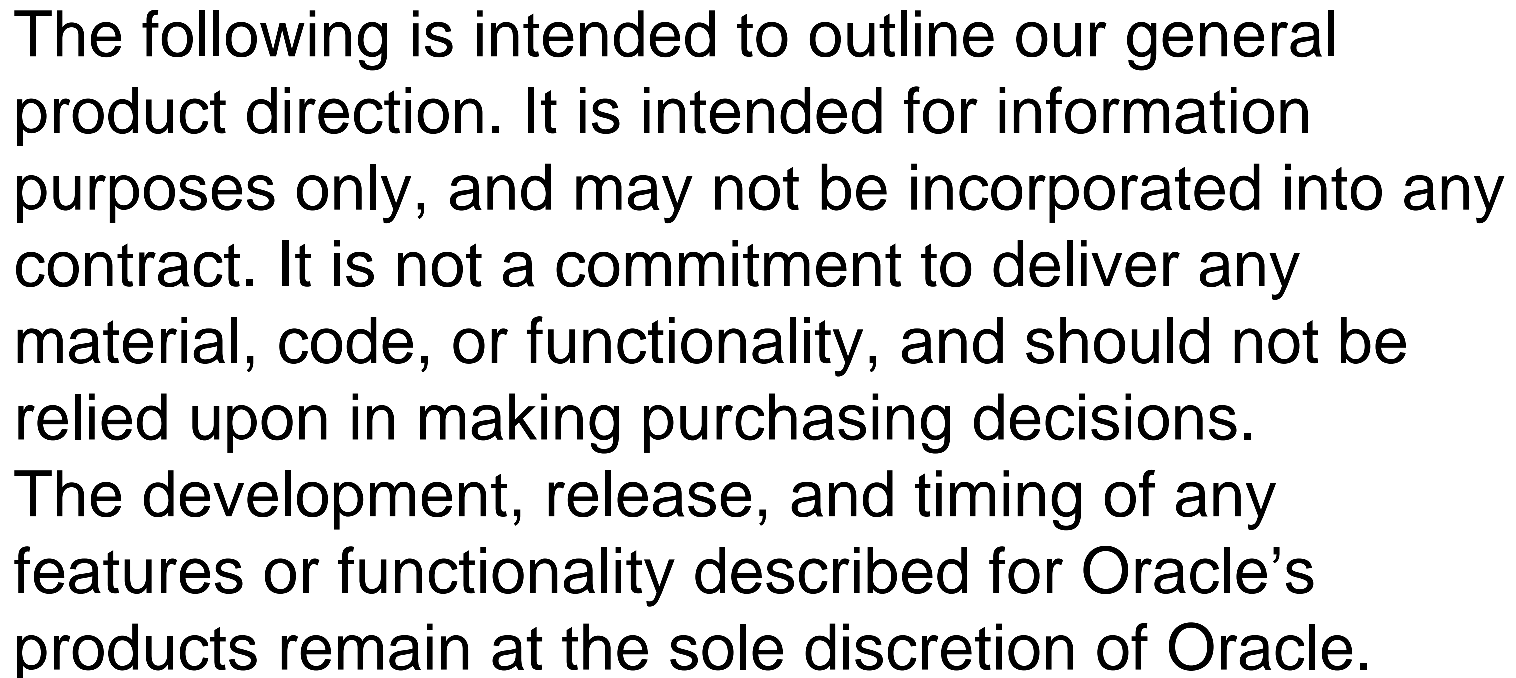
ORACLE

PL/SQL Enhancements in Oracle Database 11*g*

Thomas Kyte
Vice President, Oracle Corporation
http://asktom.oracle.com/


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.

PL/SQL Enhancements in Oracle Database 11*g*

- Every new major release of Oracle Database brings PL/SQL enhancements in these categories
  - Transparent and “knob-controlled” performance improvements
  - New language features that you use in your programs to get better performance
  - New language features that bring functionality that you earlier couldn’t achieve, or could achieve only with cumbersome workarounds
  - New language features that improve the usability of programming in PL/SQL

Transparent performance:

DML triggers are faster

DML triggers are faster

- One of our experiments showed
  - 25% speed-up for the firing update statement on a table with a row-level trigger that does DML to another table
- Your mileage may vary!

Transparent performance:

Fine Grained Dependency Tracking

The challenge

```
create table t(a number)
/
create view v as select a from t
/
alter table t add(Unheard_Of number)
/
select status from User_Objects
 where Object_Name = 'V'
/
```

- View *v* ends up invalid in 10.2 because we know only that its dependency parent has changed – at the granularity of the whole object

The challenge

```
create package Pkg is
  procedure p1;
end Pkg;
/
create procedure p is begin Pkg.p1(); end;
/
create or replace package Pkg is
  procedure p1;
  procedure Unheard_Of;
end Pkg;
/
select status from User_Objects
  where Object_Name = 'P'
/
```

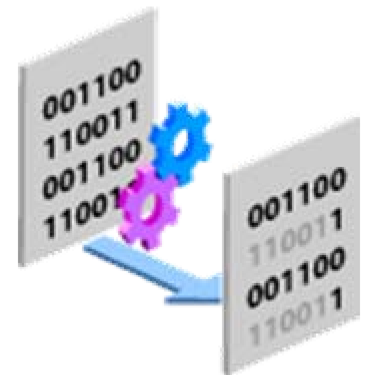
- Same goes for procedure *p*

Fine Grained Dependency Tracking

- In 11.1 we track dependencies at the level of *element within unit*
  - so we know that these changes have no consequence
- I classified this as a transparent performance improvement
  - It's certainly transparent!
  - Unnecessary recompilation certainly consumes CPU
- But – recall the "4068" family of errors – this is better seen as a transparent *availability* improvement

In-Place Redefinition Improvements

- Fast add column with default value
  - Does not need to update all rows to default value
- Invisible Indexes prevent premature use of newly created indexes
- Online index build with NO pause to DML
- No recompilation of dependent objects when
  - Columns added to tables
  - Procedures added to packages
- Easier to execute table DDL operations online
  - Option to wait for active DML operations instead of aborting



Performance “knob”:

Real native compilation

The challenge

- Through 10.2, PL/SQL compiled to a native DLL is significant faster than PL/SQL compiled for interpretation by the PVM
- Oracle translates PL/SQL source to C code and leaves the last step to a 3rd party C compiler
- BUT… some customers' religion forbids a C compiler on a production box!
- AND… other customers' religion forbids paying to license a C compiler when they've already paid to license Oracle Database!

Real native compilation

- In 11.1, Oracle translates PL/SQL source *directly* to the DLL for the current hardware
- Moreover, Oracle does the linking and loading so that the filesystem directories are no longer needed
- So PL/SQL native compilation will work out of the box – and without compromising religion
- Only one parameter remains: the on/off switch, *PLSQL_Code_Type*

Real native compilation

- As a bonus, it's faster!
  - Real native compilation is twice as fast as C native
  - The Whetstone benchmark runs 2.5x faster as real native than as C native
  - Contrived tests have shown 20x
- The new PL/SQL datatype *simple_integer* has semantics that exactly match those of the hardware's integer operations
  - Has a not null constraint
  - Wraps rather than overflowing
  - So it's faster than *pls_integer*

Performance “knob”:

Intra-unit inlining

The challenge

- Helper subprograms are used (as Steven Feuerstein teaches) to improve understandability
- Often, these are short
- Programmers sometimes agonize over the dilemma:

  readability/correctness/maintainability
  versus
  performance

The challenge

```
procedure p(Input_String varchar2) is
  ...
  function Found_Another_Word(w out varchar2)
    return boolean is ... ;
  function Is_Article(w in varchar2)
    return boolean is ... ;
begin
  while Found_Another_Word(Word) loop
    if Is_Article(Word) then
      Article_Count := Article_Count + 1;
    end if;
  end loop;
end p;
```

The challenge

```
function Found_Another_Word(w out varchar2)
  return boolean is
begin
  End_Pos := Instr(v, Space, Start_Pos);
  if End_Pos > 0 then
    w := Substr(v, Start_Pos, (End_Pos-
   Start_Pos));
    Start_Pos := End_Pos + 1;
    while Substr(v, Start_Pos, 1) = Space loop
      Start_Pos := Start_Pos + 1;
    end loop;
    return true;
  else
    return false;
  end if;
end Found_Another_Word;
```

Intra-unit inlining

```
alter procedure p compile
  PLSQL_Optimize_Level = 2
  reuse settings
/
begin p(:Big_Doc); end;
/
alter procedure p compile
  PLSQL_Optimize_Level = 3 -- New in 11.1
  reuse settings
/
begin p(:Big_Doc); end;
/
```

- ~700 milliseconds for level 2
- ~400 milliseconds for level 3

Intra-unit inlining

- Your mileage may vary!
- Using a test taken from the E-Business Suite
  - “Flexfields”
  - Pure PL/SQL data munging
  - Large package with many helper subprograms
  - Showed 20% speedup
- Using the PL/SQL Team’s benchmark suite
  - Some of the tests have no inlining opportunities
  - Showed average of 10% speedup

Performance language feature:

SQL & PL/SQL Result Caches

The challenge

- Find the greatest average value of income grouped by state over the whole population – or some similar metric
- Huge number of rows yield a few or one row
- The data changes fairly slowly (say every hour) but the query is repeated fairly often (say every second)

The challenge

```
function f1 return t1%rowtype is
  r t1%rowtype;
begin
  select a, m
  into r.a, r.b
  from (
    select a, sb m from (
      select a, Sum(b) sb from t1
      group by a)
    order by m desc)
  where Rownum = 1;
  return r;
end f1;
```

- ~ 1,000 milliseconds for each new call

SQL Query Result Cache

```
function f1 return t1%rowtype is
  r t1%rowtype;
begin
  select /*+ result_cache */ a, m
  into r.a, r.b
  from (
    select a, sb m from (
      select a, Sum(b) sb from t1
      group by a)
    order by m desc)
  where Rownum = 1;
  return r;
end f1;
```

- ~ 0 milliseconds for each new call

The challenge

- Calculate a yet more complex derived metric – like the ratio of the highest median income grouped by state to the lowest median income grouped by state over the whole population
- Now we need a PL/SQL function
- Again, the data changes fairly slowly (say every hour) but the query is repeated fairly often (say every second)

The challenge

```
function f2 return t1%rowtype

is
   ...
begin
   select a, m into r1.a, r1.b from ...;

   select a, m into r2.a, r2.b from ...;

   r.a := r1.a + r2.a;
   r.b := r1.b + r2.b;
   return r;
end f2;
```

- ~ 2,000 milliseconds for each new call

PL/SQL Function Result Cache

```
function f2 return t1%rowtype
  result_cache relies_on(t1, t2)
is
  ...
begin
  select a, m into r1.a, r1.b from ...;

  select a, m into r2.a, r2.b from ...;

  r.a := r1.a + r2.a;
  r.b := r1.b + r2.b;
  return r;
end f2;
```

- ~ 0 milliseconds for each new call

SQL & PL/SQL Result Caches

- Both are cross-session and RAC interoperable
- Both build on the same infrastructure
  - Same *Result_Cache_Size*,… initialization parameters
  - Same *DBMS_Result_Cache* management package
  - Same *v$Result_Cache_** performance views

Performance language feature:

The compound trigger

The challenge

- Insert a row into a separate audit table each time an employee's salary is changed
- Typically, very many employee rows are changed by a single update
- Find a way to use bulk inserts for the audit rows
- Through 10.2, programmers have used the "ancillary package paradigm"
  - Initialize package globals in "before statement"; batch and flush rows in "before each row; final flush in "after statement"

The compound trigger

- A compound trigger lets you implement actions for each of the table DML timing points in a single trigger
- You can define variables that are global for these sections
  - The declarations are elaborated at “before statement” time
  - You can provide explicit initialization code in the “before statement” section
  - You can provide finalization code in the “after statement” section
  - The globals are destroyed when the firing SQL finishes

The compound trigger

```
create trigger My_Compound_Trg
  for update of Salary on Employees
compound trigger
  -- These variables have firing-statement duration
  Threshold constant pls_integer := 200;

  before statement is
  begin
    ...
  end before statement;

  -- And/or "after each row"
  before each row is
  begin
    null;
  end before each row;

  after statement is
  begin
    null;
  end after statement;
end My_Compound_Trg;
/
```

The compound trigger

```
create trigger My_Compound_Trg
  for update of Salary on Employees
compound trigger

  Threshold   constant pls_integer := 200;
  type Emps_t  is table of Employee_Salaries%rowtype
    index by pls_integer;
  Emps         Emps_t;
  Idx          pls_integer := 0;

  procedure Flush_Array is
  begin
    forall j in 1..Emps.Count()
      insert into Employee_Salaries values Emps(j);
    Emps.Delete();
    Idx := 0;
  end Flush_Array;


  ...

end My_Compound_Trg;
/
```

The compound trigger

```
create trigger My_Compound_Trg
  for update of Salary on Employees
compound trigger

  ...

  after each row is
  begin
    Idx := Idx + 1;
    Emps(Idx).Employee_Id    := :New.Employee_Id;
    Emps(Idx).Salary         := :New.Salary;
    Emps(Idx).Effective_Date := Sysdate();

    if Idx >= Threshold then
      Flush_Array();
    end if;
  end after each row;


  ...

end My_Compound_Trg;
/
```

The compound trigger

```
create trigger My_Compound_Trg
  for update of Salary on Employees
compound trigger

  ...

  after statement is
  begin
    Flush_Array();
  end after statement;

end My_Compound_Trg;
/
```

Functionality:

Dynamic SQL Functional Completeness

The challenge

- You want to generate a big PL/SQL unit whose source exceeds 32k characters
- You want you expose the database only via PL/SQL subprograms; for queries with unbounded result sets you use *ref cursors*. Now the requirements change and you don't know the *where* clause – and hence the number of binds – until run-time
- The number of binds is not known until run-time but the select list is fixed; you want to use native dynamic SQL's bulk fetch

B.t.w., method 4 is what it is

- Method 4 means you don’t know the number of defines (i.e. the select list) or the number of binds until run-time
- Therefore, you need to discover the number and datatypes of the select list columns
- After much debate, we agreed that the nature of the steps that method 4 requires are better expressed via a procedural API than via language syntax
- *DBMS_Sql* is here to stay!

Dynamic SQL Functional Completeness

- *execute immediate* takes a clob
- For symmetry, *DBMS_Sql.Parse()* takes a clob
- Can transform a ref cursor into a *DBMS_Sql* cursor and vice versa
- DBMS_Sql supports ADTs
- You can do *DBMS_Sql* bulk binding with collections of your own datatype – just as you can with native dynamic SQL

Dynamic SQL Functional Completeness

```
  ...
  Cur_Num number := DBMS_Sql.Open_Cursor();
  rc Sys_Refcursor;

  cursor e is select Employee_ID, First_Name, Last_Name
                from Employees;
  type Emps_t is table of e%rowtype;
  Emps Emps_t;
begin
  DBMS_Sql.Parse(
    c=>Cur_Num, Language_Flag=>DBMS_Sql.Native, Statement=>
      'select Employee_ID, First_Name, Last_Name
         from Employees
         where Department_ID = :d and Salary > :s and ...');

  DBMS_Sql.Bind_Variable(Cur_Num, ':d', Department_ID);
  DBMS_Sql.Bind_Variable(Cur_Num, ':s', Salary);
  ...
  Dummy := DBMS_Sql.Execute(Cur_Num);
  -- Switch to ref cursor and native dynamic SQL
  rc := DBMS_Sql.To_Refcursor(Cur_Num);

  fetch rc bulk collect into Emps;
  close rc;
  ...
```

Functionality:

Fine Grained Access Control for *Utl_TCP* and its cousins

The challenge

- Oracle Database provides packaged APIs for PL/SQL subprograms to access machines (specified by host and port) using bare TCP/IP and other protocols built on it (SMTP and HTTP).
- *Utl_TCP*, *Utl_SMTP*, *Utl_HTTP*…
- If you have *Execute* on the package, you can access ANY host-port
- It's of minor interest whether the *Execute* flows via *public* or is granted directly

Fine Grained Access Control for *Utl_TCP* and its cousins

- An Access Control Element (ACE) specifies an allowed host-port
- An Access Control List (ACL) specifies a user's ACEs
- The ACEs and ACLs are managed by XDB

Functionality:

Regular expression enhancements in SQL and PL/SQL

The challenge

```
p := '\(?\d{3}\)? ?\d{3}[-.]\d{4}';

Str :=
  'bla bla (123)345-7890 bla bla
  (345)678-9012 bla bla (567)890-1234 bla bla';

Match_Found := Regexp_Like(Str, p);
```

- OK, there was at least one match. But how many are there?
- Tedious to step along *Str* finding each successive match, incrementing Pos, and counting yourself!

Regular expression enhancements in SQL and PL/SQL

```
No_Of_Matches := Regexp_Count(Str, p);
```

- *Regexp_Instr* and *Regexp_Substr* now have an optional *Subexpr* parameter that lets you target a particular substring of the regular expression being evaluated.

Functionality:

Support for “super”

The challenge

- The *Employee* supertype has an overridable member function *Monthly_Pay()* that calculates the generic basic
- The *Salesperson* subtype specializes *Monthly_Pay()* to acknowledge notions like commission based on actual sales made
- The natural implementation has *Salesperson. Monthly_Pay()* calling *Employee. Monthly_Pay()*
- Guess what? Through 10.2 you can't do it without a cumbersome workaround

Support for “super”

- The OO paradigm specifies the solution
- ANSI describes it
- It’s colloquially known as support for “super”
- 11.1 introduces this
- If you don’t know what this is, you don’t need it!

Functionality:

Read-only table
Create a disabled trigger
Specify trigger firing order
***New PLW-06009 warning (my *favorite*)**

Read-only table

```
alter table t read only
/

...

alter table t read write
/
```

- What more can I say?

Create a disabled trigger

```
create or replace trigger Trg
  before insert on My_Table for each row
  disable
begin
  :New.ID := My_Seq.Nextvak;
end;
/
```

- If you create a trigger whose body has a PL/SQL compilation error, then DML to the table fails with *"ORA-04098: trigger 'TRG' is invalid and failed re-validation"*

- So it's safer to create it disabled and to enable it only when you know it compiled without error

Specify trigger firing order

```
create or replace trigger Trg_2
  before insert on My_Table for each row
  follows Trg_1
begin
  ...
end;
/
```

- Through 10.1, you might have thought that you knew the firing order (by experimental observation) but you famously couldn't rely on it

The challenge

```
create procedure p(i in number) is
begin
  insert into My_Table(n) values(i);
exception
  when others then null;
end p;
/
```

- Someone else writes *"when others then null"* because they expect only the *Dup_Val_On_Index* exception – but (amazingly) want to "make sure" that the program won't fail.
- Now you've inherited this code and you realize that exceptions are getting swallowed

New PLW-06009 warning

```
alter procedure p compile
  PLSQL_Warnings = 'enable:all'
  reuse settings
/
```

- This now draws a warning:

```
PLW-06009: procedure "P" OTHERS handler
           does not end in RAISE or
           RAISE_APPLICATION_ERROR
```

Usability of the language:

Sequence in a PL/SQL expression

The challenge

```
create or replace trigger Trg
  before insert on My_Table for each row
declare
  s number;
begin
 -- Annoying locution
  select My_Seq.Nextval into s from Dual;
  :New.PK := n;
end;
/
```

- There’s also a performance concern

Sequence in a PL/SQL expression

```
create or replace trigger Trg
  before insert on My_Table for each row


begin


   :New.ID := My_Seq.Nextval;
end;
/
```

- Happily, the performance concern is solved generically for *any* simple “select… from Dual”

Usability of the language:

The *continue* statement

The challenge

```
<<Outer>>for i in 1..10 loop

  ...

  <<Inner>>for j in 1..Data.Count() loop
    if not Data(j).Uninteresting then
      ...
    end if;
  end loop;
end loop;
```

- The logic is cumbersome and back to front…
- …especially if, on the condition you detect, you want to start the next iteration of an enclosing loop

The *continue* statement

```
<<Outer>>for i in 1..10 loop

  ...

  <<Inner>>for j in 1..Data.Count() loop
    continue Outer when Data(j).Uninteresting;
    ...

  end loop;
end loop;
```

- Many algorithms are described, in pseudocode, using the *continue* statement

Usability of the language:

Named and Mixed Notation from SQL

The challenge

```
create function f(
  p1 in number default 1,
  ...,
  p5 in number default 5) return number
is
 v number
begin
  ...
  return v;
end f;
/
select f(p4 => 10) from Dual
/
ORA-00907: missing right parenthesis
```

Named and Mixed Notation from SQL

```
select f(p4 => 10) from Dual
/
 F(P4=>10)
----------
        21
```

Summary

Summary

- Performance
  - Transparent DML trigger performance improvement
  - Finer grained dependency tracking
  - Real PL/SQL native compilation
  - Intra-unit inlining
  - SQL & PL/SQL Result Caches
  - The compound trigger
- Notice how little effort it takes to get the benefit of these features

Summary

- Functionality
  - Dynamic SQL functional completeness
  - Fine grained access control for *Utl_TCP*, etc
  - *Regexp_Count()*, etc in SQL and PL/SQL
  - Support for “super”
  - alter table t read only
  - Create a disabled trigger; specify trigger firing order
  - “when others then null” compile-time warning

Summary

- Usability
  - Sequence in a PL/SQL expression
  - The *continue* statement
  - Named and mixed notation from SQL

Q&A