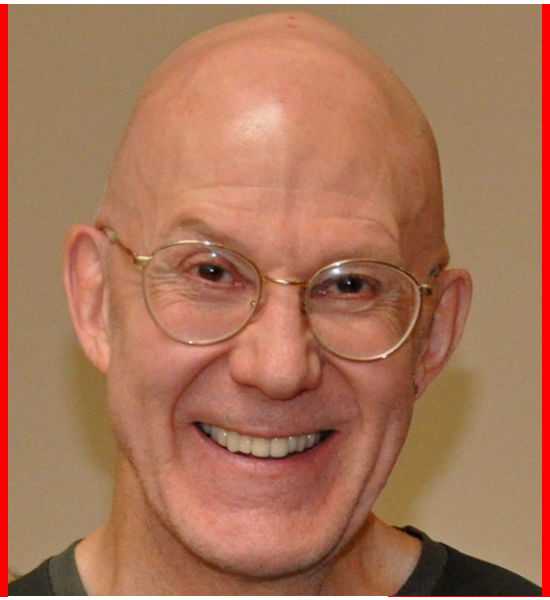

ORACLE®

Doing SQL from PL/SQL:

Best and Worst Practices

Bryn Llewellyn
Distinguished Product Manager
Database Server Technologies Division, Oracle HQ

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 remain at the sole discretion of Oracle.

Doing SQL from PL/SQL:
Best and Worst Practices

Caveat…

Doing SQL from PL/SQL:
Best and Worst Practices

- An unattributed programmers' axiom has it that rules exists to serve, not to enslave.

  "All rules were meant to be broken — including this one."

- Here's a more sensible suggestion, given as the first (meta) best practice principle.

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

Agenda

- State 22 principles in turn
- For each, explain some background
- Take the chance to explain aspects of PL/SQL that not everyone finds obvious
- Motivate studying the whitepaper

we’re off…

#1

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.

Why ?

- 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.

- The discipline will improve the quality of your thought

  …and will probably, therefore,improve the quality of your programs.

Embedded SQL

- Allows SQL syntax directly within a PL/SQL statement
- Therefore very easy to use
- Supports only the following kinds of SQL statement:
  - select
  - insert, update, delete, merge [“DML”]
  - lock table
  - commit, rollback, savepoint
  - set transaction

How does embedded SQL work ?

- You say this at compile time:

```
insert into t(PK, v1) values(j, b.v1);
...

select    a.*
into      b.The_Result
from      t a
where     a.PK = b.Some_Value;
...

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

How does embedded SQL work ?

- Your program says this at run time:

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

SELECT A.* FROM T A WHERE A.PK = :B1

...



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

In embedded SQL, dot-qualify each column name with the table alias.

Dot-qualify each PL/SQL identifier with the name of the name of the block that declares it.

```
<<b>>declare
 Some_Value t.PK%type := ...
begin

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

end;
```

Why ?

- To avoid the risk of the addition of a column to the table capturing a PL/SQL identifier
- Maximize the benefits of fine-grained dependency tracking (the compiler can’t trust your naming convention)

#3

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

Why ?

- Self-evident correctness w.r.t. injection risk
- Generic readability
- Occasional performance benefit
- ER asks for new warning

#4

Always specify the *authid* property explicitly.

Choose deliberately between *Current_User* and *Definer.*

Neither is “best”.

Why ?

- DR: protect access to tables via PL/SQL API
- IR: Avoid the risk of privilege escalation
- You will be warned

#5

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

Why ?

- Avoid the risk of subversion by a local object

#6

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.

Why ?

- Avoid the risk of injection
- Avoid avoidable hard parse

For dynamic SQL, aim to use native dynamic SQL.

Only when you cannot, use the *DBMS_Sql* API.

Why ?

- native dynamic SQL is easier to write
- And it’s faster

#8

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

Why ?

- You can’t say this one too often!

Always open a *DBMS_Sql numeric cursor* with *Security_Level* => *2*

```
Cur := DBMS_Sql.Open_Cursor(
          Security_Level=>2)
```

Why ?

- (New in 11.1.)
- Inoculate against "cursor snarfing“ (David Litchfield)
- ER asks for new warning

#10

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*.

Why ?

- *Cur%IsOpen* – to close ‘em in a catch all handler
- *Sql%RowCount* – how many rows did my DML affect?
- *Sql%Bulk_RowCount* – same for *forall*
- Observe at the earliest opportunity
- *Sql%Bulk_Exceptions* – in the *ORA-24381* handler

  (don’t forget the *save exceptions* keyword)

selecting

#11

This one is contentious!

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.

Why ?

- Bulk constructs are faster!
- But the target collections mustn’t get arbitrarily big
- ER asks for new warning

(requires the use of an *identified cursor*)

#12

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.

Why ?

- One step is better than many
- select… into is functionally complete
- Fetch into a varray declared with the maximum size that you are prepared to handle.
- Implement an exception handler for ORA-22165 (index outside the range of existing elements) to help bug diagnosis.

#13

Use the the *DBMS_Sql* API when you don't know the binding requirement or 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..

Why ?

- You have no choice.

  Method 4 is what it is –

  and the DBMS_Sql API is here to stay for that, and only that, use case

- But there's no virtue in masochism

  so twizzle to a *ref cursor* when you can

Well, you do have a choice…

- This:

```
select Count(*)
from All_Objects
where 1 = 1
and Object_Name = :b1
and (1=1 or :b2 is null)
```

- is simplified by SQL compilation to this:

```
select Count(*)
from All_Objects
where Object_Name = :b1
```

To get exactly one row, use *select... into* or *execute immediate... into*.

Take advantage of *No_Data_Found* and *Too_Many_Rows*.

Why ?

- The construct says what you mean
- The exceptions are what you need for the regrettable and the unexpected cases
- It’s fewer steps and so it’s quicker

religion

#15

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

Why ?

- How else can you enforce an API ?

Expose your database application through a PL/SQL API.

Hide all the tables in schemas that the client to the database cannot reach.

OK, this one is contentious…

- 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
  - Hide the module's implementation behind that API
- PL/SQL subprograms define an API
- Tables and SQL statements are part of a module's implementation – to be hidden from clients

Define the producer/consumer API as a function whose return datatype represents the desired data.

Hide all the SQL processing in the producer module.

Parameterize the producer function as you would parameterize the query.

Why ?

- This way, the consumer is immune to an implementation change that a requirements change might cause.
- Approach accommodates getting the rows in batches or getting all the rows in one call — where this might be a slice.

insert, update, delete -ing

#18

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

Why ?

- Lets you implement the insert of a new row where the caller mentions only some of many optional columns

```
  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;
```

#19

Use *merge* for an “upsert” requirement.

Don’t use *update... set row...* together with insert in an exception handler.

Why ?

- *merge* says what you mean in a single statement
- So it's quicker

```
 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,
    ...);
```

#20

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.

Why ?

- It’s quicker

```
 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,
    ...);
```

#21

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.

Why ?

- You have a data transformation that you need to solve
- Beyond a certain level of complexity, procedural code is easier to write and understand than declarative code
- Therefore the chances of its being correct are increased

```
 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 = Rowids(j);
    exit when Rowids.Count() < Batchsize;
  end loop;
```

11.1 lifts a notorious restriction

- **PLS-00436:**

  implementation restriction:

  cannot reference fields of
  BULK In-BIND table of records

```
loop
  fetch Cur bulk collect into Results
    limit Batchsize;
  for j in 1..Results.Count() loop
    -- This is a trivial example.
    Results(j).v1 := f(Results(j).v1);
  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;
```

Note…

- There are no special considerations for doing DML with native dynamic SQL except:
  - You can’t use the “whole row” syntax

a straggler…

Use this:

```
select ...
from   ...
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 run time.

Why ?

- Avoid native dynamic SQL with concatenated literals
- Avoid method 4 *DBMS_Sql*

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



 ...
 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));
```

finally…

search.oracle.com

- Read the detailed technical whitepaper
- It's listed on the PL/SQL Homepage

```
www.oracle.com/technetwork/database/features/plsql/
```

- Or…

Search for: **Doing SQL from PL/SQL**

In the section: **Technology Network**

Q&A