reference semantics. It is very unusual in this way in this way. Normally, quantities in PL/SQL obey value semantics. There is a tiny number of other quantities that obey reference semantics; one example is permanent lob locators (which are a rule unto themselves).

made using either embedded SQL or native dynamic SQL. *Cur* can also be used as the source for a *fetch* statement[46].

A *cursor variable* may not be declared at global level in a package spec or body.

- *strong ref cursor*

  This is a datatype declared, for example, as is *Strong_Cur_t* in *Code_23*.

```
-- Code_23
  type Result_t is record(PK t.PK%type, v1 t.v1%type);
  type Strong_Cur_t is ref cursor return Result_t;
```

  A *strong ref cursor* is specific about the number and the datatypes of the *select list* items that its *select* statement must define. A *cursor variable* whose datatype is a *strong ref cursor* can be opened only using embedded SQL.

- *weak ref cursor*

  This is a datatype declared, for example, as is *Weak_Cur_t* in *Code_24*.

```
-- Code_24
  type Weak_Cur_t is ref cursor;
```

  A *weak ref cursor* is agnostic about the number and the datatypes of the *select list* items that its *select* statement must define. A *cursor variable* whose datatype is a *weak ref cursor* can be opened using either embedded SQL or native dynamic SQL[47].

  The spec of package *Pkg2* in *Code_25*, declares the function *New_Cursor()*, parameterized in just the same way as the *explicit cursor* declared in *Pkg1* in *Code_20*, that is designed to give a value to a *cursor variable*.

```
-- Code_25
package Pkg2 is
  type Result_t is record(PK t.PK%type, v1 t.v1%type);

  type Cur_t is ref cursor
    $if $$Embedded $then return Result_t;
    $else ;
    $end

  function New_Cursor(
    PK in t.PK%type)
    return Cur_t;
  ...
end Pkg2;
```

---

46. The *ref cursor* and the *cursor variable* were introduced into PL/SQL later than the *explicit cursor* to overcome the latter's restrictions. Had history been different, and had the *ref cursor* and the *cursor variable* been introduced first (and had the *batched bulk fetch* constructs been supported from the beginning), it would very unlikely that a project to introduce the *explicit cursor* would have been justified.

47. Using a *cursor variable* whose datatype is a *strong ref cursor* has the small advantage that you will get a compile-time error rather than a run-time error in the case that you attempt to fetch into a *record* or a set of scalars that don't match the shape of the *select list*. Otherwise, it has no benefit and brings a minor source text maintenance cost. Starting in Oracle9*i* Database, package *Standard* declares the *weak ref cursor* type *Sys_RefCursor*; using this saves some typing and communicates its meaning immediately to readers.

The body of package *Pkg2* in *Code_26*, defines the function.[48].

```
-- Code_26
package body Pkg2 is
  function New_Cursor(
    PK in t.PK%type)
    return Cur_t
  is
    Cur_Var Cur_t;
  begin

    open Cur_Var for
      $if $$Embedded $then
        select    a.PK, a.v1
        from      t a
        where     a.PK > New_Cursor.PK
        order by  a.PK;
      $else
        '
          select    a.PK, a.v1
          from      t a
          where     a.PK > :b1
          order by  a.PK'
        using in New_Cursor.PK;
      $end

    return Cur_Var;
  end New_Cursor;
  ...
end Pkg2;
```

Code_25 and *Code_26* use conditional compilation[49] to emphasize the small differences, and the large similarities, between the two ways to open the *cursor variable*.

---

48. There is hardly ever a reason, in real code, to use native dynamic SQL with a SQL statement whose text is declared using a *constant* when the kind of the statement is *select*, *insert*, *update*, *delete*, *merge* or *anonymous PL/SQL block*. As mentioned earlier, a possible reason is that a to-be-referenced table or PL/SQL unit doesn't exist at compile time but is created, somehow, before the code runs. Unless the code is an install script, the use case that seems to suggest this approach should be examined very carefully.

49. Conditional compilation was introduced in Oracle Database 10*g* Release 2. The identifier *Embedded* is called a CC Flag and its value is obtained in the source text by writing *$$Embedded* (a so-called inquiry directive); its value is set by commands like this (before *create or replace*):

```
alter session set Plsql_CCflags = 'Embedded:true'
```

or this (for an existing package body *Pkg*):

```
alter package Pkg compile
  Plsql_CCflags = 'Embedded:false'
  reuse settings
```

Code_27 shows how the *cursor variable Cur_Var*, initialized using *New_Cursor()*, might be used. The loop construct is called an *infinite cursor fetch loop*[50].

```
-- Code_27
declare
  Cur_Var Pkg2.Cur_t :=
    Pkg2.New_Cursor(PK=>Some_Value);
  r Pkg2.Result_t;
begin
  loop
    fetch Cur_Var into r;
    exit when Cur_Var%NotFound;
    Process_Record(r);
  end loop;
  close Cur_Var;
end;
```

The *infinite cursor fetch loop* (in *Code_27*), though functionally equivalent to the *explicit cursor for loop* (in *Code_22*), is undoubtedly more verbose. The *infinite cursor fetch loop* can be used both with a *cursor variable* and an *explicit cursor*; but the *explicit cursor for loop* can be used only with an *explicit cursor*. Moreover, from Oracle Database 10*g*, the *explicit cursor for loop* is significantly faster than the *infinite cursor fetch loop*. This is because the optimizing compiler is able safely to implement the former, under the covers, using array fetching[51]. Such an optimization is unsafe for the latter because the optimizer cannot always prove that other interleaved fetches do not happen from the same *explicit cursor* or *cursor variable* elsewhere in the code[52]. However, we shall see that all this is of no practical interest because neither approach is ever preferred to *batched bulk fetch* (see *Code_29*) or *entire bulk fetch* (see *Code_32* and *Code_33*).

- *identified cursor*

  Because the source text of an *infinite cursor fetch loop* is identical for both an *explicit cursor*, and a *cursor variable*, and because (as we shall see in *Code_29* and *Code_34*) there are other constructs that have this property, it is useful to have a term of art for the superclass of *explicit cursor* and *cursor variable*. In fact, there is no such term — so this paper will introduce the term *identified cursor* for precisely that purpose. This then allows a nice formulation of the distinction between a cursor-less PL/SQL construct, which is supported under the covers by an *implicit cursor* and a construct that uses an *identified cursor* where the programmer invents an identifier for either an *explicit cursor* or a *cursor variable* and, to some extent at least, instructs the PL/SQL system how to manage the supporting *session cursor*.

---

50. Notice that the *explicit cursor for loop* is legal only for an *explicit cursor* but that the *infinite cursor fetch loop* is legal for both a *explicit cursor* and a *cursor variable*. However, as we shall see, this is of no practical interest.

51. Here, "array fetching" refers to the programming technique in server-side OCI.

52. The limitation that the optimizer cannot prove that other interleaved fetches do not happen from the same *explicit cursor* or *cursor variable* elsewhere in the code reflects the optimizer technology currently in use (i.e. in Oracle Database 11*g*).

- *DBMS_Sql numeric cursor*

  This is the return value of the function *DBMS_Sql.Open_Cursor()* and it can be assigned to an ordinary variable (say, *Cur*) of datatype *number* (or a subtype of *number* like *integer*). When you have finished processing the SQL statement, you call *DBMS_Sql.Close_Cursor()* using *Cur* as the actual value for its *in out* formal parameter, *c*. Provided that on calling it, *Cur* denotes an existing open *DBMS_Sql numeric cursor*, this will set *Cur* to *null*. You can discover if the current value of *Cur* denotes an open *DBMS_Sql numeric cursor* by calling *DBMS_Sql.Is_Open()*. Every subprogram in the *DBMS_Sql* API has an *in* formal parameter for which the actual value must be a an existing open *DBMS_Sql numeric cursor* except for *Open_Cursor()*, which returns such a value, and *Close_Cursor()*, where the parameter mode is *in out*. Calling any of these with a value of *Cur* that does not denote an existing open *DBMS_Sql numeric cursor* causes the error *ORA-29471: DBMS_SQL access denied*; once this error has occurred in a session, all subsequent calls to any subprogram in the *DBMS_Sql* API causes the same error[53].

  The *Open_Cursor()* function has two overloads. The first has no formal parameters; the second overload, new in Oracle Database 11*g*, has one formal: *Security_Level* with allowed values *1* and *2*. When *Security_Level* = *2*, the *Current_User* and the enabled roles must be the same[54] for all calls to the the *DBMS_Sql* API as they were for the most recent call to *Parse()*. When *Security_Level* = *1*, the *Current_User* and the enabled roles must be the same for calls to *Bind_Variable()*, *Execute()*, and *Execute_And_Fetch()* as they were for the most recent call to *Parse()*, but calls to *Define_Column()*, *Define_Array()*, *Fetch_Rows()*, and so on are unrestricted. This paper recommends encapsulating calls to the the *DBMS_Sql* API in a producer PL/SQL unit (see *"Approaches for producer/consumer modularization"* on page 41). The following best practice principle follows from that recommendation:

  Principle_8

Always open a *DBMS_Sql numeric cursor* with *DBMS_Sql.Parse(Security_Level=>2);*

  Always use the overload of the *DBMS_Sql.Parse()* that has the formal parameter *Security_Level*[55] and always call it with the actual value *2* to insist that all operations on the *DBMS_Sql numeric cursor* are done with the same *Current_User* and enabled roles.

---

53. This is new behavior in Oracle Database 11*g*. The purpose is to inoculate against so-called scanning attacks that attempt to hijack an open *DBMS_Sql numeric cursor* and, for example, re-bind and re-execute in order to see restricted data.

54. More carefully, the enabled roles must be the same as, or a superset of, the enabled roles at the time of the most recent *Parse()* call.

55. Enhancement request 6620451 asks for a compiler warning when the overload of the *DBMS_Sql.Parse()* that has no formal parameters is used.

- *explicit cursor attribute*

  When *Cur* is either an *explicit cursor* or a *cursor variable*, then these reflectors[56] are available: *Cur%IsOpen* (returns *boolean*), *Cur%NotFound* (returns *boolean*)[57], and *Cur%RowCount* (returns *integer*). Unless *Cur%IsOpen* is true, then an attempt to reference the other *explicit cursor attributes* fails[58]. Because both an *explicit cursor* and a *cursor variable* may be opened only for a *select* statement, then the meanings of *Cur%RowCount* and *Cur%NotFound* are obvious: the former gives the total number of rows fetched to date during the lifetime of the cursor; and the latter remains *true* until all rows have been fetched. We shall see that if the programmer adopts the approaches for processing SQL statements that this document recommends, then for all common use cases *Cur%RowCount* and *Cur%NotFound* are of no practical interest. *Cur%IsOpen* is potentially useful in an exception handler as *Code_28* shows[59].

```
-- Code_28
if Cur_Var%IsOpen then
  close Cur_Var;
end if;
```

- *implicit cursor attribute*

  These reflectors report information about the execution of the current, or most-recently finished if none is current, SQL statement for which an *implicit cursor* is used. Because an *implicit cursor* supports, among other things, the *execute immediate* statement, this statement may be of any kind. Here is the list of scalar reflectors: *Sql%IsOpen*, *Sql%NotFound*[60], *Sql%RowCount*. They return the same datatypes and have the same meanings as the correspondingly named *explicit cursor attributes*. (Because *SQL* is a reserved word in PL/SQL, there is no risk of confusion between the *implicit cursor attributes* and their *explicit cursor attribute* counterparts.) However, precisely because an *implicit cursor* is managed by the PL/SQL system, it is never interesting to observe *Sql%IsOpen*; its existence is just a curiosity. Perhaps surprisingly (but in fact for the same reason — that an *implicit cursor* is managed by the PL/SQL system) both *Sql%NotFound* and *Sql%RowCount* can be referenced when *Sql%IsOpen* is false. *Sql%NotFound* is always equal to *Sql%RowCount = 0* and so, for that reason, can be forgotten. *Sql%RowCount* reports the number of rows affected by the latest *select*, *insert*, *update*, *delete*, or *merge* statement. However, it is never interesting for a *select* statement because other, more direct, methods give the

---

56. They behave like functions: they can be used in expressions but not as assignment targets.

57. There is also *Cur%Found*, but it isn't worth mentioning because its value is always equal to *not Cur%NotFound* and it is hardly ever used. (If *Cur%NotFound* is *null*, then *Cur%Found* is also *null*.)

58. The error is *ORA-01001: invalid cursor*.

59. When a *cursor variable* goes out of scope, the PL/SQL run-time system looks after closing it. However, it cannot harm to attend to this safety measure explicitly in an exception handler. It is simpler always to do this than it is to reason about whether or not the measure is needed.

60. Of course, there is also *Sql%Found* whose value is always equal to *not Sql%NotFound*.

same answer — as will be seen. *Sql%RowCount* is always zero after statements of other kinds, and is never interesting there.

Here is the list of non-scalar reflectors: *Sql%Bulk_RowCount*, *Sql%Bulk_Exceptions*. Each is interesting in connection with the *forall* statement, and only there. The *forall* statement supports these, and only these, kinds of SQL statement: *insert*, *update*, *delete*, and *merge*. *Sql%Bulk_RowCount* is a collection, indexed by a numeric datatype, whose element datatype is *pls_integer*. It reports the number of rows affected by each iteration of the *forall* statement; the index is the iteration number, running consecutively from *1* to the *Count()* of the collection that was used to drive the statement. *Sql%Bulk_Exceptions* is interesting only when *save exceptions* is used in the *forall* statement, and can be accessed only in an exception handler for the *Bulk_Errors* exception, *ORA-24381*. It is a collection, indexed by *pls_integer*, whose element datatype is a based on a *record*; the first field is *Error_Index* and the second field is *Error_Number*, both *pls_integer*. The index runs consecutively from *1* to however many iterations of the *forall* statement caused an exception. *Error_Index* is the iteration number, in the range from *1* to the *Count()* of the collection that was used to drive the statement. *Error_Number* is equal to the Oracle error number corresponding to the exception (as would be used in the *Pragma Exception_Init* statement) multiplied by *-1*[61].

Each new execution of a SQL statement by a method that uses an *implicit cursor* overwrites the values that the *implicit cursor attributes* reported for the previous such SQL statement. This fact is the reason for the following best practice principle:

Principle_9

The only *explicit cursor attribute* you need to use is *Cur%IsOpen*. The only *implicit cursor attributes* you need are *Sql%RowCount*, *Sql%Bulk_RowCount*, and *Sql%Bulk_Exceptions*.

When the approaches that this paper recommends are followed, the only useful *explicit cursor attribute* is *Cur%IsOpen*. There is never a need to use the other *explicit cursor attributes*. The only scalar *implicit cursor attribute* of interest is *Sql%RowCount*. Always observe this in the PL/SQL statement that immediately follows the statement that executes the SQL statement of interest using an *implicit cursor*. The same rationale holds for the *Sql%Bulk_RowCount* collection. *Sql%Bulk_Exceptions* must be used only in the exception handler for the *Bulk_Errors* exception; place this in a *block statement* that has the *forall* statement as the only statement in its executable section.

Summary

We now have the firm foundation of carefully defined terms of art with which to answer the questions that were listed at the start of this section.

- In answer to *"Question 0: In what domains of discourse is the term cursor used, and does it mean the same thing in each domain?"*, the term is used in two distinct domains: the overt (PL/SQL's syntax and semantics) and the covert (PL/SQL's runt-time implementation). In a bigger picture, the term is also used in discussing

61. The fact that *Error_Number* is a positive number confuses some users. It's best seen as a bug that will never be fixed because of the number of extant programs that would be broken by such a behavior change.

the syntax and semantics of other environments that support the processing of SQL statement — but these are of no interest in this paper[62].

- In answer to *"Question 1: Who manages the cursor (opening it, parsing the SQL statement, and so on through to closing it) — the programmer or the PL/SQL system?"*, there is a set of cursor-less PL/SQL constructs where the PL/SQL system determines how to manage the *session cursor* that supports the SQL statement without any language constructs that command how to do this. In such cases, the *session cursor* is called an *implicit cursor*. (The junior programmer who writes code in conformance with an application architecture that someone else has invented, might survive for years without even hearing the word *cursor*.) In the other cases (and in embedded SQL and native dynamic SQL, these are always for *select* statements) the programmer uses one of two explicit language constructs to determine the management of the *session cursor*. The constructs are the *explicit cursor* and the *cursor variable*. Of course, with the the *DBMS_Sql* API, the programmer micromanages the *session cursor* by using subprograms that map more-or-less one-to-one to the cursor management primitives exposed by the OCI.
- In answer to *"Question 2: Does the cursor have a programmer-defined identifier, and if it has, how can this be used?"*, the cursor-less PL/SQL constructs tautologically do not allow programmer-defined identifiers but for an *explicit cursor*, a *cursor variable*, and a *DBMS_Sql numeric cursor*, the programmer invents an identifier. The identifier for an *explicit cursor* is like the identifier for a subprogram: it cannot be used in assignments (which implies that it can't be used as a subprogram's formal parameter). In contrast, the identifiers for a *cursor variable* and for a *DBMS_Sql numeric cursor* can be used in assignments and as formal parameters for subprograms.
- In answer to *"Question 3: Is the cursor opened using embedded SQL, native dynamic SQL, or the DBMS_Sql API?"*, an *implicit cursor* is managed in response to certain kinds of embedded SQL and native dynamic SQL statements but never as a consequence of using the *DBMS_Sql* API. An *explicit cursor* is associated only with an embedded SQL *select* statement. A *cursor variable* may be opened for either an embedded SQL *select* statement or a native dynamic SQL *select* statement. A *DBMS_Sql numeric cursor* can, of course, be used for any kind of SQL statement.

Finally in this section, notice that *cursor*[63] by itself is not a useful term of art; on the contrary, without qualification, it has no meaning — unless, that is, the immediately surrounding sentences use the proper term of art so that the

---

62. The SQL*Plus scripting language supports the command `VARIABLE Cur REFCURSOR` to allow *:Cur* to be written as a placeholder for a *cursor variable* in SQL statement command.

63. The reason for choosing the word *cursor* is obvious: it runs along a result set and holds the current position. At the time the term came into use (recall that, at this paper's date, Oracle Corporation is celebrating 30 years of history) visual display units (the green-screen representation of a 80-characters wide scrollable teletype roll) needed a similar notion for the current character position and adopted the same term *cursor*. The term belongs to the discussion of the processing a *select* statement — but it has been extended to denote, with more or less precision, *any* structure that is involved in the processing of *any* kind of SQL statement.

unqualified *cursor* is a useful elision to avoid indigestible prose. This leads to the following best practice principle:

Learn the terms of art: *session cursor*, *implicit cursor*, *explicit cursor*, *cursor variable*, and *DBMS_Sql numeric cursor*. Use them carefully and don't abbreviate them.

Principle_10

When discussing a PL/SQL program, and this includes discussing it with oneself, commenting it, and writing its external documentation, aim to avoid the unqualified use of "cursor". Rather, use the appropriate term of art: *session cursor*, *implicit cursor*, *explicit cursor*, *cursor variable*, or *DBMS_Sql numeric cursor*. The discipline will improve the quality of your thought and will probably, therefore, improve the quality of your programs.

APPROACHES FOR *SELECT* STATEMENTS

This section capitalizes unashamedly on the assumption that the reader has general familiarity with the subject matter and treats the optimal approaches in detail, even if the notions are sometimes regarded (wrongly, as will be seen) as difficult; it touches only in passing on non-optimal approaches which, for historical reasons, are regarded as easier.

Selecting many rows — unbounded result set

One common use for database PL/SQL programs is in the batch preparation of reports where each row (and its associated details) of a very large table needs to be processed. In such scenarios, each row can be processed in isolation.

All PL/SQL application developers who program using Oracle Database come, sooner or later, to understand that SQL and PL/SQL are executed by different virtual machines. This means that when a PL/SQL subprogram executes a SQL statement there is at least one, and possibly many, so-called context switches from the PL/SQL virtual machine, to the SQL virtual machine, and back to the PL/SQL virtual machine. The context switch is accompanied by a representation change for data because PL/SQL uses a format that is optimized for in-memory use and SQL uses a format that is optimized for on-disk use. The context switch inevitably incurs a cost; performance, therefore, can be improved by minimizing the number of context switches that occur during the execution of a SQL statement[64].

Oracle8*i* Database (by now, ancient history[65]) introduced the PL/SQL constructs for bulk SQL — so the constructs are now supported in every supported version of Oracle Database. For a *select* statement, these allow many, and in the limit all, result rows to be fetched with a single context switch. There is no reason, in code intended for production use, to prefer PL/SQL's non-bulk constructs when executing a *select* statement that returns many rows[66].

---

64. The context switch discussion might seem uncomfortable; but it is worth remembering that any alternative suffers much worse from the same effect. Consider how a client-side program, written for example in C or Java, executes SQL. There is here, too, necessarily a context switch. But it is more dramatic: the round-trip, which for the PL/SQL-to-SQL context switch takes place within the address space of a single executable program, here takes place between distinct programs running on different machines and is mediated by yet other programs that implement the network communication; and the data representation is sometimes transformed twice (between the client program's representation and the on-wire representation, and between the on-wire and SQL's on-disk representation).

65. The PL/SQL User's Guide and Reference, Release 8.1.6 is still available online from the Oracle Technology Network website. Its publication date is 1999.

66. Everyone who is fluent in PL/SQL uses it frequently for generating disposable *ad hoc* reports. In such write-once, use-a-few-times cases, the *implicit cursor for loop* shown in *Code_2* is a fair approach because it takes slightly less effort to program than the bulk approach shown in *Code_29*.

Programming the *fetch* loop

Of course, the target into which many rows are to be fetched must be a collection. The most natural way to model this is to use a collection of *records* whose fields correspond to the *select list* items. *Code_29* shows an example.

```
-- Code_29
-- Cur is already open here.
-- The fetch syntax is the same for
-- an explicit cursor and a cursor variable.
loop
  fetch Cur bulk collect into Results limit Batchsize;
  -- The for loop doesn't run when Results.Count() = 0
  for j in 1..Results.Count() loop
    Process_One_Record(Results(j));
  end loop;
  exit when Results.Count() < Batchsize;
end loop;
close Cur;
```

Notice that, when the requirements document informs the programmer that the result set from the query used to open the cursor *Cur* can be arbitrarily big, the programmer cannot safely fetch all the results in a single context switch. Rather, they must be fetched in batches whose maximum size can be safely set at compile time — and the use of the *limit* clause requests this.

Some programmers don't immediately understand how to specify the exit criterion and are tempted to test *Cur%NotFound*. This is inappropriate because, to their surprise, it first becomes *false* when the *fetch* statement still *does* get rows, but when the number of fetched rows is less than the batchsize. The size of the last, and partial, batch is very likely to be nonzero. Therefore, exiting when *Cur%NotFound* is *false* will, in general, silently cause buggy behavior[67]. The appropriate quantity to test is, therefore, the number of rows fetched this time: *Results.Count()*. While it is self-evidently correct to place *exit when Results.Count() < 1;* immediately after the *fetch* statement, this will, in general, mean that one more fetch is attempted than is strictly needed. *Code_29* shows a correct approach that avoids that tiny cost. It does, of course, carry the burden of ensuring correct behavior when the total number of rows happens to be an integral multiple of the batchsize. Usually, though, the results processing is driven by a *1 . . Results.Count()* loop, and then the correctness comes for free.

The conclusion is that *explicit cursor attributes* are not useful in the implementation of the *batched bulk fetch*.

A reasonable value for *Batchsize* is a couple of hundred, or several hundred[68]. There is no reason to change this at run time, and so *Batchsize* is best declared as

67. *Cur%RowCount* is no help either; it is incremented with each successive *fetch* statement and eventually is left equal to the total number of rows fetched over all batches.

68. Experiments show that when the batchsize is just a few (say 3, or 5) then a small increase makes a very big reduction in the time needed to process all the results; but that when it is a few hundred, then diminishing returns are observed for further increases — even when the increase is from a few hundred to a few thousand. You might want to experiment with this yourself.

a *constant*; correspondingly, the collection is best declared as a *varray* with a maximum size equal to *Batchsize*. *Code_30* shows the declaration[69].

```
-- Code_30
Batchsize constant pls_integer := 1000;

type Result_t is record(PK t.PK%type, v1 t.v1%type);
type Results_t is varray(1000) of Result_t;
Results Results_t;
```

Opening the cursor

It is significant that there is no syntax to express the *batched bulk fetch* in a cursor-less PL/SQL construct; the approach requires the use of an *identified cursor*. *Code_31* shows the three ways to establish the *identified cursor Cur* so that *Code_29* is viable.

```
-- Code_31
  ...
  $if $$Approach = 1 $then
    cursor Cur is
      select    a.PK, a.v1
      from      t a
      where     a.PK > b.Some_Value
      order by  a.PK;
  $elsif $$Approach = 2 or $$Approach = 3 $then
    Cur Sys_Refcursor;
    $if $$Approach = 3 $then
      Stmt constant varchar2(200) := '
        select    a.PK, a.v1
        from      t a
        where     a.PK > :b1
        order by  a.PK';
    $end
  $end
begin
  $if $$Approach = 1 $then
    open Cur;
  $elsif $$Approach = 2 $then
    open Cur for
      select    a.PK, a.v1
      from      t a
      where     a.PK > b.Some_Value
      order by  a.PK;
  $elsif $$Approach = 3 $then
    open Cur for Stmt using Some_Value;
  $end
  ...
```

When the CC Flag *Approach* is *1*, then *Cur* is established as an *explicit cursor*; when *Approach* is *2*, then it is established as a *cursor variable* and is opened using embedded SQL; and when *Approach* is *3*, then it is established as a *cursor variable* and is opened using native dynamic SQL.

However, as the comment in *Code_29* points out, the fetching code is identical for these two kinds of cursor. This, of course, prompts the question "When is a *explicit cursor* preferred, and when is a *cursor variable* preferred?" Recall that when the use case requires dynamic SQL, then only a *cursor variable* can be used. So the question is relevant only when the requirements can be met with embedded SQL. This is the simple answer: use an *explicit cursor* in this case

69. Through Oracle Database 11*g*, PL/SQL has no convenient way to avoid the textual repetition of the magic number *1000* in *Code_30*. (While a conditional compilation inquiry directive, for example *$$Batch_Size*, could be used, this technique would bring the bigger disadvantage of breaking the encapsulation that the PL/SQL unit that contains *Code_30* seeks to provide.)

because the goal of declaring it and associating it with the intended *select* statement is achieved with more compact code.

However, there are other considerations; but these are best discussed in the context of concrete use cases. This is deferred to *"Approaches for producer/consumer modularization"* on page 41.

The discussion in this section is summarized in this best practice principle:

When you don't know how many rows your query might get, use *fetch... bulk collect into* with the *limit* clause inside an *infinite loop*.

Principle_11

When many rows are to be selected, and the result set may be arbitrarily big, process them in batches by using *fetch... bulk collect into* with the *limit* clause inside an *infinite loop*. Use a *constant* as the value for the *limit* clause to define the batchsize; 1000 is a reasonable value. Fetch into a *varray* declared with the same size. Don't test the *%NotFound* cursor attribute to terminate the loop. Instead, use *exit when Results.Count() < Batchsize;* as the last statement in the loop; ensure correct processing in the edge case that the last fetch gets exactly zero rows. When embedded SQL is sufficient, use an *explicit cursor*. When you need native dynamic SQL, use a *cursor variable*.

Selecting many rows — bounded result set

Another common use for database PL/SQL programs to is fetch a master row and all its associated detail rows. The canonical masters example is the *Orders* table; and the canonical details example is the *Order_Line_Items* table. In the case that these tables support the back-end of an internet shopping site, it is safe to assert a maximum number of line items, say one thousand, that an order may have and to implement a business rule to ensure that this is not exceeded[70]. Should the *batched bulk fetch* approach be used with the recommended batchsize, then there would never be more than one batch to fetch. Therefore, that approach is unnecessary and, instead, the simpler *entire bulk fetch* can be used.

Code_32 shows an example using embedded SQL[71].

```
-- Code_32
declare
  Target_Varray_Too_Small exception;
  pragma Exception_Init(Target_Varray_Too_Small, -22165);
begin
  select             a.PK, a.v1
  bulk collect into  x.Results
  from               t a
  where              a.PK > x.Some_Value
  order by           a.PK;
exception when Target_Varray_Too_Small then
  Raise_Application_Error(-20000,
    'Fatal: Business Rule 12345 violated.');
end;
```

---

70. It is very unlikely indeed that someone ordering books or DVDs on line would mange to fill his shopping cart with 1,000 items. And, should that happen, it is even less likely that a polite message saying "The shopping cart can't hold more than 1,000 items. Please check out and then start to fill another cart. You will receive a 10% discount on items in the next cart if you order them today." would result in much customer dissatisfaction or loss of business.

71. No attempt is made in this paper to invent realistic examples. We assume that the reader has enough experience not to need this. Our focus is the various techniques for doing SQL from PL/SQL — and we claim that neutral examples are more helpful than ones that clutter the discussion with extraneous detail.

The *ORA-22165* error occurs when you attempt to access an element in a *varray* using an index that is not in the range *1..<varray size>*. It is useful, as a proactive bug diagnosis technique, to write the *entire bulk fetch* statement in a tightly surrounding *block statement* that provides a handler for this error.

Code_33 shows an example using native dynamic SQL.

```
-- Code_33
declare
  Stmt constant varchar2(200) := '
    select    a.PK, a.v1
    from      t a
    where     a.PK > :b1
    order by  a.PK';
  Target_Varray_Too_Small ...
begin
  execute immediate Stmt
    bulk collect into Results
    using Some_Value;
exception when Target_Varray_Too_Small then
  Raise_Application_Error(-20000,
    'Fatal: Business Rule 12345 violated.');
end;
```

Notice that each example uses a cursor-less PL/SQL construct. Just for the sake of comparison, *Code_34* shows the *entire bulk fetch* flavor that uses an *identified cursor*, *Cur*.

```
-- Code_34
-- Cur is already open here.
-- The fetch syntax is the same for
-- an explicit cursor and a cursor variable.
fetch Cur bulk collect into Results;
close Cur;
```

The same code as is shown in *Code_31* is used to establish *Cur*.

The total code volume required for the *Code_34* approach is noticeably greater than *Code_32* or *Code_33*, to which these versions are functionally equivalent. The theoretical difference is that the use of an *identified cursor* allows the code that defines the *select* statement and the code that fetches from it to be split between different modules. However, this paper claims, in *"Approaches for producer/consumer modularization"* on page 41, that such a modularization scheme is, in general, inappropriate.

Readers whose religion bans hard coded limits that, when exceeded cause fatal errors, must simply always use *batched bulk fetch*[72]. However, they should not forget the case that the *select* statement is designed, as *Code_35* shows, specifically

72. Enhancement request 6616605 asks for a new PL/SQL compiler warning when *entire bulk fetch* (in any of its three flavors, *select... bulk collect into*, *execute immediate... bulk collect into*, or *fetch... bulk collect into*) is used.

to get the Nth slice of an unbounded result set. Here, at least, there is no doubt that *entire bulk fetch* can be used safely.

```
-- Code_35
-- Set the lower and upper bound of the slice.
lb := Slice_Size*(Slice_No - 1) + 1;
ub := lb + Slice_Size - 1;

with Unbounded as (
  select    a.PK, a.v1, Rownum r
  from      t a
  order by  a.PK
  )
select              Unbounded.PK, Unbounded.v1
bulk collect into   b.Results
from                Unbounded
where               Unbounded.r between b.lb and b.ub;
```

We shall return to a discussion of the slicing of an unbounded result set in *"Approaches for producer/consumer modularization"* on page 41.

The discussion in this section is summarized in this best practice principle:

When you do know how the maximum number of rows your query might get, use *select... bulk collect into* or *execute immediate... bulk collect into* to fetch all the rows in a single step.

Principle_12

When many rows are to be selected, and the result set can be safely assumed to be of manageable maximum size, fetch all the rows in a single step. Use the cursor-less PL/SQL constructs *select... bulk collect into* when embedded SQL is possible, and *execute immediate... bulk collect into* when dynamic SQL is needed. Fetch into a *varray* declared with the maximum size that you are prepared to handle. Implement an exception handler for *ORA-22165* to help bug diagnosis.

Selecting many rows — *select list* or binding requirement not known until run-time

It is common, when implementing information systems that support job functions within organizations, to represent the main information entity as a single database table with many columns. (Of course, there may also be various detail tables.) An obvious example is a personnel system that records, for each employee, the kinds of facts that the famous *HR.Employees* does. A common requirement for such systems is to support an end-user query interface where a match condition can be entered for any entity-attribute (which maps to a column in the table), and where conditions can be left blank if they are not interesting. Oracle Corporation has such an internal system and the query screen for this functionality has about a dozen such fields. Because each of the attributes is a text value (like last name, first name, job title, and so on) the conditions never need to involve tests like "greater than" or "between" — but other similar systems might have numeric attributes like salary where such tests were useful. Even with the luxury that each test corresponds to a *like* predicate in the ultimate *select* statement's *where* clause, the number of distinct where clauses that the query screen might require is very much too big[73] to support each one with a SQL statement whose text is fixed, and therefore whose binding requirement is known, at compile- time.

---

73. For *N* columns, the number of distinct *where* clauses is equal to the sum of the number of ways to choose 1 from *N*, to choose 2 from *N*, and so on, up to the number of ways to choose *(N-1)* from *N* and *N* from *N*. This is 2^N*-1*. For 12 columns, this is 4095.

There is a large class of applications where, while the query screen does require flexibility for choosing the criteria of interest, the display of the results is fixed by the functional specification. For the implementation, this means that the *select list* is known at compile time. For such use cases, while the *DBMS_Sql* API is needed to support the binding, native dynamic SQL can be used to get the results. (This is simpler to program and runs faster.) *Code_36* shows the approach[74].

```
-- Code_36
DBMS_Sql_Cur := DBMS_Sql.Open_Cursor(Security_Level=>2);

-- Build the select statement
...
DBMS_Sql.Parse(DBMS_Sql_Cur, Stmt, DBMS_Sql.Native);

-- More eleborate logic is needed when the values to be bound
-- are not all the same datatype.
for j in 1..No_Of_Placeholders loop
  DBMS_Sql.Bind_Variable(
    DBMS_Sql_Cur, ':b'||To_Char(j), Bind_Values(j));
end loop;
Dummy := DBMS_Sql.Execute(DBMS_Sql_Cur);

declare
  Cur_Var Sys_Refcursor :=
    DBMS_Sql.To_Refcursor(DBMS_Sql_Cur);
begin
  loop
    fetch Cur_Var bulk collect into Results limit Batchsize;
    for j in 1..Results.Count() loop
      Process_One_Record(Results(j));
    end loop;
    exit when Results.Count() < Batchsize;
  end loop;
  close Cur_Var;
end;
```

The critical feature of the use of the *DBMS_Sql* API for this use case is that it allows facts that are first known at run time to determine the control flow and the actual arguments for the invocation of *Bind_Variable()*. Such a scheme is impossible with native dynamic SQL because the *using* clause (in either the *execute immediate... into* statement or the *open Cur for* statement, when *Cur* is a *cursor variable*) is frozen at compile time[75].

Notice the use of the *batched bulk fetch*. The use case tends to indicate that the result set will be unbounded.

The logic for building the SQL statement will be application specific. The general plan is to test each *in* formal parameter that represents a user-entered criterion. When it is not null, then text is appended to the SQL statement to express the condition it denotes. The unit of appended text has the form

74. The approach depends on using the *DBMS_Sql.To_Refcursor()* function. This is new in Oracle Database 11*g*. This could be useful also when the overall system architecture prefers that the fetching is implemented in the database client even when knowledge of, and security surrounding, the SQL statement are hidden in the database. That, of course, requires that the database PL/SQL subprogram that the client calls has a return datatype based on *ref cursor*. This is possible, now, even when the binding requires the *DBMS_Sql* API. However, we argue in *"Approaches for producer/consumer modularization"* on page 41 that other approaches are usually better.

and cn = :bm where *cn* is the denoted column and *m* is a runner that denotes the ordinal value of the current condition. *Code_37* Sketches the approach.

```
-- Code_37
for j in 1..User_Criteria.Count() loop
  if User_Criteria(j) is not null then
    No_Of_Placeholders := No_Of_Placeholders + 1;
    Stmt := Stmt||' and '||
      Column_Names(j)||' = :b'||To_Char(No_Of_Placeholders);
    Bind_Values(No_Of_Placeholders) := User_Criteria(j);
  end if;
end loop;
```

It is good practice, as has been explained, to represent the starting string for the SQL statement with a *constant*[76]. The real logic would need to be more complicated if the SQL statement should use = when a bare criterion is entered and should use *like* when it contains wildcard characters (% and _). And it would be yet more complicated if the columns of interest had a mixture of datatypes like *varchar2*, *number*, and *date*. and if the user could specify inequality operators (say, by choosing from a pop-up list).

Sometimes, and usually in connection with information systems that support job functions within organizations, the requirements specification states that the user must be able to configure which attributes are included in the report. For the implementation, this means, of course, that the *select list* is not known at compile

---

75. We have seen cases where customers have attempted to overcome the fact that the *using* clause is frozen at compile time by programatically generating and executing an *anonymous PL/SQL block* that, in turn, uses native dynamic SQL. They usually feel so pleased with their solution — that they have overcome a supposed limitation by building the *using* clause programatically — that they fail to see that the ingenuity of the approach disguises its shortcomings. The approach requires that the values which would be dynamically bound with *DBMS_Sql.Bind_Variable()* be, instead, encoded as literals in the text of the *anonymous PL/SQL block*. (To avoid that, you would need to execute the *anonymous PL/SQL block* using the *DBMS_Sql* API, which would subvert the point of the device.) This, in turn, means that each generated *anonymous PL/SQL block* will differ textually from the previous one; the consequence, of course, is a *hard parse* for each execution.

    We have even seen this approach proposed as a best practice. Be warned: it is a classic example of a worst practice!

76. A popular device is to append *where 1=1* to the *constant* text before starting the loop so that the concatenation logic can be more straightforward.

time and so the the *DBMS_Sql* API must be used, too, used to get the results. *Code_38* sketches the approach.

```
-- Code_38
declare
  ...
  type Results_t is table of DBMS_Sql.Varchar2_Table
    index by pls_integer;
  Results Results_t;
begin
  -- Open the DBMS_Sql_Cur,
  -- build and parse the select statement,
  -- bind to the placeholders, and execute as in Code_36.
  -- This will set No_Of_Select_List_Items.
  ...
  loop
    -- Tell it to fill the target arrays
    -- from element #1 each time.
    for j in 1..No_Of_Select_List_Items loop
      DBMS_Sql.Define_Array(
        DBMS_Sql_Cur, j, Results(j), Batchsize, 1);
    end loop;

    Dummy := DBMS_Sql.Fetch_Rows(DBMS_Sql_Cur);

    for j in 1..No_Of_Select_List_Items loop
      -- Have to delete explicitly. NDS does it for you.
      Results(j).Delete();
      DBMS_Sql.Column_Value(DBMS_Sql_Cur, j, Results(j));
    end loop;

    for j in 1..Results(1).Count() loop
      -- Process the results.
      ...
    end loop;

    exit when Results(1).Count() < Batchsize;
  end loop;
  DBMS_Sql.Close_Cursor(DBMS_Sql_Cur);
end;
```

We limit the ambition level of *Code_38* to just a sketch because a complete solution is voluminous. The fact that the *select list* items are in general of different datatypes requires a very elaborate approach if targets of corresponding datatypes are to be used. However, when the ultimate aim of the processing is to prepare a human-readable report, the approach can be greatly simplified by converting each *select list* item into a *varchar2*, ideally using appropriate *To_Char()* function invocations in the *select list*. Now an *index by pls_integer* table whose element datatype is the supplied *DBMS_Sql.Varchar2_Table* collection can be used as the fetch target. This means that, once *No_Of_Select_List_Items* is known (as a result of building the *select list*), then *Define_Array()* and *Column_Value()* can be invoked in loops that apply the same operation to each *select list* item. Of course, the comment "Process the results" implies tortuous programming; but the logic is quite ordinary, and no longer has anything to do with the SQL processing.

Notice that, here, the use of the *DBMS_Sql* API allows facts that are first known at run time to determine the control flow and the actual arguments for the invocation of *Define_Array()* and *Column_Value()*. Such a scheme is impossible with native dynamic SQL because the *into* clause (in either the *execute immediate... into* statement or the *fetch... bulk collect into* statement) is frozen at compile time.

It is conceivable, though unlikely, that the requirements specified for an application lead to an implementation design where the binding requirements are

known at compile time but the composition of the *select list* is not known until run time. This allows native dynamic SQL to be used to open a *cursor variable* and the *DBMS_Sql* API to be used only where it is needed — to handle the fetching of the results. For completeness, *Code_39* shows this approach.

```
-- Code_39
open Cur_Var for Stmt using Some_Value;
DBMS_Sql_Cur := DBMS_Sql.To_Cursor_Number(Cur_Var);

-- The fetch loop is identical to that shown in Code_38.
loop
  for ... loop
    DBMS_Sql.Define_Array(..., Results(j), ...);
  end loop;

  Dummy := DBMS_Sql.Fetch_Rows(DBMS_Sql_Cur);

  for ... loop
    ...
    DBMS_Sql.Column_Value(..., Results(j));
  end loop;

  for j in 1..Results(1).Count() loop
    -- Process the results.
    ...
  end loop;

  exit when Results(1).Count() < Batchsize;
end loop;
```

Notice that if the user-interface is required to allow specifying the criteria by which to order the results, this adds no further complexity because this has no effect on the implementation of the binding (the *order by* clause will not use placeholders) or of the fetching.

The discussion in this section is summarized in this best practice principle:

Use the the *DBMS_Sql* API when you don't know the binding requirement of what the *select list* is until run time. If you do, at least, know the *select list*, use *To_Refcursor()* and then *batched bulk fetch*.

Principle_13

Avoid temptation to compose the *where* clause using literals — and especially to concatenate a *where* clause which has been explicitly typed by the user. The performance is likely to be noticeably worse than an approach that binds to placeholders; and, with a directly typed *where* clause, it isn't feasible to use *Sys.DBMS_Assert.Enquote_Literal()* to inoculate against SQL injection. When the binding requirement is not known until run time, use the *DBMS_Sql* API to parse, bind, and execute the SQL statement. If the *select list* is known at compile time, use *To_Refcursor()* to transform the *DBMS_Sql numeric cursor* to a *cursor variable* and then use *batched bulk fetch*. If the *select list* is not known until run time, use the *DBMS_Sql* API to fetch the results too. In the unlikely case that the binding requirement is known at compile time but the *select list* is not known until run time, use native dynamic SQL to open a *cursor variable* and then use *To_Cursor_Number()* to transform the *cursor variable* to a *DBMS_Sql numeric cursor*. Then use the *DBMS_Sql* API to fetch the results.

Selecting a single row

The obvious example of this use case is getting a row that is identified by its primary key. However, it is perhaps less common than might at first be imagined: if the row comes from a master table it is very likely that its associated details will be required at the same time; and if the row comes from a detail table, it is very likely that all rows for a particular master will be required at the same time. Nevertheless, there are some use cases where just a single row is required.

Code_40 shows an example using embedded SQL.

```
-- Code_40
select    a.PK, a.v1
into      b.The_Result
from      t a
where     a.PK = Some_Value;
```

Code_41 shows an example using native dynamic SQL.

```
-- Code_41
declare
  Stmt constant varchar2(200) := '
    select    a.PK, a.v1
    from      t a
    where     a.PK = :b1';
begin
  execute immediate Stmt
    into The_Result
    using Some_Value;
```

Notice the symmetry between *Code_40* and *Code_32* and between *Code_41* and *Code_33*; each example uses a cursor-less PL/SQL construct. And, as with *Code_34* and again just for the sake of comparison, *Code_42* shows the *entire bulk fetch* flavor that uses an *identified cursor*, *Cur*.

```
-- Code_42
-- Cur is already open here.
-- The fetch syntax is the same for
-- an explicit cursor and a cursor variable.
fetch Cur into The_Result;
close Cur;
```

Code_43 shows the three ways to establish the *identified cursor Cur* so that *Code_42* is viable. It is very similar to that shown in *Code_31* for the multirow case.

```
-- Code_43
  $if $$Approach = 1 $then
    cursor Cur is
      select    a.PK, a.v1
      from      t a
      where     a.PK = b.Some_Value;
  $elsif $$Approach = 2 or $$Approach = 3 $then
    Cur Sys_Refcursor;
    $if $$Approach = 3 $then
      Stmt constant varchar2(200) := '
        select    a.PK, a.v1
        from      t a
        where     a.PK = :b1';
    $end
  $end
begin
  $if $$Approach = 1 $then
    open Cur;
  $elsif $$Approach = 2 $then
    open Cur for
      select    a.PK, a.v1
      from      t a
      where     a.PK = b.Some_Value;
  $elsif $$Approach = 3 $then
    open Cur for Stmt using Some_Value;
  $end
```

As with the multirow case, the total code volume required for the *Code_42* approach is noticeably greater than *Code_40* or *Code_41*, to which these versions are functionally equivalent. The corresponding arguments apply[77].

Notice that the very nature of a scenario where the binding requirement is not known until run time determines that the query is likely to return more than one row. So the *DBMS_Sql* API is never appropriate for the case where the

No_Data_Found exception is regrettable and the *Too_Many_Rows* exception is unexpected. The case where exactly one row is expected but the *select list* is not known until run time seems so unlikely that, again, the *DBMS_Sql* API would not be required.

The discussion in this section is summarized in this best practice principle:

To get exactly one row, use *select... into* or *execute immediate... into*. Take advantage of *No_Data_Found* and *Too_Many_Rows*.

Principle_14
When exactly one row is to be selected, fetch the row in a single step. Use the cursor-less PL/SQL constructs *select... into* when embedded SQL is possible, and *execute immediate... into* when dynamic SQL is needed. Take advantage of the regrettable *No_Data_Found* exception and the unexpected *Too_Many_Rows* exception.

Approaches for producer/consumer modularization

When designing the modularization of database PL/SQL programs, it is common to choose to implement these two different kinds of processing in different PL/SQL units:

- the execution of the SQL statements (for *select*, *insert*, *update*, *delete*, and *merge*, *lock table*, *commit* or *rollback*)
- the processing of the data that is retrieved or used to make changes in tables.

When the discussion is limited to just the *select* operation (which is this major section's focus), then the producer/consumer metaphor applies nicely: ideally, the producer is responsible for all the SQL and the consumer knows nothing of it. We shall see, however, that such a clean distinction has too many practical drawbacks (at least through Oracle Database 11*g*) to make it the recommended approach when the consumer is outside of the database.

An obvious reason for at least some separation of duties is to enforce an access control regime. Many business rules (for example, what kinds of data-dependent changes are allowed, and how denormalizations are to be maintained) are most safely and effectively implemented by limiting all access to table data to dedicated data-access PL/SQL units. This is easily enforced by using definer's rights data-access PL/SQL units in the same schema as the tables and by implementing the PL/SQL units that consume or prepare this data in other schemas. The *Execute* privilege on the data-access PL/SQL units is granted to the other schemas, but the *Select*, *Insert*, *Update*, and *Delete* privileges on the tables are not. The same scheme is useful for access by clients of the database. This best practice principle is self-evidently beneficial:

Expose your database application through a dedicated schema that has only private synonyms for the objects that define its API.

Principle_15
Create a dedicated schema, say *Some_App_API*, for an application to which database clients will connect in order to access that application's functionality. Implement all the application's database objects in schemas other than

77. Some users have a superstitious belief that *fetch... into* is to be preferred to *select... into* or *execute immediate... into* for asserted reasons of performance and functionality. These reasons don't stand up to scrutiny: *fetch... into* performs worse than *select... into* and *execute immediate... into*; and when the requirements are to get exactly one row using a unique key, the possibility of the *Too_Many_Rows* exception is only helpful. Typically, the *No_Data_Found* exception, though regrettable, is not unexpected; recovery is possible, and this should be programmed in a tightly enclosing exception handler.

Some_App_API, with closely guarded passwords, and limit the objects in *Some_App_API* to just private synonyms. Create these synonyms for exactly and only those of the application's database objects that are intended to expose its client API. Grant to *Some_App_API* only those privileges that are needed to allow the intended access to these objects.

Many of Oracle Corporation's major customers tighten the regime by adopting this best practice principle:

Principle_16
Restrict the object types that *Some_App_API*'s private synonyms exposes to just PL/SQL units. Hide all the tables in other schemas, and do not grant any privileges on these to *Some_App_API*[78].

Expose your database application through a PL/SQL API. Hide all the tables in schemas that the client to the database cannot reach.

When the task is to retrieve table data, how should the producer/consumer API be designed? For a within-database modularization scheme, there are three possibilities:

- Expose the declaration of a parameterized *explicit cursor* in a package spec and hide its definition in the body. Let the client manage opening, fetching from, and closing the cursor.
- Expose the declaration of a parameterized function whose return datatype is based on *ref cursor* in a package spec and hide its definition in the body. Let the client manage fetching from and closing the *cursor variable* to which the function's return is assigned.
- Expose the declaration of a function whose return datatype is designed to represent the data that is to be retrieved in a package spec and hide its definition in the body. The return datatype is naturally implemented as a *record* (when the parameterization always denotes a single row), or as a collection of *records* or an XML document when the parameterization denotes many rows.

For defining the API that the database exposes to a consumer outside of the database, the first approach is viable only when the client is implemented in PL/SQL — in other words, when it is Oracle Forms. It also suffers from the fact that it supports only those query requirements that can be satisfied using embedded SQL. It turns out that there is no need to prefer it even when the consumer is inside the database, and so we shall give it no more attention.

The second approach is always viable for defining the API that the database exposes to a consumer outside of the database. All client environments that support the processing of SQL statements in Oracle Database include APIs that support the execution of an *anonymous PL/SQL block* and in particular, they allow an appropriate client datastructure to be bound to a placeholder that, were it written as a variable in the block, would be declared as a *cursor variable*. In spite of the fact the this approach (because the client needs to know something of SQL

78. This is nothing other than a specialization, for Oracle Database, of a universal best practice principle of software engineering: decompose your system into modules; expose each module's functionality, at a carefully designed level of abstraction, with a clean API; and hide the module's implementation behind that API. In Oracle Database, PL/SQL subprograms provide the means to define an API; and tables and the SQL statements that manipulate their contents are clearly part of a module's implementation, and should be hidden from clients to the database.

mechanics[79] in order to implement the fetching from the *ref cursor*) arguably breaks the theoretically ideal modularization concept, many customers prefer it and use it successfully in mission-critical production code. They find the theoretical ideal (which needs uses collections of *ADT*s) too cumbersome[80].

The third approach needs some modification to make it generally viable for defining the API that the database exposes to the client. Unfortunately, not all of these APIs support the binding of an appropriate client datastructure to a placeholder that plays the role of the actual argument in the invocation of a subprogram where the corresponding formal parameter is a *record* or collection of *records*. However, they all support binding when the target formal parameter is an *ADT*[81] or collection of *ADTs*. *"Appendix C: alternative approaches to populating a collection of records with the result of a select statement"* on page 71 shows a few constructs for this purpose. Notice that the choice of an XML document as the return datatype is particularly interesting when the client's purpose is to prepare a human-readable report.

The third is the theoretically cleanest API design. The producer encapsulates everything to do with the retrieval of the data and the consumer sees only the data itself in a suitably specified representation.

It is interesting to review, in the context of this third approach, how to fetch an unbounded result set first in the context of a stateful relationship between consumer and producer and then in the case that the relationship is stateless[82].

Stateful producer/consumer relationship

In the stateful case, where for example, the consumer and producer are implemented as PL/SQL subprograms in packages in different schemas[83], what is the state, and who holds it? The state is, of course, the member of the result set which is next to be fetched and it is held in the *cursor variable*[84]. However, through Oracle Database 11*g*, a *cursor variable* may not be declared at global level in a package spec or body[85]. The solution is that the producer must hand back a

---

79. In particular, the consumer needs to implement the equivalent of *entire bulk fetch* or *batched bulk fetch* in its programming environment.

80. We will, nevertheless, explain how to implement this.

81. We use *ADT* in this paper as a convenient synonym for the result of *create type... as object(...)*. It is too confusing to refer to it as an object whose *Object_Type* is *type* and that is an *object type* rather than a *collection type*!

82. Of course, the relationship is automatically stateful when the consumer is inside the database. And it is often stateless when the consumer is outside the database.

83. These days, it is relatively uncommon to in a *de novo* project to build a user-facing application, to choose an architecture where the user interface has a stateful connection to the database. The norm, almost without exception, is to implement the user interface in a stateless HTML browser. One notable exception is an IDE for database development, for example Oracle Corporation's Sql Developer.

84. Here we appreciate the metaphorical meaning of the term *cursor*.

85. There is no fundamental reason for this restriction. Enhancement request 6619359 asks to lift it.

handle to the state so that the consumer can hang onto it until all batches have been processed[86]. *Code_44* shows the spec of the *Producer* package.

```
-- Code_44
package Producer is
  type Result_t is record(PK t.PK%type, v1 t.v1%type);
  type Results_t is table of Result_t index by pls_integer;
  function The_Results(
    Some_Value in t.PK%type,
    Cur_Var in out Sys_Refcursor)
    return Results_t;
end Producer;
```

And *Code_45* shows its body.

```
-- Code_45
package body Producer is
  function The_Results(
    Some_Value in t.PK%type,
    Cur_Var in out Sys_Refcursor)
    return Results_t
  is
    Stmt constant varchar2(200) := '
      select    a.PK, a.v1
      from      t a
      where     a.PK > :b1
      order by  a.PK';
    Batchsize constant pls_integer := 1000;
    Results Results_t;
  begin
    if Cur_Var is null then
      open Cur_Var for Stmt using Some_Value;
    end if;

    fetch Cur_Var bulk collect into Results limit Batchsize;

    if Results.Count() < Batchsize then
      close Cur_Var;
      Cur_Var := null;
    end if;

    return Results;

  exception when others then
    if Cur_Var%IsOpen then
      close Cur_Var;
    end if;
    raise;
  end The_Results;
end Producer;
```

Code_46 shows the *Consumer* procedure.

```
-- Code_46
procedure Consumer is
  Some_Value constant t.PK%type := 0;
  Cur_Var Sys_Refcursor := null;
  Results Producer.Results_t;
begin
  loop
    Results := Producer.The_Results(Some_Value, Cur_Var);
    for j in 1..Results.Count() loop
      ...
    end loop;
    exit when Cur_Var is null;
  end loop;
end Consumer;
```

If you can be certain that there will never be a requirement to use native dynamic SQL, then you could instead use an *explicit cursor* declared at top

86. This is a very common paradigm. It is used by, for example, *DBMS_Sql* and by *Utl_File*.

level in the package body[87]. This would simplify the design because the consumer would not need to "hold on" to the *ref cursor* between calls to the producer. I chose to use a *ref cursor* because the approach works both for embedded SQL and for native dynamic SQL. Of course, if the requirements forced the use of the *DBMS_Sql* API, then the design of the producer might need to be radically different. Here, the state is the number that was returned by *Open_Cursor()* and this can easily be held in a variable declared at top level in the body of the producer package. Apart from the fact that, again, the consumer would not need to "hold on" to the *ref cursor* between calls, the consumer would be agnostic about these difference in the implementation of the producer. A defensive design might expose a *ref cursor* at the API for all three methods to implement the producer and simply let it have no significance when the implementation used an *explicit cursor* or the *DBMS_Sql* API[88].

Stateless producer/consumer relationship

In Oracle Database applications that implement the user interface in a stateless HTML browser, it is very common to show query results in batches with "next page" and "previous page" buttons or with buttons that allow an immediate jump to the Nth page. The architecture implies that each page view request is satisfied by a call from the midleware to the database. This database call is serviced by a different session than serviced the call for the previous page view[89]. This means, then, that the query that gets the required page must simply be parameterized to do just that. Unlike in the stateful regime, it can't carry on from where it had got to last time. *Code_35* shows how such a query is written. It is a perfect match for the *entire bulk fetch* approach.

The discussion in this section is summarized in this best practice principle:

Define the producer/consumer API as a function whose return datatype represents the desired data. Hide all the SQL processing in the producer module. That way, the consumer is immune to an implementation change that a requirements change might cause. Simply parameterize the producer function as you would parameterize the query. This approach accommodates getting the rows in batches or getting all the rows in one call — where this might be a slice.

Principle_17

Define the producer/consumer API as a function whose return datatype represents the data that is produced. Hide everything to do with the SQL processing, including the fetching, in the producer module. When the query parameterization specifies exactly one row, use a *record* or *ADT* with the same shape as the *select list*. In this case, use one of the cursor-less PL/SQL constructs *select... into* or *execute immediate... into* depending on the requirement for dynamic SQL. When the query parameterization specifies many rows, use a collection of *records* or a collection of *ADTs*. In this case, use *entire bulk fetch* when this is certain to be safe; this allows the use of one of the cursor-less PL/SQL constructs *select... bulk collect into* or *execute immediate... bulk collect into*. If *entire bulk fetch* is unsafe, use *batched bulk fetch* when the producer/consumer relationship is stateful. This requires a *identified cursor*. If embedded SQL if sufficient, then use an *explicit cursor* declare

87. Notice that none of the approaches shown in this paper have capitalized on the ability to separate the declaration of an *explicit cursor* in the package specification from its definition in the package body.

88. Recall, too, the existence from Oracle Database 11*g*, of the *DBMS_Sql.To_Refcursor()* function. Provided that the composition of the *select list* is known at compile time, more straightforward fetch code can be written using native dynamic SQL.

89. This paradigm deliberately relinquishes the classical preference to see read-consistent query results.

at global level in the body of the producer package. This will hold state across the calls from the consumer to get each batch. If dynamic SQL is needed, use a *cursor variable*. Pass this back to the consumer with each batch of results so that the consumer can hold on to it. When the producer/consumer relationship is stateless, use result set slicing. Implement the delivery of each slice with *entire bulk fetch*. If requirements dictate it, use the *DBMS_Sql* API and hide all of the code that uses this in the producer module.

Oracle Database 11*g* brings the PL/SQL function result cache. Briefly, the programmer uses the keyword *result_cache* (in the same syntax spot where *authid* is used) to hint that, for every distinct combination of actual arguments with which the function is invoked, the return value should be cached to save the cost of recomputing it on a subsequent invocation with the same actual arguments. The cache is accessible to all sessions. Significantly, the return datatype may be based on a *record* or an *ADT* or a collection of one of these. This is the PL/SQL flavor of *memoization*[90], tailored for the case that the calculation of function's return value relies upon data retrieved from tables. The declarative *relies_on* clause lets the programmer list the tables whose contents affect the function's result. If changes are committed to any of these tables, then the cached results for the function are purged. When the producer function returns a single row or a small number of rows, and when the table data on which the function relies changes infrequently, performance can be significantly improved by marking the function with *result_cache*. A very obvious application of this is the function that returns the mapping between a surrogate primary key and the corresponding human-readable unique key for populating a list of values user-interface control.

90. The technique is well known in software engineering. Wikipedia provides an account that starts like this: *In computing, memoization is an optimization technique used primarily to speed up computer programs... coined by Donald Michie in 1968... A memoized function "remembers" the results corresponding to some set of specific inputs. Subsequent calls with remembered inputs return the remembered result rather than recalculating it...*

APPROACHES FOR *INSERT*, *UPDATE*, *DELETE*, AND *MERGE* STATEMENTS

It is relatively common that user interaction requires a single row *insert*, *update*, *delete*, or *merge*, so this use case is treated first. Unlike for *select*, these operations, for the multirow case, need never be supported for an unbounded number of rows. Because the data that represents the intended operation arises first in PL/SQL datastructures, it can be batched appropriately for the bulk multirow operation.

The code illustrations in this section use embedded SQL and, as we shall see, the statements are always singleton cursor-less PL/SQL constructs; there is no possibility to use an *identified cursor* and so the operations all use an *implicit cursor*. This means that the transition, should the use case require it, to native dynamic SQL is relatively mechanical. This is discussed briefly in *"Using native dynamic SQL for insert, update, delete, and merge"* on page 57. We don't discuss the use of the *DBMS_Sql* API for these kinds of SQL statement; rather, we show how this can be avoided in use cases where your first reaction might be to believe that it is needed[91].

Single row operation

The challenge that the single row scenario presents, for *insert* and *update*, is to handle that case that values are specified for only some of the columns.

Single row *insert*

Code_47 shows two possible cases for *insert*.

```
-- Code_47
insert into t(PK, n1) values (b.PK, b.n1);
...
insert into t(PK, v1) values (b.PK, b.v1);
```

Consider the design and implementation of a procedure *Insert_Row_Into_T()* with the declaration[92] shown in *Code_48*.

```
-- Code_48
procedure Insert_Row_Into_T(
  PK in t.PK%type,
  n1 in t.n1%type := null,
  n1_Specified in boolean := false,
  ...
  v1 in t.v1%type := null,
  v1_Specified in boolean := false,
  ...)
  authid Current_User;
```

The discussion in *"Selecting many rows — select list or binding requirement not known until run-time"* on page 35 showed that, with about a dozen columns, each of which may or not be mentioned in the SQL statement, there is a combinatorial explosion: very many more than 1,000 distinct embedded SQL statements would

91. Anyone who has mastered the use of the *DBMS_Sql* API for *select* statements where the binding requirement or the composition of the *select list* are unknown until run time will find using it for *insert*, *update*, *delete*, or *merge* statements no harder.

92. You might wonder why every formal that corresponds to a column for which the value is optional has a partner *boolean* to indicate if it was specified. Why not just test in the procedure to see if the parameter was defaulted by testing if it *is null*? The answer is that, in general, it's possible that *null* might be the deliberately intended value.

be needed to meet all possible cases. Of course, just as with the *select* statement for which the binding requirement isn't known at compile time, it would be possible to use the the *DBMS_Sql* API. But recall the best practice principle (see page 17) that instructs you to avoid this when there's a good alternative. The alternative in this case is to use a statement that specifies every column value and that uses the column default values for those columns that are not specified.

Oracle9*i* Database Release 2 introduced support for using a *record* as a bind argument in embedded SQL. This allows a compact approach to this challenge. *Code_49* shows one way to access the default values[93/ 94].

```
-- Code_49
insert into  t(PK)
values       (PK)
returning    PK, n1, n2, v1, v2
into         New_Row;

if n1_Specified then
  New_Row.n1 := n1;
end if;
...
update  t
set     row = New_Row
where   t.PK = New_Row.PK;
```

This approach, though, has the disadvantage that two PL/SQL to SQL to PL/SQL context switches[95] are needed when one ought to be sufficient. A better way to obtain the default values is to use a *record* type that defines them. Programmers sometimes forget that the anonymous *t%rowtype* and, correspondingly, items like *t.PK%type* do not inherit column constraints and default values from the table[96]. However, this need not be a problem if a discipline is followed when building the application's installation and patch/upgrade scripts. *Code_81* (on page 73) shows the approach that was used for the test table *t* that is used for the examples in this paper. This suggests the following best practice principle:

For each application table, maintain a template *record* type that defines the same constraints and defaults.

Principle_18

Maintain a package that exposes a *record* type declaration for each of the application's tables. The declaration must repeat the specification of column name, datatype, constraint, and default value that characterizes the table.

---

93. It might be more efficient to return the *Rowid*, but this would prevent the use of a *record*.

94. You might be tempted to write *returning row into A_Record*. This syntax is not supported. Enhancement request 6621878 asks that it be.

95. See *"Selecting many rows — unbounded result set"* on page 30.

96. The reasons for this are rather subtle. For example, a schema level table may have a column with a *not null* constraint that doesn't have a default value. (You get an error on *insert* if you don't provide a value.) But PL/SQL insists that a *not null* variable or *record* field has a default. Other complexities would arise if the table column a had a check constraint.

With the template *record* type in place, *Code_49* can be rewritten as *Code_50*.

```
-- Code_50
  New_Row Tmplt.T_Rowtype;
begin
  New_Row.PK := PK;

  if n1_Specified then
    New_Row.n1 := n1;
  end if;
  ...
  insert into t values New_Row;
```

Single row *update*

Code_51 shows two possible cases for *update*.

```
-- Code_51
update  t a
set     a.n1 = b.n1
where   a.PK = b.PK;
...
update  t a
set     a.v1 = b.v1
where   a.PK = b.PK;
```

Consider the design and implementation of a procedure *Insert_Row_Into_T()* with the declaration[97] shown in *Code_52*.

```
-- Code_52
procedure Update_T_Row(
  PK in t.PK%type,
  n1 in t.n1%type := null,
  n1_Specified in integer := 0,
  ...
  v1 in t.v1%type := null,
  v1_Specified in integer := 0
  ...)
  authid Current_User;
```

Of course, an implementation that tried to use embedded SQL statements like those shown in *Code_51* would suffer from a combinatorial explosion. One approach, in the same spirit as *Code_49*, is to retrieve the intended new row into a *record*, to change only the specified fields, and then to use *update... set row...* *Code_53* shows this[98].

```
-- Code_53
select  * into The_Row from t a
where   a.PK = Update_T_Row.PK
for update;

if n1_Specified then
  The_Row.n1 := n1;
end if;
...
update t a set row = The_Row where a.PK = Update_T_Row.PK;
```

97. The reason for declaring *n1_Specified* and so on as *integer* will soon be clear.

98. *Code_53* uses a version of *Update_T_Row()* where *n1_Specified* and so on are *boolean*.

Code_54 shows an approach[99] that avoids the double PL/SQL to SQL to PL/SQL context switch.

```
-- Code_54
update   t a
set      a.n1 = case n1_Specified
                  when 0 then a.n1
                  else        Update_T_Row.n1
                end,
         a.v1 = case v1_Specified
                  when 0 then a.v1
                  else        Update_T_Row.v1
                end
where    a.PK = Update_T_Row.PK;
```

The reason the *n1_Specified* and so on are declared as *integer* is now clear: SQL does not understand the *boolean* datatype.

Single row *delete*

A single row must be identified by a unique key, so the dilemma of a binding requirement that isn't known until run time doesn't arise. For completeness, *Code_55* shows an example. (It is sometimes required to keep an audit of all the values in to deleted row).

```
-- Code_55
  Old_Row t%rowtype;
begin
  delete     from t a
  where      a.PK = b.PK
  returning  a.PK, a.n1, a.n2, a.v1, a.v2
  into       Old_Row;
```

Single row *merge*

Support for the *merge* statement was added to SQL in Oracle9*i* Database and embedded SQL automatically inherited this support[100]. Its declared purpose is to *select* rows, from a source table, for *update* or *insert* into different destination table with a compatible shape. The choice is made according to the identity of values between pairs of named columns in the source and destination tables. The functionality, for obvious reasons, is sometimes informally called "upsert". This section shows how you can use the *merge* statement in PL/SQL to upsert a row represented as PL/SQL variables.

It helps to start with a pure SQL example. Suppose that table *t1* has exactly the same column definition as table *t* and has some rows whose value of *PK* are represented in *t* and some where this is not the case. *Code_56* shows the SQL statement[101] that will *update* the rows in *t* with a matching *PK* value using the

---

99. We have not yet done a performance study. The theoretical penalty is that the approach shown in *Code_53*, and that shown in *Code_54*, touch every field when they need not. However, the alternative that avoid this disadvantage, using the the *DBMS_Sql* API, brings its own performance disadvantages.

100. Oracle9*i* Database brought the so-called common SQL parser to the PL/SQL compiler. In earlier releases, there were cases of statements in the class that embedded SQL supports that caused a PL/SQL compilation error. The workaround, no longer necessary of course, was to use dynamic SQL for those kinds of statement.

101. It helps to use the SQL*Plus command `SET SQLBLANKLINES ON` so that the readability of long statements like this can be improved by interleaving blank lines.

values that these matching rows have in *t1* and that will simply *insert* the remaining rows from *t1* where *PK* doesn't match.

```
-- Code_56
merge into  t Dest
using       t1 Source
on          (Dest.PK = Source.PK)

when matched then update set
  Dest.n1 = Source.n1,
  Dest.n2 = Source.n2,
  Dest.v1 = Source.v1,
  Dest.v2 = Source.v2

when not matched then insert values (
  Source.PK,
  Source.n1,
  Source.n2,
  Source.v1,
  Source.v2)
```

The syntax seems to be verbose. But recall that the source and destination tables need not have the same column names. There is no shorthand for the common case that *Code_56* illustrates.

Of course, the SQL shown in *Code_56* can be made into a legal PL/SQL embedded SQL statement simply by adding a trailing semicolon. But to be useful, it must correspond to a SQL statement that uses placeholders — and, famously, a placeholder is not legal in the place of an identifier. The regular SQL statement shown in *Code_57*[102] gives the clue.

```
-- Code_57
merge into  t Dest
using       (select
                 1      PK,
                51      n1,
               101      n2,
               'new v1' v1,
               'new v2' v2
             from Dual) Source
on          (Dest.PK = Source.PK)

when matched then update set
  Dest.n1 = Source.n1,
  Dest.n2 = Source.n2,
  Dest.v1 = Source.v1,
  Dest.v2 = Source.v2

when not matched then insert values (
  Source.PK,
  Source.n1,
  Source.n2,
  Source.v1,
  Source.v2)
```

102. The statement illustrates the impossibility of inventing rules that can guarantee the comfortable formatting of SQL statements. I tried to improve it using the *with* clause as I did in *Code_35* but couldn't find the syntax.

It's easy now to see[103] how to use this device in a PL/SQL unit where the source row is presented, after some processing, in the *record Result*; *Code_58* shows how.

```
-- Code_58
  Result t%rowtype;
begin
  ...
  merge into  t Dest
  using       (select
                  Result.PK PK,
                  Result.n1 n1,
                  ...,
                  Result.v1 v1,
                  ...
               from Dual d) Source
  on          (Dest.PK = Source.PK)

  when matched then update set
    Dest.n1 = Source.n1,
    ...,
    Dest.v1 = Source.v1,
    ...

  when not matched then insert values (
    Source.PK,
    Source.n1,
    ...,
    Source.v1,
    ...);
```

For comparison, *Code_59* shows how the same "upsert" requirement could be met without using the *merge* statement.

```
-- Code_59
  Result t%rowtype;
begin
  ...
  begin
    insert into t values Result;
  exception when Dup_Val_On_Index then
    update  t a
    set     row = b.Result
    where   a.PK = b.Result.PK;
  end;
```

Code_59 seems more attractive than *Code_58* because it enjoys the benefit that using a *record* brings for code maintenance in the face of table format changes. However, a performance experiment will show that the "official" *merge* approach is substantially faster. Most people are prepared to pay some maintenance cost for correct code which performs better than correct code which takes less effort to write. That leads to the following best practice principle:

Use *merge* for an "upsert" requirement.
Don't use *update... set row...* together with *insert* in an exception handler.

Principle_19

When you have an "upsert" requirement, use *merge* rather than implementing *update... set row...* and providing an exception handler for *Dup_Val_On_Index* that implements the corresponding *insert*.

103. It may be easy to see, but it isn't effortless to write the syntax! However, the semantic power (and therefore performance) that this gives is worth the effort.

Multirow operation

It is common that a PL/SQL unit computes a stream of values that might be used to execute a particular *insert*, *update*, *delete*, or *merge* statement many times in succession. Beginners might write code like *Code_60* shows.

```
-- Code_60
for ... loop
  ...
  PK := ...
  n1 := ...
  ...
  update  t a
  set     a.n1 = b.n1, ...
  where   a.PK = b.PK;
end loop;
```

Oracle8*i* Database introduced the *forall* statement to improve the efficiency of repeated *insert*, *update*, *delete*, and *merge*[104] statements. *Code_61* shows how *Code_60* is rewritten to use it.

```
-- Code_61
for ... loop
  ...
  PKs(j) := ...
  n1s(j) := ...
  ...
end loop;

forall j in 1..PKs.Count() loop
  update  t a
  set     a.n1 = b.n1s(j), ...
  where   a.PK = b.PKs(j);
```

The efficiency improvement comes because, when the number of elements in the collections that are bound is *N*, the implementation of the *forall* statement manages each of the *N* implied executions of the SQL statement in a single PL/SQL to SQL to PL/SQL context switch[105] rather than the *N* switches that *Code_60* causes. The improvement is typically by a factor of several times; any trivial experiment shows this.

The most important thing to say about the *forall* statement is simple: use it. There is quite simply no reason not to. *Code_61* is no harder to write or to understand than is *Code_60*. Nor is its expressivity any less. *Code_61*, for example, can be trivially rewritten to use the *set row* syntax with a *record* bind. And the performance benefit is huge.

---

104. The *merge* statement wasn't introduced until Oracle9*i* Database; from its introduction, it has been supported by the *forall* statement.

105. See *"Selecting many rows — unbounded result set"* on page 30.

Handling exceptions caused when executing the *forall* statement

Code_62 shows how to modify the single row approach shown in *Code_60* to let execution continue if a particular iteration causes a regrettable, but not unexpected, exception.

```
-- Code_62
for ... loop
  ...
  PK := j;
  n1 := j;
  ...
  begin
    update  t a
    set     a.n1 = b.n1, ...
    where   a.PK = b.PK;
  exception
    when Dup_Val_On_Index then
      n := n + 1;
      The_Exceptions(n).Error_Index := j;
      The_Exceptions(n).Error_Code := SqlErrm();
    when ... then
      ...
  end;
end loop;
```

The_Exceptions is an *index by pls_integer* table whose element is a *record* with one field to hold the number of each iteration that caused an exception and another field to hold the error code. An exception other than *Dup_Val_On_Index*, for example an out of space error, would by design (because, in the bigger picture, only this one is considered recoverable) bubble up for a higher layer to handle appropriately.

Code_63 shows how to modify the bulk approach shown in *Code_61* to let execution continue if a particular iteration causes an exception.

```
-- Code_63
for ... loop
  ...
  PKs(j) := ...
  n1s(j) := ...
  ...
end loop;

declare
  Bulk_Errors exception;
  pragma Exception_Init(Bulk_Errors, -24381);
begin
  forall j in 1..PKs.Count() save exceptions
    update  t a
    set     a.n1 = b.n1s(j), ...
    where   a.PK = b.PKs(j);
exception
  when Bulk_Errors then
    for j in 1..Sql%Bulk_Exceptions.Count() loop
      The_Exceptions(j).Error_Index :=
        Sql%Bulk_Exceptions(j).Error_Index;
      The_Exceptions(j).Error_Code :=
        -Sql%Bulk_Exceptions(j).Error_Code;
    end loop;
end;
```

You might not see the loop that copies from the predefined *Sql%Bulk_Exceptions* collection to the local *The_Exceptions* collection in real code. *Code_63* is written this way just to make the point that the information content of *Sql%Bulk_Exceptions* in the bulk approach is the same as that of the hand-populated *The_Exceptions* in the single row approach.

Notice a subtle semantic difference. The single row approach allowed a handler that would catch only *Dup_Val_On_Index*. The use of *save exceptions* in the bulk

approach acts like using a *when others* handler — which is actually implemented by catching *ORA-24381*. If the design says that only *Dup_Val_On_Index* is recoverable, then you must traverse the *Sql%Bulk_Exceptions* in the handler for *ORA-24381* and deliberately raise a new exception when a value of *Error_Code* is found other than the one that corresponds to *Dup_Val_On_Index*.

Digression: DML Error Logging

DML Error Logging was introduced in Oracle Database 10*g* Release 2. By using special syntax in, for example, an *insert* statement, you can request that if particular row causes an error (a *varchar2* value might be bigger that its target field can hold), then the offending row is skipped and the operation continues quietly with the next row. Furthermore, information about the skipped row will be written (in an autonomous transaction) to the table that you nominate for that purpose.

The feature was introduced with a particular scenario in mind: the use of *insert... select...* to bulk load a table from a source with known dirty data[106] — data that could either be discarded or fixed up manually after its detection and then used in a second round of loading. It brings a noticeable performance benefit with respect to the hand-coded approach that uses PL/SQL to step through the source rows in an *implicit cursor for loop*, inserting each into the target table, and dealing with exceptions as they occur. This hand-coded approach can be optimized by using *batched bulk fetch* and a *forall* statement for the *insert*. (An example of this kind of approach is shown in *Code_68* on page 59.) Even with such an optimization, the approach that skips bad data using DML Error Logging is noticeable faster, and, as a single SQL statement, much easier to program, than the PL/SQL approach.

This observation has led to some developers to think that DML Error Logging should be recommended as a generic alternative, in code which for other reasons already is written to use PL/SQL and in particular to use a *forall* statement, as an alternative to using the *save exceptions* clause and implementing a handler for *ORA-24381*. A more cautious recommendation is to consider carefully the purpose of the *forall* statement and especially what action is to be taken when an attempted *insert*, *update*, *delete* (or *merge*) is attempted.

Notice that there's a semantic difference between the two approaches that you observe, typically, more with *update* and *delete* than with *insert*. This is because a common use of the *forall* statement with *insert* is to insert a single row from a source PL/SQL collection in each iteration; but a common use with *update* and *delete* is to affect many rows with each iteration. The granularity of failure with the *forall* statement is the iteration. But the granularity of failure with DML Error Logging is the single row. Neither paradigm is universally better than the other; the appropriate choice is determined by the particular requirements. Notice, too, that DML Error Logging commits failed data into a table that must be considered to be one of an application's ordinary objects. The scheme doesn't have an intrinsic notion of the session. So in a multiuser application (in contrast to an administrator-driven bulk load), that notion would have to be introduced by a suitable custom-designed tagging mechanism. That would imply the need for custom-designed housekeeping for the contents of the

106. A canonical use case presents data from a foreign system, on the filesystem, as an external table.

errors table. A further consideration is that DML Error Logging cannot handle every kind of error. For example, it cannot handle violated deferred constraints, Out-of-space errors, or an *update* or *merge* operation that raises a unique constraint or index violation.

Referencing fields of a *record* in the *forall* statement

Oracle Database 11*g* brings the end to a restriction that some programmers have complained bitterly about. Consider *Code_64*.

```
-- Code_64
loop
  fetch Cur bulk collect into Results limit Batchsize;

  for j in 1..Results.Count() loop
    ...
  end loop;

  forall j in 1..Results.Count()
    update t a
    set    a.v1 = b.Results(j).v1
    where  Rowid = b.Results(j).Rowid;

  exit when Results.Count() < Batchsize;
end loop;
```

In earlier versions, this code fails to compile, causing the error *PLS-00436: implementation restriction: cannot reference fields of BULK In-BIND table of records*. The workaround had to be to use a separate collection of scalars for each table column. *Code_68* (see page 59) shows how *Code_64* has to be written in earlier versions of Oracle Database.

Bulk *merge*

It is tempting to apply the same reasoning to "bulkifying" the single row *merge* statement as we used to transform the single *insert*, *update*, and *delete* using the *forall* statement. *Code_65* shows the code this would produce.

```
-- Code_65
  type Results_t is table of t%rowtype index by pls_integer;
  Results Results_t;
begin
  ...
  forall j in 1..Results.Count()
    merge into  t Dest
    using       (select
                     Results(j).PK PK,
                     Results(j).n1 n1,
                     ...,
                     Results(j).v1 v1,
                     ...
                 from Dual d) Source
    on          (Dest.PK = Source.PK)

    when matched then update set
      Dest.n1 = Source.n1,
      ...,
      Dest.v1 = Source.v1,
      ...

    when not matched then insert values (
      Source.PK,
      Source.n1,
      ...,
      Source.v1,
      ...);
```

Consider, though, how the *merge* statement is described: its purpose is to *select* rows, from a source table, for *update* or *insert* into different destination table. We

can take advantage of this if we remember that the *table* operator can be used with a PL/SQLvariable whose datatype is a collection of *objects* — provided, that is, that the *object* datatype and the collection datatype are defined at schema level. *Code_66* shows the SQL*Plus script that would do this.

```
-- Code_66
create type Result_t is object(
  PK number,
  n1 number,
  ...
  v1 varchar2(30),
  ...)
/
create type Results_t is table of Result_t
/
```

This understanding lets us express the purpose of *Code_65* rather differently in *Code_67*.

```
-- Code_67
  Results Results_t;
begin
  ...
  merge into  t Dest
  using       (select * from table(Results)) Source
  on          (Dest.PK = Source.PK)

  when matched then update set
    Dest.n1 = Source.n1,
    ...,
    Dest.v1 = Source.v1,
    ...

  when not matched then insert values (
    Source.PK,
    Source.n1,
    ...,
    Source.v1,
    ...);
```

Code_65 asks for its SQL statement to be rebound and re-executed many times. But *Code_67* asks for its SQL statement to be bound and executed just once. Not surprisingly, then, it is faster. It does, though, require slightly different programming to populate the table of *objects* that it does to populate the table of *records* — but the difference is cosmetic rather than fundamental.

The discussion in this section is summarized in this best practice principle:

Use the *forall* statement rather than repeating a single-row statement. Handle ORA-24381 when it's safe to skip over a failed iteration. For bulk *merge*, use the *table* operator with a collection of *objects*.

Principle_20

Always use the *forall* statement for repetitive execution of a particular *insert*, *update*, or *delete* statement in preference to the equivalent single-row approach. Use the keyword *save exceptions* and provide a handler for *ORA-24381* when you can continue safely after a particular iteration fails. For bulk *merge*, use the *table* operator with a collection of *objects* to represent the to-be-merged rows.

Using native dynamic SQL for *insert*, *update*, *delete*, and *merge*

The transformation of working code which uses embedded SQL to use native dynamic SQL instead is relatively mechanical. You always use the *execute immediate* statement. The embedded SQL text is replaced with a string variable[107] holding the same text that has placeholders instead of the PL/SQL variables; this is used with *execute immediate*; binding is achieved with the *using* clause; and the output of a SQL statement's *returning* clause is captured using *out* binds. (You never use the *into* clause.) The only point to note is that

embedded SQL that uses the *set row* syntax has no counterpart in native dynamic SQL; rather, you have to mention each *record* field explicitly.

107. The operand of *execute immediate* is very likely to be a *varchar2*. However, Oracle Database 11*g* brought the possibility that it might be a *clob*. This is useful for the programmatic generation of PL/SQL units whose source text exceeds the 32k capacity limit for a *varchar2*. In earlier releases, exceeding this limit meant rewriting your code to use the *DBMS_Sql* API.

SOME USE CASES

This section examines some commonly occurring scenarios and discusses the best approach to implement the requirements.

Changing table data in response to query results

Various scenarios arise where table data needs to be transformed in bulk — either in place, or into a new table. Sometimes, the data in the source table needs to be distributed between more than one destination table[108]. Sometimes, the required rules cannot be expressed using only SQL expressions in *update* statements or *insert into... select ... from...* statements. *Code_68* shows how *batched bulk fetch* and the *forall* statement can be used together for this kind of processing.

```
-- Code_68
  cursor Cur is
    select Rowid, a.v1 from t a for update;

  type Rowids_t is varray(1000) of Rowid;
  Rowids Rowids_t;

  type vs_t is varray(1000) of t.v1%type;
  vs vs_t;

  Batchsize constant pls_integer := 1000;
begin
  ...
  loop
    fetch Cur bulk collect into Rowids, vs limit Batchsize;
    for j in 1..Rowids.Count() loop
      -- This is a trivial example.
      vs(j) := f(vs(j));
    end loop;
    forall j in 1..Rowids.Count()
      update  t a
      set     a.v1 = b.vs(j)
      where   Rowid = b.Rowids(j);
    exit when Rowids.Count() < Batchsize;
  end loop;
```

Of course, when the transformation is trivial and can be expressed using a PL/SQL function, then a simpler approach is possible. *Code_69* implements exactly the same effect as *Code_68*.

```
-- Code_69
...
update t set v1 = f(v1);
```

It isn't necessary, even, to use a PL/SQL subprogram to issue the *update* statement in *Code_69*, but this might be useful if the subprogram is part of the API that hides direct SQL access to the table(s). However, *Code_68* is just an illustration of the technique. There are transformations that cannot be expressed as is this one in *Code_69*.

---

108. This scenario arises particularly in connection with the upgrade of an application from one version to the next. Sometimes, the vendor needs to change the design of the application's tables in order to add functionality or to improve performance. For example, an upgrade might add a column that holds a denormalization. The upgrade script would need to populate such a column in bulk.

It is interesting to compare the performance of *Code_68* and *Code_69*: does *Code_68*, as an approach, itself bring an automatic performance penalty? It seems not too[109].

Finally, we should consider how the naïve, single row approach looks and performs. *Code_70* shows it.

```
-- Code_70
for r in (select Rowid, a.v1 from t a for update) loop
  r.v1 := f(r.v1);
  update t a set a.v1 = r.v1
  where Rowid = r.Rowid;
end loop;
```

There is no doubt that *Code_70* is more concise than *Code_68*. But, it is noticeably slower[110].

A variation on *Code_70* might use an *explicit cursor*, *Cur*, and the *where current of Cur* construct. For completeness, *Code_71* shows this.

```
-- Code_71
  cursor Cur is
    select a.v1 from t a for update;
  v1 t.v1%type;
begin
  open Cur;
  loop
    fetch Cur into v1;
    exit when Cur%NotFound;
    v1 := f(v1);
    update  t a
    set     a.v1 = v1
    where   current of Cur;
  end loop;
  close Cur;
```

Code_72 shows the SQL statements that the PL/SQL compiler generates[111] from the source code shown in *Code_71*.

```
-- Code_72
SELECT A.V1 FROM T A FOR UPDATE
UPDATE T A SET A.V1 = V1 WHERE ROWID = :B1
```

This shows that the *where current of Cur* construct is no more than syntax sugar for what *Code_70* achieves by selecting the *Rowid* explicitly. It has no counterpart in bulk constructs, but that is in no way a drawback[112].

Here, then, is the best practice principle:

Principle_21

When many rows need to be transformed using an approach that can be

Don't be afraid to get rows with *batched bulk fetch*, process them in PL/SQL, and to put each batch back with a *forall* statement. The approach carries no noticeable performance cost compared to using a PL/SQL function directly in a SQL statement.

109. You might like to try your own experiments on your own data.

110. Our tests, using a table with 2,000,000 rows, showed that *Code_68* and *Code_69* used the same CPU time — measured with *DBMS_Utility.Get_CPU_Time()* — to within the measurement accuracy of a few percent. However, *Code_70* was slower by a factor of *1.5x*.

111. *Code_72* is discovered using a query like this:

```
select   Sql_Text
from     v$Sql
where    Lower(Sql_Text) not like '%v$sql%'
and     (Lower(Sql_Text) like 'select%a.v1%from%t%' or
         Lower(Sql_Text) like 'update%t%a%set%')
```

Code_5 has been formatted by hand to make it easier to read.

expressed only in PL/SQL, retrieve them with *batched bulk fetch*, process them, and use the results of each batch with a *forall* statement — either to *update* the source rows, using the *Rowid*, or to *insert* to different table(s). If needed, the approach can be combined with *merge*. The approach itself, compared to using a PL/SQL function directly in a suitable SQL statement, brings no noticeable performance penalty.

Number of *in list* items unknown until run time

If, for some relatively improbable reasons, you need a query whose *where* clause uses an *in list* with, say, exactly five items, then you can write the embedded SQL statement without noticing the challenge that will present itself later when the requirements change to reflect the more probable use case. *Code_73* shows this unlikely statement.

```
-- Code_73
select              a.PK, a.v1
bulk collect into   b.Results
from                t a
where               a.v1 in (b.p1, b.p2, b.p3, b.p4, b.p5);
```

Code_74 expresses the intention of the far more likely statement.

```
-- Code_74
select              a.PK, a.v1
bulk collect into   b.Results
from                t a
where               a.v1 in (b.ps(1), b.ps(2), b.ps(3),
                             b.ps(4), b.ps(5), b.ps(6),
                             b.ps(7), b.ps(8), b.ps(9),
                             ...                      );
```

The problem is immediately apparent: we cannot bear to write an explicit reference to each element in a collection using a literal for the index value — and even if we could, the text would become unmanageably voluminous[113]. Rather, we need a syntax that expresses the notion "all the elements in this collection, however many that might be". Such a syntax exists and is supported in embedded SQL; *Code_75* shows it.

```
-- Code_75
  ps Strings_t;
begin
  select              a.PK, a.v1
  bulk collect into   b.Results
  from                t a
  where               a.v1 in (select  Column_Value
                               from    table(b.ps));
```

112. In general, it is risky to rely on the constancy of *Rowid*. For example, following an *alter table... shrink* command, the *Rowid* for a row with a particular primary key might change as a consequence of the row movement. However, there is no risk during the interval between a issuing a *select... for update* and the *commit* or *rollback* that ends that transaction because an attempted *alter table... shrink* from another session will wait until the present one ends its transaction.

113. You might hard code 1,000 such explicitly indexed elements and then at run time, set as many elements as you need to the intended values and the remainder to *null*. This has the correct semantics because a test for equality with *null* has the same effect as an equality test that fails.

However, this seems to be relatively little known, possibly because it uses the *table* operator[114]. The datatype of *ps* must be declared at schema level. *Code_76* shows the SQL*Plus script that creates it.

```
-- Code_76
create type Strings_t is table of varchar2(30)
/
```

It is not uncommon for programmers who don't know about the *table* operator to satisfy the functionality requirement by building the text of the SQL statement at run time. Either they use literal values and execute the statement with native dynamic SQL, or they use placeholders and then are forced to execute the statement with the *DBMS_Sql* API to accommodate the fact the the binding requirement isn't known until run time. Both these workarounds are shocking examples of worst practice[115]. (We have heard of other programmers who have first inserted the *in list* values into a global temporary table so that the intended query can be expressed as a join to that.) The best practice principle, then, is obvious:

Principle_22

When you need the functionality of an *in list* whose element count is not known until run time, populate a collection, whose datatype must be defined at schema level, with the values and then use *"where x in (select Column_Value from table(The_Values))"*. Don't even consider alternative approaches.

Use *"where x in (select Column_Value from table(The_Values))"* for the functionality of an *in list* whose element count you don't know until

114. The use of the *table* operator is explained in the section *Manipulating Individual Collection Elements with SQL* in the Object-Relational Developer's Guide. The PL/SQL Language Reference book mentions it, with no explanation, only in the *PL/SQL Language Elements* section.

115. The native dynamic SQL approach leads to a proliferation of distinct statement texts and therefor causes excessive *hard parses*. (In naïve hands, it also exposes a risk for SQL injection.) Using the the *DBMS_Sql* API reduces the number of *hard parses*, but *N* hard parses when one is sufficient is still *N-1* too many.

CONCLUSION

The first section of the paper, *Embedded SQL, native dynamic SQL and the DBMS_Sql API* on page 5, presented a straightforward review of the vast apparatus for doing SQL from PL/SQL and it attempted to clarify, by introducing some new terminology, notions that users often find to be rather confusing[116].

The remaining expository sections, *Approaches for select statements* on page 30 and *Approaches for insert, update, delete, and merge statements* on page 47, deliberately took a different stance. Their aim was to identify and recommend a subset of the available functionality that is small enough to hold in a coherent mental model and yet rich enough for most practical purposes. This is bound to be something of a compromise. One who understood absolutely everything about how to do SQL from PL/SQL (and, with that, understood the history of introduction of features), and who had a huge practical experience of having used every technique to advantage in a real-world application, is best placed to choose the perfect technique for a particular challenge in a new project. But such people are not the intended readers of this paper!

Rather, this paper is intended for readers who need to write acceptably performant code with a reasonable uniformity of approach so that others may easily understand and maintain it. Of course, correctness is a non-negotiable goal. We argue that the chances of correctness are hugely increased, over a set of programs with largely similar intent, when the variety of programming techniques used is relatively small, and when these techniques are matched to requirements in a uniform way.

We hope that the paper has achieved its goal of helping such readers.

Bryn Llewellyn,
PL/SQL Product Manager, Oracle Headquarters
bryn.llewellyn@oracle.com
21-September-2008

116. The reason for this confusion is not least because of terminology which Oracle Corporation has introduced organically and somewhat myopically over the many years of PL/SQL's history.

APPENDIX A: CHANGE HISTORY

19-September-2008

- First published version.

21-September-2008

- Rewording the best practice principle about the choice of *authid*. Correcting minor errors in response to feedback from colleagues.

APPENDIX B: SUMMARY OF BEST PRACTICE PRINCIPLES

This appendix collects the best practice principles that have been advocated in this paper. An unattributed programmers' axiom has it that rules exists to serve, not to enslave[117]. A much more sensible suggestion is listed here as the first (meta) principle.

Seek approval from an experienced colleague before disobeying any of the following best practice principles

- *Principle_0:* Do not deviate from any of the following principles without first discussing, with a more experienced PL/SQL programmer, the use case that seems to warrant such deviation.

In embedded SQL, dot-qualify each column name with the from list item alias. Dot-qualify each PL/SQL identifier with the name of the name of the block that declares it.

- *Principle_1:* When writing an embedded SQL statement, always establish an alias for each from list item and always qualify each column with the appropriate alias. Always qualify the name of every identifier that you intend the be resolved in the current PL/SQL unit with the name of the block in which it is declared. If this is a block statement, give it a name using a label. The names of the aliases and the PL/SQL bocks must all be different. This inoculates against name capture when the referenced tables are changed and, as a consequence, increases the likelihood that the fine-grained dependency analysis will conclude that the PL/SQL unit need not be invalidated. (page 9)

Declare every PL/SQL variable with the constant keyword unless the block intends to change it.

- *Principle_2:* Use the constant keyword in the declaration of any variable that is not changed after its initialization. Following this principle has no penalty because the worst that can happen is that code that attempts to change a constant will fail to compile — and this error will sharpen the programmer's thinking. The principle has a clear advantage for readability and correctness. (page 11)

Always specify the authid property explicitly. Decide carefully between Current_User and Definer.

- *Principle_3:* Always specify the authid property explicitly in every PL/SQL unit; choose between definer's rights and invoker's rights after a careful analysis of the purpose of the unit. (page 12)

117. A variation on this theme adds paradox by saying *"All rules were meant to be broken — including this one"*.

Use the Owner to dot-qualify the names of objects that ship with Oracle Database.

✔ *Principle_4:* References to objects that Oracle Corporation ships with Oracle Database should be dot-qualified with the Owner. (This is frequently, but not always, Sys.) This preserves the intended meaning even if a local object, whose name collides with that of the intended object, is created in the schema which will be current when name resolution is done. (page 13)

Strive to use SQL statements whose text is fixed at compile time. When you cannot, use a fixed template. Bind to placeholders. Use DBMS_Assert to make concatenated SQL identifiers safe.

✔ *Principle_5:* Strive always to use only SQL statements whose text is fixed at compile time. For select, insert, update, delete, merge, or lock table statements, use embedded SQL. For other kinds of statement, use native dynamic SQL. When the SQL statement text cannot be fixed at compile time, strive to use a fixed syntax template and limit the run-time variation to the provision of names. (This implies using placeholders and making the small effort to program the binding.) For the names of schema objects and within-object identifiers like column names, use Sys.DBMS_Assert.Simple_Sql_Name(). If exceptional requirements mandate the use of a literal value rather than a placeholder, use Sys.DBMS_Assert.Enquote_Literal(). For other values (like, for example, the value for NLS_Date_Format in Code_8) construct it programatically in response to parameterized user input. (page 15)

For dynamic SQL, aim to use native dynamic SQL. Only when you cannot, use the DBMS_Sql API.

✔ *Principle_6:* For dynamic SQL, always use native dynamic SQL except when its functionality is insufficient; only then, use the DBMS_Sql API. For select, insert, update, delete, and merge statements, native dynamic SQL is insufficient when the SQL statement has placeholders or select list items that are not known at compile time. For other kinds of SQL statement, native dynamic SQL is insufficient when the operation is to be done in a remote database. (page 17)

When using dynamic SQL, avoid literals in the SQL statement. Instead, bind the intended values to placeholders.

✔ *Principle_7:* Avoid using concatenated literals in a dynamically created SQL statement; rather, use placeholders in place of literals, and then at run time bind the values that would have been literals. This maximizes the reuse of sharable SQL structures. (page 19)

Always open a DBMS_Sql numeric cursor with DBMS_Sql.Parse(Security_Level=>2);

✔ *Principle_8:* Always use the overload of the DBMS_Sql.Parse() that has the formal parameter Security_Level and always call it with the actual value 2 to insist that all operations on the DBMS_Sql numeric cursor are done with the same Current_User and enabled roles. (page 25)

The only explicit cursor attribute you need to use is Cur%IsOpen. The only implicit cursor attributes you need are Sql%RowCount, Sql%Bulk_RowCount, and Sql%Bulk_Exceptions.

- *Principle_9:* When the approaches that this paper recommends are followed, the only useful explicit cursor attribute is Cur%IsOpen. There is never a need to use the other explicit cursor attributes. The only scalar implicit cursor attribute of interest is Sql%RowCount. Always observe this in the PL/SQL statement that immediately follows the statement that executes the SQL statement of interest using an implicit cursor. The same rationale holds for the Sql%Bulk_RowCount collection. Sql%Bulk_Exceptions must be used only in the exception handler for the Bulk_Errors exception; place this in a block statement that has the forall statement as the only statement in its executable section. (page 27)

Learn the terms of art: session cursor, implicit cursor, explicit cursor, cursor variable, and DBMS_Sql numeric cursor. Use them carefully and don't abbreviate them.

- *Principle_10:* When discussing a PL/SQL program, and this includes discussing it with oneself, commenting it, and writing its external documentation, aim to avoid the unqualified use of "cursor". Rather, use the appropriate term of art: session cursor, implicit cursor, explicit cursor, cursor variable, or DBMS_Sql numeric cursor. The discipline will improve the quality of your thought and will probably, therefore, improve the quality of your programs. (page 29)

When you don't know how many rows your query might get, use fetch... bulk collect into with the limit clause inside an infinite loop.

- *Principle_11:* When many rows are to be selected, and the result set may be arbitrarily big, process them in batches by using fetch... bulk collect into with the limit clause inside an infinite loop. Use a constant as the value for the limit clause to define the batchsize; 1000 is a reasonable value. Fetch into a varray declared with the same size. Don't test the %NotFound cursor attribute to terminate the loop. Instead, use exit when Results.Count() < Batchsize; as the last statement in the loop; ensure correct processing in the edge case that the last fetch gets exactly zero rows. When embedded SQL is sufficient, use an explicit cursor. When you need native dynamic SQL, use a cursor variable. (page 33)

When you do know how the maximum number of rows your query might get, use select... bulk collect into or execute immediate... bulk collect into to fetch all the rows in a single step.

- *Principle_12:* When many rows are to be selected, and the result set can be safely assumed to be of manageable maximum size, fetch all the rows in a single step. Use the cursor-less PL/SQL constructs select... bulk collect into when embedded SQL is possible, and execute immediate... bulk collect into when dynamic SQL is needed. Fetch into a varray declared with the maximum size that you are prepared to handle. Implement an exception handler for ORA-22165 to help bug diagnosis. (page 35)

Use the the DBMS_Sql API when you don't know the binding requirement of what the select list is until run time. If you do, at least, know the select list, use To_Refcursor() and then batched bulk fetch.

- *Principle_13:* Avoid temptation to compose the where clause using literals — and especially to concatenate a where clause which has been explicitly typed by the user. The performance is likely to be noticeably worse than an approach that binds to placeholders; and, with a directly typed where clause, it isn't feasible to use Sys.DBMS_Assert.Enquote_Literal() to inoculate against SQL injection. When the binding requirement is not known until run time, use the DBMS_Sql API to parse, bind, and execute the SQL statement. If the select list is known at compile time, use To_Refcursor() to transform the DBMS_Sql numeric cursor to a cursor variable and then use batched bulk fetch. If the select list is not known until run time, use the DBMS_Sql API to fetch the results too. In the unlikely case that the binding requirement is known at compile time but the select list is not known until run time, use native dynamic SQL to open a cursor variable and then use To_Cursor_Number() to transform the cursor variable to a DBMS_Sql numeric cursor. Then use the DBMS_Sql API to fetch the results. (page 39)

To get exactly one row, use select... into or execute immediate... into. Take advantage of No_Data_Found and Too_Many_Rows.

- *Principle_14:* When exactly one row is to be selected, fetch the row in a single step. Use the cursor-less PL/SQL constructs select... into when embedded SQL is possible, and execute immediate... into when dynamic SQL is needed. Take advantage of the regrettable No_Data_Found exception and the unexpected Too_Many_Rows exception. (page 41)

Expose your database application through a dedicated schema that has only private synonyms for the objects that define its API.

- *Principle_15:* Create a dedicated schema, say Some_App_API, for an application to which database clients will connect in order to access that application's functionality. Implement all the application's database objects in schemas other than Some_App_API, with closely guarded passwords, and limit the objects in Some_App_API to just private synonyms. Create these synonyms for exactly and only those of the application's database objects that are intended to expose its client API. Grant to Some_App_API only those privileges that are needed to allow the intended access to these objects. (page 41)

Expose your database application through a PL/SQL API. Hide all the tables in schemas that the client to the database cannot reach.

- *Principle_16:* Restrict the object types that Some_App_API's private synonyms exposes to just PL/SQL units. Hide all the tables in other schemas, and do not grant any privileges on these to Some_App_API. (page 42)

Define the producer/consumer API as a function whose return datatype represents the desired data. Hide all the SQL processing in the producer module. That way, the consumer is immune to an implementation change that a requirements change might cause. Simply parameterize the producer function as you would parameterize the query. This approach accommodates getting the rows in batches or getting all the rows in one call — where this might be a slice.

- *Principle_17:* Define the producer/consumer API as a function whose return datatype represents the data that is produced. Hide everything to do with the SQL processing, including the fetching, in the producer module. When the query parameterization specifies exactly one row, use a record or ADT with the same shape as the select list. In this case, use one of the cursor-less PL/SQL constructs select... into or execute immediate... into depending on the requirement for dynamic SQL. When the query parameterization specifies many rows, use a collection of records or a collection of ADTs. In this case, use entire bulk fetch when this is certain to be safe; this allows the use of one of the cursor-less PL/SQL constructs select... bulk collect into or execute immediate... bulk collect into. If entire bulk fetch is unsafe, use batched bulk fetch when the producer/consumer relationship is stateful. This requires a identified cursor. If embedded SQL if sufficient, then use an explicit cursor declare at global level in the body of the producer package. This will hold state across the calls from the consumer to get each batch. If dynamic SQL is needed, use a cursor variable. Pass this back to the consumer with each batch of results so that the consumer can hold on to it. When the producer/consumer relationship is stateless, use result set slicing. Implement the delivery of each slice with entire bulk fetch. If requirements dictate it, use the DBMS_Sql API and hide all of the code that uses this in the producer module. (page 45)

For each application table, maintain a template record type that defines the same constraints and defaults.

- *Principle_18:* Maintain a package that exposes a record type declaration for each of the application's tables. The declaration must repeat the specification of column name, datatype, constraint, and default value that characterizes the table. (page 48)

Use merge for an "upsert" requirement. Don't use update... set row... together with insert in an exception handler.

- *Principle_19:* When you have an "upsert" requirement, use merge rather than implementing update... set row... and providing an exception handler for Dup_Val_On_Index that implements the corresponding insert. (page 52)

Use the forall statement rather than repeating a single-row statement. Handle ORA-24381 when it's safe to skip over a failed iteration. For bulk merge, use the table operator with a collection of objects.

- *Principle_20:* Always use the forall statement for repetitive execution of a particular insert, update, or delete statement in preference to the equivalent single-row approach. Use the keyword save exceptions and provide a handler for ORA-24381 when you can continue safely after a particular iteration fails. For bulk merge, use the table operator with a collection of objects to represent the to-be-merged rows. (page 57)

Don't be afraid to get rows with batched bulk fetch, process them in PL/SQL, and to put each batch back with a forall statement. The approach carries no noticeable performance cost compared to using a PL/SQL function directly in a SQL statement.

- *Principle_21:* When many rows need to be transformed using an approach that can be expressed only in PL/SQL, retrieve them with batched bulk fetch, process them, and use the results of each batch with a forall statement — either to update the source rows, using the Rowid, or to insert to different table(s). If needed, the approach can be combined with merge. The approach itself, compared to using a PL/SQL function directly in a suitable SQL statement, brings no noticeable performance penalty. (page 60)

Use "where x in (select Column_Value from table(The_Values))" for the functionality of an in list whose element count you don't know until

- *Principle_22:* When you need the functionality of an in list whose element count is not known until run time, populate a collection, whose datatype must be defined at schema level, with the values and then use "where x in (select Column_Value from table(The_Values))". Don't even consider alternative approaches. (page 62)

APPENDIX C: ALTERNATIVE APPROACHES TO POPULATING A COLLECTION OF *RECORDS* WITH THE RESULT OF A *SELECT* STATEMENT

Code_77 shows the most obvious approach: the results are bulk fetched into a collection of *records* and then the values are copied in an explicit loop into a collection of *ADTs* with the same shape.

```
-- Code_77
select PK, n1, n2, v1, v2
bulk collect into Records
from t
order by PK;

Objects.Extend(Records.Count());
for j in 1..Records.Count() loop
  Objects(j) := Object_t(
    Records(j).PK,
    Records(j).n1,
    Records(j).n2,
    Records(j).v1,
    Records(j).v2);
end loop;
```

Code_78 shows a more compact approach that dispenses with the need for the interim collection of *records* by constructing the required *ADT* as a *select list* element which can then be bulk fetched directly into the collection of *ADTs*. Surprisingly, though, this approach is noticeably slower than that shown in *Code_77*.

```
-- Code_78
select Object_t(PK, n1, n2, v1, v2)
bulk collect into Objects
from t
order by PK;
```

Code_79 shows a more compact approach that constructs the required collection of *ADTs* directly in SQL as a *select list* element in a single row which can then be fetched directly. This approach is about the same speed as that shown in *Code_77*.

```
-- Code_79
select cast(multiset(
    select Object_t(PK, n1, n2, v1, v2)
    from t
    order by PK)
  as Objects_t)
  into Objects
  from Dual;
```

The quickest two approaches are, in turn, noticeably slower than simply fetching into a collection of *records* and then processing the results in that representation. However, this, through Oracle Database 11*g*, is the price that must be paid in order to expose the database through just a PL/SQL API so that an arbitrary client can use it.

APPENDIX D: CREATING THE TEST USER *USR*, AND THE TEST TABLE *USR.T(PK NUMBER, V1 VARCHAR2(30), ...)*

Code_80 shows a convenient SQL*Plus script to start any *ad hoc* experiment in your own test database.

```
-- Code_80
CONNECT Sys/p@111 AS SYSDBA
declare
  User_Does_Not_Exist exception;
  pragma Exception_Init(User_Does_Not_Exist, -01918);
begin
  begin
    execute immediate 'drop user Usr cascade';
  exception when User_Does_Not_Exist then null; end;
  execute immediate '
    grant Create Session, Resource to Usr identified by p';
end;
/
alter session set Current_Schema = Usr
/
```

Notice that by remaining connected as *Sys*, *ad hoc* queries against views like *DBA_Objects*, *v$Sql*, and *v$Parameter* and commands like *alter system* are possible without any fuss. Of course, you should do this only in a disposable database[118]. Do be careful when using this device, though. Depending on the purpose of the experiment that the SQL*Plus script will implement, the convenience of being able to access all objects may confuse the observations. This is most conspicuously the case in an experiment whose purpose is to investigate, or illustrate, the behavior of invoker's rights PL/SQL units.

118. I make a cold backup of any freshly created database and write a script to shutdown the database, restore this, and startup the database. It takes only a couple of minutes to run on my developers' size Intel/Linux machine. The ability quickly and reliably to restore a clean test environment is hugely liberating when contemplating potentially dangerous experiments to test or confirm one's understanding of how things work.

Code_81 shows a procedure that creates a convenient test table, *t*, with a parameterized number of rows. It's very useful for performance testing to be able to test the code quickly on a small dataset and then time it on a big set.

```
-- Code_81
procedure Usr.Create_Table_T(No_Of_Batches in pls_integer)
  authid Current_User
is
  Batchsize constant pls_integer := 1000;
  No_Of_Rows constant pls_integer := Batchsize*No_Of_Batches;
  n integer := 0;

  type PKs_t is table of number       index by pls_integer; PKs Pks_t;
  type n1s_t is table of number       index by pls_integer; n1s n1s_t;
  type n2s_t is table of number       index by pls_integer; n2s n2s_t;
  type v1s_t is table of varchar2(30) index by pls_integer; v1s v1s_t;
  type v2s_t is table of varchar2(30) index by pls_integer; v2s v2s_t;
begin
  declare
    Table_Does_Not_Exist exception;
    pragma Exception_Init(Table_Does_Not_Exist, -00942);
  begin
    execute immediate 'drop table Usr.t';
  exception when Table_Does_Not_Exist then null; end;

  execute immediate '
    create table Usr.t(
      PK number                      not null,
      n1 number       default 11     not null,
      n2 number       default 12     not null,
      v1 varchar2(30) default ''v1'' not null,
      v2 varchar2(30) default ''v2'' not null)';

  execute immediate '
    create or replace package Usr.Tmplt is
      type T_Rowtype is record(
        PK number        not null := 0,
        n1 number        not null := 11,
        n2 number        not null := 12,
        v1 varchar2(30)  not null := ''v1'',
        v2 varchar2(30)  not null := ''v2'');
    end Tmplt;';

  for j in 1..No_Of_Batches loop
    for j in 1..Batchsize loop
      n := n + 1;
      PKs(j) := n;
      n1s(j) := n*n;
      n2s(j) := n1s(j)*n;
      v1s(j) := n1s(j);
      v2s(j) := n2s(j);
    end loop;
    forall j in 1..Batchsize
      execute immediate '
        insert into  Usr.t(PK,     n1,     n2,     v1,     v2)
        values            (:PK,    :n1,    :n2,    :v1,    :v2)'
        using              PKs(j), n1s(j), n2s(j), v1s(j), v2s(j);
  end loop;

  execute immediate
    'alter table Usr.t add constraint t_PK primary key(PK)';
end Create_Table_T;
```

The procedure also creates a package that exposes the *record* type *T_Rowtype* with the same shape as table *t* and carrying the same constraints and default values. The is useful for, for example, inserting a new row into *t* using embedded SQL when only some of the values are specified. (A variable declared using *t%rowtype* doesn't pick up constraints and default values from the table.) This is used to advantage in the code shown in *"Single row insert"* on page 47.

Some might argue the procedure *Create_Table_T()* would be improved by declaring the repeated strings like *"PK number"* and the default values in *varchar2 constants*. It is left as is in this paper to make it easier to read. Other approaches are possible. For example, a PL/SQL subprogram could create the *record* template package for each of a list of tables, by accessing columns like *Column_Name*, *Data_Type*, and *Data_Default* from the *All_Tab_Cols* catalog view. Keeping everything in step would be the concern of the installation and patch/upgrade scripts for the application.

The procedure has to use dynamic SQL for the *insert* statement, even though the statement text is fixed at compile time, because the target table for the *insert* does not exist at compile time.

Doing SQL from PL/SQL: Best and Worst Practices
September 2008
Bryn Llewellyn, PL/SQL Product Manager, Oracle Headquarters

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