Effective PL/SQL

Thomas Kyte
http://asktom.oracle.com/

Agenda

- Why PL/SQL?
- Write as little as you can
- Use Packages
- Use Static SQL
- Bulk processing
- Implicit or Explicit?
- Beware of some features
- Things to definitely do

Why PL/SQL

Why Use PL/SQL

- It is a ‘real’ language
  - It is not a scripting language
  - It is not a ‘toy’, it is used to code ‘real’ things
- It is the API to the database

It is the most efficient language for data manipulation

- If your *goal* is to procedurally process data (after ensuring a single SQL statement cannot do your work!) then *PL/SQL* is simply the most productive language to do so

It is the most efficient language for data manipulation

```
Create or replace procedure
process_data( p_inputs in varchar2 )
As
Begin
   For x in ( select * from emp
                 where ename like p_inputs )
   Loop
      Process( X );
   End loop
End;
```

- SQL datatypes are PL/SQL datatypes
- Tight coupling between the two languages
- Code short cuts (implicit cursors)
- Protected from many database changes

It is the most efficient language for data manipulation

```
static PreparedStatement
pstmt = null;

public static void
process_data
( Connection conn, String inputs )
throws Exception
{
int    empno;
String ename;
String job;
int    mgr;
String hiredate;
int    sal;
int    comm;
int    deptno;

if ( pstmt == null )
  pstmt = conn.prepareStatement
   ("select * from emp " +
    "where ename like ? " );

pstmt.setString( 1, inputs );
ResultSet rset =
   pstmt.executeQuery();
……
```

```
while( rset.next() )
{
  empno    = rset.getInt(1);
  ename    = rset.getString(2);
  job      = rset.getString(3);
  mgr      = rset.getInt(4);
  hiredate = rset.getString(5);
  sal      = rset.getInt(6);
  comm     = rset.getInt(7);
  deptno   = rset.getInt(8);
  process( empno, ename, job, mgr,
           hiredate, sal, comm, deptno );
}
rset.close();
Pstmt.close();
}
```

- SQL datatypes are not Java types (consider number(38) issues…)
- No coupling between the two languages, entirely procedural (what about SQLJ?)
- No code short cuts (statement caching)
- Not protected from many database changes (and no dependencies either!)

PL/SQL epitomizes portability and reusability

- It is the most advanced portable language I’ve ever seen
  - It is callable from every other language out there
  - Anything that can connect to the database can use and reuse it
- Sure – there are things like SOA and Services that let X call Y
  - But these introduce their own level complexities
  - And if your service is a database server, it would be best to be written *in the database*
- If you can connect to the database – you can use and reuse PL/SQL from anything

Many mistakes made in other languages using the database are avoided

- Bind Variables
  - If you use static SQL in PL/SQL *it is impossible to not use bind variables correctly.*
  - You have to use cumbersome dynamic SQL to do it wrong.
- Parse Once, Execute many
  - PL/SQL does statement caching
  - You have to either configure and enable its use in other languages or
  - Do it your self (refer back to java code)
- Schema Changes are safer
  - `Alter table t modify c1 varchar2(255);`
  - We can find all uses of T (*I didn't know you were using that, sorry..)*
  - We can make the change without having to change code

However

- As with *any* language you can
  - Write really good code
  - Write really average code
  - Write really really really bad code
- You can make the same mistakes with PL/SQL that you can with every other language
  - By not understanding the language
  - By not understanding some implementation details
  - By not understanding SQL
  - By not designing
  - And so on…

Write as Little as you can

Code…

- Write as much code:
  - As you have to
  - But as little as you can…
- Think in SETS
- Use (really *use* – not just ‘use’) SQL

```
Begin
  For x in ( select * from table@remote_db )
  Loop
       Insert into table ( c1, c2, … )
       values ( x.c1, x.c2,… );
  End loop;
End;
```

```
Insert into table (c1,c2,…)
select c1,c2,… From table@remote_db
LOG ERRORS ( some_variable )
REJECT LIMIT UNLIMITED;
… code to handle errors
for tag some_variable …
```

Use PL/SQL constructs only when SQL cannot do it

- Another coding *‘technique’* I see frequently:

```
For a in ( select * from t1 )
Loop
   For b in ( select * from t2
              where t2.key = a.key )
   Loop
      For c in ( select * from t3
                 where t3.key = b.key )
      Loop
         …
```

- The developer did not want to “burden” the database with a join

More Code = More Bugs
Less Code = Less Bugs

- This code speaks for itself.
- So does this.

- Always look at the procedural code and ask yourself "is there a set based way to do this *algorithm*"
  - For example …

More Code = More Bugs
Less Code = Less Bugs

```
insert into t ( .... )
 select EMPNO, STATUS_DATE, ....
   from t1, t2, t3, t4, ....
  where ....;

 loop
    delete from t
     where (EMPNO,STATUS_DATE)
        in ( select EMPNO,
                    min(STATUS_DATE)
               from t
              group by EMPNO
             having count(1) > 1 );
    exit when sql%rowcount = 0;
 end loop;
```

For any set of records with more than one EMPNO, remove rows with the oldest STATUS_DATE.

Additionally – If the last set of EMPNO records all have the same STATUS_DATE, remove them all.

```
EMPNO      STATUS_DATE
--------   ------------
1          01-jan-2009
1          15-jun-2009
1          01-sep-2009

…
1000       01-feb-2009
1000       22-aug-2009
1000       10-oct-2009
1000       10-oct-2009
```

```
(1,01-jan-2009,3)   (1001,01-feb-2009,4)
(1,15-jan-2009,2)   (1001,22-aug-2009,3)
```

More Code = More Bugs
Less Code = Less Bugs

```
insert /* APPEND */ into t ( .... )
select EMPNO, STATUS_DATE, ......
   from ( select EMPNO, STATUS_DATE, .... ,
                 max(STATUS_DATE)
                       OVER ( partition by EMPNO ) max_sd,
                 count(EMPNO)
                  OVER ( partition by EMPNO,STATUS_DATE ) cnt
   from t1, t2, t3, t4, …
  where … )
  where STATUS_DATE = max_sd
   and cnt = 1;
```

- This was a data warehouse load (load 2-3-4 times the data you want, then delete? Ouch)
- It was wrong – procedural code is no easier to understand than set based code, documentation is key

More Code = More Bugs
Less Code = Less Bugs

```
/* Load table t using history tables.  History tables have
   multiple records per employee.  We need to keep the
   history records for each employee that have the maximum
   status date for that employee.  We do that by computing
   the max(status_date) for each employee (partition by EMPNO
   finding max(status_date) and keeping only the records such
   that the status_date for that record = max(status_date)
   for all records with same empno */

insert /* APPEND */ into t ( .... )
 select EMPNO, STATUS_DATE, ......
    from ( select EMPNO, STATUS_DATE, .... ,
                  max(STATUS_DATE)
                        OVER ( partition by EMPNO ) max_sd
             from t1, t2, t3, t4, …
            where … )
   where STATUS_DATE = max_sd;
```

More Code = More Bugs
Less Code = Less Bugs

```
  /* Load table t using history tables.  History tables have
     multiple records per employee.  We need to keep the
     history records for each employee that have the maximum
     status date for that employee.  We do that by numbering
     each history record by empno, keeping only the records such
     that it is the first record for a EMPNO after sorting by
     status_date desc.  REALIZE: this is not deterministic if
     there are two status_dates that are the same for a given
     employee! */
insert /* APPEND */ into t ( .... )
 select EMPNO, STATUS_DATE, ......
    from ( select EMPNO, STATUS_DATE, .... ,
               row_number() OVER ( partition by EMPNO
                                     order by STATUS_DATE desc ) rn
                  from t1, t2, t3, t4, …
                 where … )
   where rn=1;
```

“Here is a last bit of advice on writing as little as possible: When you are writing code, make sure your routines (methods, procedures, functions, or whatever you want to call them) fit on a screen. You should be able to see the logic from start to finish on your monitor. Buy the biggest monitor you can, and make the routines fit on your screen. This rule forces you to think modularly, so you break up the code into bite-sized snippets that are more easily understood”

Use Packages

They break the dependency chain

- Most relevant in Oracle Database 10g Release 2 and before:

```
ops$tkyte%ORA10GR2> create or replace procedure p1 as begin null; end;
ops$tkyte%ORA10GR2> create or replace procedure p2 as begin p1; end;

OBJECT_NAME                    STATUS  TO_CHAR(LAST_DD
------------------------------ ------- ---------------
P1                             VALID   04-oct 12:15:54
P2                             VALID   04-oct 12:15:54

ops$tkyte%ORA10GR2> create or replace procedure p1 as begin /* updated */ null; end;


OBJECT_NAME                    STATUS  TO_CHAR(LAST_DD
------------------------------ ------- ---------------
P1                             VALID   04-oct 12:15:58
P2                             INVALID 04-oct 12:15:54
```

ORACLE

They increase your namespace

- You can have only one procedure P in a schema
  - What about EBR?
  - Killer Feature of 11g Release 2
- With packages, you can have as many procedure P's as you need
  - Less chance of developer X using the same 'name' as developer Y since only package names would clash
- A single package has many procedures/functions
  - Reduces dictionary "clutter"
  - Organizes things, related code goes together
  - Promotes modularity

They support overloading

- A feature which is viewed as
  - Positive by some
  - Negative by others
- Overloading can be very useful in API packages
  - 259 out of 728 ‘SYS’ packages employ this technique

They support encapsulation

- Helps live up to the “fit on a screen” rule
  - Many small subroutines that are no use outside of the package
  - Hide them in the package body, no one can see them
  - Reduces clutter in the dictionary
- Allows you to group related functionality together
  - Makes it obvious what pieces of code are to be used together
- They support elaboration code
  - When package is first invoked, complex initialization code may be executed

Use Static SQL

Static SQL is checked at compile time

- You know the SQL will (probably) execute
  - It is syntactically correct
  - It could still raise an error (divide by zero, conversion error, etc)
  - It might be semantically incorrect, but that is a bug in your logic, not a criticism of static SQL

PL/SQL understands the dictionary

- It will create record types for you
- It will allow you to define variables based on the database types
- The compiler does more work, so you don't have to.

One word - dependencies

- All referenced objects – tables, views, other bits of code, etc – are right there in the dictionary.
- No more “Oh sorry, I didn’t know you were using that”
- If something changes – we know right away if something is broken
  - Grants – lose one that you need, code will stay invalid
  - Drop column – drop one that you reference, code will stay invalid
  - Modify length of column – if you reference that, code will recompile with new size.

Static SQL makes parse once, execute many a reality

- Dynamic SQL makes it easy to lose out on this benefit.
- With DBMS_SQL, you have to cache the ‘cursor’ yourself and make sure you use it over and over (eg: do not call dbms_sql.close() until you *have to*)
- With native dynamic SQL, you need to make sure you use the same SQL text over and over to cache statements
  - And if you are doing that, why did you use dynamic SQL again?
  - Different in 9i and before than 10g and later
- Impossible to be SQL Injected with static SQL! Far too easy to be SQL Injected with dynamic SQL.

Dynamic SQL – when to use then?

- Dynamic SQL is something you want to use when static SQL is no longer practical—when you would be writing *hundreds or thousands* of lines of code, and can replace it with a very small bit of *safe (sql injection)* dynamic SQL.

- When you've shown that using static SQL would not be practical – that is, it is *never your first choice.*

Bulk Up

Bulk Processing Defined:

- A method to bother the database less often
- A method to reduce round trips (even from PL/SQL to SQL – there is a ‘round trip’ involved
- A method to utilize fewer resources in general
- A method to maintain data structures in better shape

- You get some data (more than a row), process it, and send it all back (to update/insert/delete).

Bulk Processing

- You need to do it when…
  - You retrieve data from the database
  - AND you send it back to the database
- You need NOT do it when…
  - You retrieve data from the database
  - <this space left intentionally blank>
  - For example…

Bulk Processing

- You need to do it when… THIS IS BAD CODE

```
For x in ( select * from t where … )
Loop
         process(x);
        update t set … where …;
End loop;
```

  - Implicit array fetch for select
  - Not so for update… Details on next slide
- You need NOT do it when… THIS IS OK CODE

```
For x in (select * from t where …)
Loop
        dbms_output.put_line( … t.x … );
End loop;
```

  - Implicit array fetch for select
  - No going back to database

Bulk Processing

```
create or replace procedure bulk
as
    type ridArray is table of rowid;
    type onameArray is table
             of t.object_name%type;

    cursor c is select rowid rid, object_name
                  from t t_bulk;
    l_rids      ridArray;
    l_onames    onameArray;
    N           number := 100;
begin
    open c;
    loop
        fetch c bulk collect
        into l_rids, l_onames limit N;
        for i in 1 .. l_rids.count
        loop
            l_onames(i)  := substr(l_onames(i),2)
                            ||substr(l_onames(i),1,1);
        end loop;
        forall i in 1 .. l_rids.count
            update t
               set object_name = l_onames(i)
             where rowid = l_rids(i);
        exit when c%notfound;
    end loop;
    close c;
end;
```

```
create or replace procedure slow_by_slow
as
begin
    for x in (select rowid rid, object_name
                from t t_slow_by_slow)
    loop
        x.object_name := substr(x.object_name,2)
                         ||substr(x.object_name,1,1);
        update t
           set object_name = x.object_name
         where rowid = x.rid;
    end loop;
end;
```

Bulk Processing

```
SELECT ROWID RID, OBJECT_NAME FROM T T_BULK

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total      721      0.17       0.17          0      22582          0       71825
********************************************************************************
UPDATE T SET OBJECT_NAME = :B1 WHERE ROWID = :B2

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute    719     12.83      13.77          0      71853      74185       71825
Fetch        0      0.00       0.00          0          0          0           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total      720     12.83      13.77          0      71853      74185       71825
```

```
SELECT ROWID RID, OBJECT_NAME FROM T T_SLOW_BY_SLOW

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total      721      0.17       0.17          0      22582          0       71825
********************************************************************************
UPDATE T SET OBJECT_NAME = :B2 WHERE ROWID = :B1

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute  71824     21.25      22.25          0      71836      73950       71824
Fetch        0      0.00       0.00          0          0          0           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total    71825     21.25      22.25          0      71836      73950       71824
```

But of course, the bulkier the better…

```
SELECT ROWID RID, OBJECT_NAME FROM T T_BULK

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total      721      0.17       0.17          0      22582          0       71825
********************************************************************************
UPDATE T SET OBJECT_NAME = :B1 WHERE ROWID = :B2

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute    719     12.83      13.77          0      71853      74185       71825
Fetch        0      0.00       0.00          0          0          0           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total      720     12.83      13.77          0      71853      74185       71825
```

Lots less code too! (dml error logging if you need)

```
update t set object_name = substr(object_name,2) || substr(object_name,1,1)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      1.30       1.44          0       2166      75736       71825
Fetch        0      0.00       0.00          0          0          0           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        2      1.30       1.44          0       2166      75736       71825
```

Returning Data

To return data to a client program

- Either
  - Simple, formal OUT parameters
  - Ref cursor for all result sets

- Do not run a query
  - To populate a collection
  - To return collection to client

- Just run the query (open CURSOR for SQL_STMT)
  - Ease of programming, everything can handle a cursor
  - Flexibility (client decides how many rows to deal with, less memory intensive)
  - Performance – client might never get to the last row (probably won't)

Implicit versus Explicit

Implicit versus Explicit

- Implicit
  - With this type of cursor, PL/SQL does most of the work for you. You don’t have to open close, declare, or fetch from an implicit cursor.

```
For x in ( select * from t where … )
Loop
         …
End loop;
```

- Explicit
  - With this type of cursor, you do all of the work. You must open, close, fetch, and control an explicit cursor completely.

```
Declare
  cursor c is select * from t where …;
  l_rec  c%rowtype;
Open c;
Loop
   fetch c into l_rec;
   exit when c%notfound;
   …
End loop;
Close c;
```

Implicit versus Explicit

- There is a myth that explicit cursors are superior in performance and usability to implicit cursors.

- The opposite is generally true
  - Implicit cursors have implicit array fetching, Explicit cursors do not
  - Implicit cursors have many of their operations hidden in the PL/SQL runtime (C code) as opposed to explicit cursors being coded by you in PL/SQL code.
  - Implicit cursors are *safer* than explicit cursors code-wise
    - Select into checks (at least and at most one row)
    - Cursors opened/closed for you – implicitly – no ‘leaks’
    - Both implicit and explicit cursors however are cached by PL/SQL
      - But ref cursors are not…

Single Row Processing

- Implicit

```
Select … INTO <plsql variables>
  from …
 where …;
```

- Explicit

```
Declare
   cursor c is select … from … where …;
Begin
   open c;
   fetch c into <plsql variables>;
   if (c%notfound) then
      raise no_data_found;
   end if;
   fetch c into <plsql variables>;
   if (c%found) then
      raise too_many_rows;
   end if;
   close c;
```

- These two bits of code do the same thing. Which is more efficient?

Single Row Processing

- This is a bug
- This is a bug I see a lot
- It is bad, but at least it probably returns NULL

```
Function get_something return …
is
   cursor c is select … from … where …;
Begin
   open c;
   fetch c into <plsql variables>;
   close c;
   return plsql variables
```

- This is a 'worse' bug
- This I see even more
- Combines the "do it yourself join" with "do it the hard way" to get the wrong answer after a longer period of time with lots more development time

```
----------------------------------------

Begin
    open c1;
    loop
        fetch c1 into l_data;
        exit when c1%notfound;
        open c2( l_data );
        fetch c2 into l_something_else;
        close c2;
        process( l_data, l_something_else);
    end loop;
    close c1;
```

Multi-Row Processing

- Which is "easier"?
- Which is more "bug free"?

- Which one array fetches for us?
- The explicit code would be *a lot* more intense if we wanted to array fetch.

```
create or replace procedure implicit
as
begin
    for x in ( select * from dept )
    loop
        null;
    end loop;
end;
----------------------------------------
create or replace procedure explicit
as
    l_rec   dept%rowtype;
    cursor c is select * from dept;
begin
    open c;
    loop
        fetch c into l_rec;
        exit when c%notfound;
    end loop;
    close c;
end;
```

Multi-Row Processing

- This is nice too – all of the benefits of implicit, the factoring out of the SQL usually attributed to explicit cursors.

- This code in the tiny font (to make it fit on the screen) does the same thing as the above bit of code. Which is more efficient? Which is less bug likely?

```
create or replace procedure implicit
As
    cursor c is select * from dept;
begin
    for x in C
    loop
        null;
    end loop;
end;
----------------------------------------
create or replace procedure explicit
as
    type array is table of dept%rowtype;
    l_rec   array;
    n       number := 2;
    cursor c is select * from dept;
begin
    open c;
    loop
        fetch c bulk collect into l_rec limit N;
        for i in 1 .. l_rec.count
        loop
            null;
        end loop;
        exit when c%notfound;
    end loop;
    close c;
end;
```

Multi-Row Processing

- So, is there ever a time to use explicit cursors?
  - Never say Never
  - Never say Always
  - I always say
- Bulk forall processing probably mandates explicit cursors
- Ref Cursors mandate explicit cursors

```
create or replace procedure bulk
as
    type ridArray is table of rowid;
    type onameArray is table of t.object_name%type;
    cursor c is select rowid rid, object_name
                  from t t_bulk;
    l_rids      ridArray;
    l_onames    onameArray;
    N           number := 100;
begin
    open c;
    loop
        fetch c bulk collect
        into l_rids, l_onames limit N;
        for i in 1 .. l_rids.count
        loop
            l_onames(i) := substr(l_onames(i),2)
                           ||substr(l_onames(i),1,1);
        end loop;
        forall i in 1 .. l_rids.count
            update t
               set object_name = l_onames(i)
             where rowid = l_rids(i);
        exit when c%notfound;
    end loop;
    close c;
end;
```

Beware of…

Beware – When others

- When others
- Autonomous Transactions
- Triggers

Things to do…

“Instrument your code. Make it debuggable. Make it so that tracing the code is easy. Identify yourself with DBMS_SESSION. Now DBMS_MONITOR can trace you. Add copious calls to DBMS_APPLICATION_INFO, now we can see who you are, what you are doing and how long you’ve been doing it. Add calls to a logging package (for example http://log4plsql.sourceforge.net/) to enable remote debugging of your code. Use conditional compilation to enable extra intensive debug code to be ‘deployed’ to production.”

“Use the tools – SQL Developer, built in source code debugging. Hierarchical profiling – for performance. Tracing tools to see how your SQL is doing. ASH/AWR reports. PL/SCOPE.   Learn the tools, then use them.”

“Always test things out – especially advice. I used to advise to use BULK COLLECT for everything. That changed in Oracle Database 10g when they started implicitly doing that for us. There is advice on the ‘internet’ to never use implicit cursor – always use explicit. It is wrong. If they suggest it is “faster” and you cannot see it being “faster”, question the advice.”

“Question Authority, Ask Questions”

ORACLE®