
ORACLE®

ORACLE

New PL/SQL Capabilities in Oracle Database 12*c*

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 remains at the sole discretion of Oracle.

Agenda

- Improved client <> PL/SQL <> SQL interoperability
- A new security capability
- Improved programmer usability
- Miscellaneous

Performance improvement for PL/SQL functions called from SQL

- *Example:* pretty-print an integer

```
select      PK,
  Print(n1) "n1",
  Print(n2) "n2",
  Print(n3) "n3"
from t
```

```
 1       1 K          1 G        566 G
 2       1 K        157 M        416 G
 3       2 K          1 G        971 G
 4     578 byte       1 G          1 T
 5       2 K          1 G        220 G
 6       1 K          2 G          1 T
 7      48 byte       1 G          2 T
 8     992 byte      42 M          3 T
 9     794 byte       2 G          1 T
10       2 K        302 M        672 G
```

The “algorithm”

- Pretty-print an integer as a multiple of an appropriate power of 1024: plain, K, M, B, or T

```
function Print(n in integer) return varchar2 authid Definer is
  K  constant number not null := 1024;
  M  constant number not null := K*K;
  G  constant number not null := M*K;
  T  constant number not null := G*K;
begin
  return
    case
      when n   <= K-1 then To_Char(n,   '999999')||'byte'
      when n/K <= K-1 then To_Char(n/K, '999999')||'K'
      when n/M <= K-1 then To_Char(n/M, '999999')||'M'
      when n/G <= K-1 then To_Char(n/G, '999999')||'G'
      else                 To_Char(n/T, '999999')||'T'
    end;
end Print;
```

Try it in pure SQL!

```
select
  PK,
  case
    when n1            <= 1023 then To_Char(n1,               '999999')||' byte'
    when n1/1024       <= 1023 then To_Char(n1/1024,          '999999')||' K'
    when n1/1048576    <= 1023 then To_Char(n1/1048576,       '999999')||' M'
    when n1/1073741824 <= 1023 then To_Char(n1/1073741824,    '999999')||' G'
    else                       To_Char(n1/1099511627776, '999999')||' T'
  end
  "n1",
  case
    when n2            <= 1023 then To_Char(n2,               '999999')||' byte'
    when n2/1024       <= 1023 then To_Char(n2/1024,          '999999')||' K'
    when n2/1048576    <= 1023 then To_Char(n2/1048576,       '999999')||' M'
    when n2/1073741824 <= 1023 then To_Char(n2/1073741824,    '999999')||' G'
    else                       To_Char(n2/1099511627776, '999999')||' T'
  end
  "n2",
  case
    when n3            <= 1023 then To_Char(n3,               '999999')||' byte'
    when n3/1024       <= 1023 then To_Char(n3/1024,          '999999')||' K'
    when n3/1048576    <= 1023 then To_Char(n3/1048576,       '999999')||' M'
    when n3/1073741824 <= 1023 then To_Char(n3/1073741824,    '999999')||' G'
    else                       To_Char(n3/1099511627776, '999999')||' T'
  end
  "n3"
from t
```

Get the performance of SQL with the clarity and reusability of PL/SQL

```
function Print(n in integer) return varchar2 authid Definer is
  pragma UDF;
  K  constant number not null := 1024;
  M  constant number not null := K*K;
  G  constant number not null := M*K;
  T  constant number not null := G*K;
begin
  return
    case
      when n   <= K-1 then To_Char(n,   '999999')||'byte'
      when n/K <= K-1 then To_Char(n/K, '999999')||'K'
      when n/M <= K-1 then To_Char(n/M, '999999')||'M'
      when n/G <= K-1 then To_Char(n/G, '999999')||'G'
      else                 To_Char(n/T, '999999')||'T'
    end;
end Print;
```

Declare the PL/SQL function in the subquery's *with* clause

```
with
  function Print(n in integer) return varchar2 is
    K  constant number not null := 1024;
    M  constant number not null := K*K;
    G  constant number not null := M*K;
    T  constant number not null := G*K;
  begin
    return
      case
        when n   <= K-1 then To_Char(n,   '999999')||' byte'
        when n/K <= K-1 then To_Char(n/K, '999999')||' K'
        when n/M <= K-1 then To_Char(n/M, '999999')||' M'
        when n/G <= K-1 then To_Char(n/G, '999999')||' G'
        else                 To_Char(n/T, '999999')||' T'
      end;
  end Print;
select       PK,
  Print(n1)  "n1",
  Print(n2)  "n2",
  Print(n3)  "n3"
from t
```

Performance comparison

▪ Pure SQL is fastest	5.0x
▪ Schema-level function with pragma UDF is close	3.9x
▪ Function in the with clause is similar	3.8x
▪ Pre-12.1 ordinary schema-level function is very much the slowest	1.0 – the baseline

Binding values of PL/SQL-only datatypes into SQL statements

- Before 12.1, you could bind only values of SQL datatypes
- In 12.1, you can bind PL/SQL index-by-pls_integer tables (of records) and booleans
  - **from** client-side programs – OCI or both flavors of JDBC – and from PL/SQL
  - **to** anonymous blocks, statements using functions, or statements using the *table* operator

Binding a PL/SQL index-by table to SQL

- Before 12.1, you could invoke a function with a collection actual, or select from a collection, but
  - The type had to be defined at schema-level
  - Therefore it had to be a nested table or a varray
  - A non-scalar payload had to be an ADT
- New in 12.1
  - The type can be defined in a package spec – can be *index by pls_integer* table
  - The payload can be a *record* – but the fields must still be SQL datatypes

The collection

```
package Pkg authid Definer is
  type r is record(n integer, v varchar2(10));
  type t is table of r index by pls_integer;
  x t;
end Pkg;
```

Example:
binding an IBPI to a PL/SQL function in SQL

```
function f(x in Pkg.t) return varchar2 authid Definer is
  r varchar2(80);
begin
  for j in 1..x.Count() loop
    r := r||...;
  end loop;
  return r;
end f;
```

```
procedure Bind_IBPI_To_Fn_In_SQL authid Definer is
  v varchar2(80);
begin
  select f(Pkg.x) into v from Dual;
  ...
  execute immediate 'select f(:b) from Dual' into v
    using Pkg.x;
end Bind_IBPI_To_Fn_In_SQL;
```

Example:
binding to the operand of the *table* operator

```
procedure Select_From_IBPI authid Definer is
  y Pkg.t;
begin
  for j in (select n, v from table(Pkg.x)) loop
    ...
  end loop;

  execute immediate 'select n, v from table(:b)'
  bulk collect into y
  using Pkg.x;
  for j in 1..y.Count() loop
    ...
  end loop;
end Select_From_IBPI;
```

Example:
binding an IBPI to an anonymous block

```
procedure p1(x in Pkg.t) authid Definer is
begin
  for j in 1..x.Count() loop
    ...;
  end loop;
end p1;
```

```
procedure Bind_IBPI_To_Anon_Block authid Definer is
begin
  execute immediate 'begin p1(:b); end;' using Pkg.x;
end Bind_IBPI_To_Anon_Block;
```

Example:
binding a boolean to an anonymous block

```
procedure p2(b in boolean) authid Definer is
begin
  DBMS_Output.Put_Line(case b
                         when true  then 'True'
                         when false then 'False'
                         else            'Null'
                       end);
end p2;
```

```
procedure Bind_Boolean_To_Anon_Block authid Definer is
  Nil constant boolean := null; -- workaround for existing bug
begin
  execute immediate 'begin p2(:b); end;' using true;
  execute immediate 'begin p2(:b); end;' using false;
  execute immediate 'begin p2(:b); end;' using Nil;
end Bind_Boolean_To_Anon_Block;
```

Binding PL/SQL types in JDBC

- Before 12.1
  - Generate a schema level object type to mirror the structure of the non-SQL package type
  - Populate and bind the object into a custom PL/SQL wrapper around the desired PL/SQL subprogram
  - Convert the object to the package type in the wrapper and call the PL/SQL subprogram with the package type

Binding PL/SQL types in JDBC

- New in 12.1
  - PL/SQL package types supported as binds in JDBC
  - Can now execute PL/SQL subprograms with non-SQL types
  - Supported types include *records*, *index-by tables*, *nested tables* and *varrays*
  - *Table%rowtype*, *view%rowtype* and package defined *cursor%rowtype* also supported. They're technically record types

Example 1: Bind a single record from Java into a PL/SQL procedure, modify it, and bind it back out to Java

```
package Emp_Info is
  type employee is record(First_Name   Employees.First_Name%type,
                          Last_Name    Employees.Last_Name%type,
                          Employee_Id  Employees.Employee_Id%type,
                          Is_CEO       boolean);

  procedure Get_Emp_Name(Emp_p in out Employee);
end;
```

Example 1:

- Use the *EmpinfoEmployee* class, generated by JPub, to implement the *Employee* formal parameter

```
{  …
      EmpinfoEmployee Employee = new EmpinfoEmployee();
      Employee.setEmployeeId(new java.math.BigDecimal(100));  // Use Employee ID 100

      // Call Get_Emp_Name() with the Employee object
      OracleCallableStatement cstmt =
         (OracleCallableStatement)conn.prepareCall("call EmpInfo.Get_Emp_Name(?)");
      cstmt.setObject(1, Employee, OracleTypes.STRUCT);

      // Use "PACKAGE.TYPE NAME" as the type name
      cstmt.registerOutParameter(1, OracleTypes.STRUCT, "EMPINFO.EMPLOYEE");
      cstmt.execute();

      // Get and print the contents of the Employee object
      EmpinfoEmployee oraData =
           (EmpinfoEmployee)cstmt.getORAData(1, EmpinfoEmployee.getORADataFactory());
      System.out.println("Employee: " + oraData.getFirstName() + " " + oraData.getLastName());
      System.out.println("Is the CEO? " + oraData.getIsceo());
}
```

Example 2: populate a collection of *table%rowtype* using a bulk collect statement, and pass the collection as an *out* parameter back to the caller

```
package EmpRow is
  type Table_of_Emp is table of Employees%Rowtype;
  procedure GetEmps(Out_Rows out Table_of_Emp);
end;
```

```
package Body EmpRow is
  procedure GetEmps(Out_Rows out Table_of_Emp) is
  begin
    select *
    bulk collect into Out_Rows
    from Employees;
  end;
end;
```

Example 2:

```
{ …
   // Call GetEmps() to get the ARRAY of table row data objects
   CallableStatement cstmt = conn.prepareCall("call EmpRow.GetEmps(?)");

    // Use "PACKAGE.COLLECTION NAME" as the type name
   cstmt.registerOutParameter(1, OracleTypes.ARRAY, "EMPROW.TABLE_OF_EMP");
   cstmt.execute();

   // Print the Employee Table rows
   Array a = cstmt.getArray(1);
   String s = Debug.printArray ((ARRAY)a, "",
                                ((ARRAY)a).getSQLTypeName () +"( ", conn);
   System.out.println(s);
}
```

Binding PL/SQL-only datatypes into SQL statements: restrictions

- The PL/SQL-only datatypes must be declared in a package spec
- The record fields of the IBPI must be SQL datatypes
- Only IBPI, not index-by-varchar2
- Cannot bind into *insert*, *update*, *delete*, or *merge*
- Cannot bind using DBMS_Sql

Agenda

- Improved client <> PL/SQL <> SQL interoperability
- **A new security capability**
- Improved programmer usability
- Miscellaneous

Granting a role to a PL/SQL unit

- Consider this best practice
  - Give access to an application's data only via PL/SQL subprograms
  - Reinforce this by having end-user sessions authorize as a different database owner than the one that owns the application's artifacts
  - Arrange this by using definer's rights units in a single schema or a couple of schemas. Then grant Execute on these to end-users – but don't grant privileges on the tables to end-users
- This means that each unit can access very many tables because the owner of the units can

Granting a role to a PL/SQL unit

- 12.1 lets us have a fine-grained scheme where each unit with the same owner can have different privileges on the owner's tables
  - The end-user is low-privileged, just as in the old scheme
  - The units are invoker's rights, so "as is" would not allow end-users to access the data
  - The privilege for each unit is elevated for exactly and only that unit's purpose by granting a role that has the appropriate privileges to the unit. Such a role cannot be disabled.
  - The unit's owner must already have that same role (but it need not be enabled)

Granting a role to a PL/SQL unit

- This scenario lets us illustrate the idea
  - There are two users *App* and *Client*
  - There are two tables *App.t1* and *App.t2*
  - There are two IR procedures *App.Show_t1* and *App.Show_t2* to run *select* statements against the tables
  - Client has *Execute* on *App.Show_t1* and *App.Show_t2*
  - *App* creates two roles *r_Show_t1* and *r_Show_t2*
  - *App* grants *Select* on *App.t1* to *r_Show_t1* – and similar for ~2
  - *App* grants *r_Show_t1* to *App.Show_t1* – and similar for ~2

Granting a role to a PL/SQL unit

```
create procedure Show_t1 authid Current_User is
begin
  for j in (select Fact from App.t1 order by 1) loop -- Notice the schema-qualification
    ...
  end loop;
end Show_t1;
/
grant Execute on App.Show_t1 to Client
/
-- this has the side-effect of granting the role to App with Admin option
-- other non-schema object types like directories and editions behave the same
create role r_Show_t1
/
grant select on t1 to r_Show_t1
/
grant r_Show_t1 to procedure Show_t1
/
```

```
select Object_Name, Object_Type, Role
from User_Code_Role_Privs
/
```

```
.......     .........   .........
SHOW_T1     PROCEDURE   R_SHOW_T1
```

Granting a role to a PL/SQL unit

- When *Client* invokes *App.Show_t1*, then no matter what careless mistakes the programmer of the procedure might later make, its power is limited to just what the role confers.

Granting a role to a PL/SQL unit

- This new feature has no effect on static references at PL/SQL compilation time

The “inherit privileges” privilege

- Functional requirement
  - Reduce the risk that would be caused should Oracle-shipped code owned by a highly privileged user (esp. e.g. Sys) have a SQL injection vulnerability.
  - An IR unit executes with the security regime of the invoker. So if a DR unit owned by Sys has an injection vulnerability, then an unscrupulous person who can authorize a session as a Scott-like user could write an IR unit and exploit the injection vulnerability to get it invoked with Sys's security regime.
  - The new feature closes this loophole because, as shipped, Sys has granted "inherit privileges" only to a small number of other Oracle-maintained users. The same holds for about 30 other Oracle-maintained users.

The “inherit privileges” privilege

- Follow-on requirement
  - Had to cause no change in behavior for customer-created code -- at least to the extent that this followed Oracle's guidelines
  - Caveat is illustrated by an extant customer-created DR unit owned by Sys that called an IR unit owned by a customer-created user. This would break on upgrade to 12.1. But this is so very much against the rules that we're comfortable with this.

“bequeath Current_User” views

- The Current_User who issues the SQL against the view is seen in IR functions invoked in the view’s defining subquery
- Compare this with the “classic” DR view where the view owner is seen in IR functions invoked in the view’s defining subquery

Agenda

- Improved client <> PL/SQL <> SQL interoperability
- A new security capability
- **Improved programmer usability**
- Miscellaneous

Whitelist

- You can declare that a particular unit may be referenced only by other listed units
- You cannot list the anonymous block and so a whitelisted unit cannot be called dynamically and cannot be invoked from outside of the database

accessible by clause

```
package Helper authid Definer accessible by (Good_Unit, Bad_Unit)
is
  procedure p;
end Helper;
```

```
package body Good_Unit is
  procedure p is
  begin
    Helper.p();
    ...
  end p;
end Good_Guy;
```

```
package body Bad_Unit is
  procedure p is
  begin
    Helper.p();
    ...
  end p;
end Bad_Guy;
```

PLS-00904: insufficient privilege to access object HELPER

Improved call stack introspection

- Before 12.1, you used three functions in the *DBMS_Utility package*
  - *Format_Call_Stack()*
  - *Format_Error_Stack()*
  - *Format_Error_Backtrace()*
- New in 12.1
  - The package *UTL_Call_Stack* solves the same problem properly

Code to be introspected

```
package body Pkg is
  procedure p is
    procedure q is
      procedure r is
        procedure p is
        begin
          Print_Call_Stack();
        end p;
      begin
        p();
      end r;
    begin
      r();
    end q;
  begin
    q();
  end p;
end Pkg;
```

Pre 12.1 Print_Call_Stack()

```
procedure Print_Call_Stack authid Definer is

begin
  DBMS_Output.Put_Line(DBMS_Utility.Format_Call_Stack());
end;
```

```
----- PL/SQL Call Stack -----
  object      line  object
  handle    number  name
0x631f6e88        12  procedure USR.PRINT_CALL_STACK
0x68587700         7  package body USR.PKG
0x68587700        10  package body USR.PKG
0x68587700        13  package body USR.PKG
0x68587700        16  package body USR.PKG
0x69253ca8         1  anonymous block
```

- See bug 2769809 filed by Bryn, Jan 2003

12.1 Print_Call_Stack()

```
procedure Print_Call_Stack authid Definer is
  Depth pls_integer := UTL_Call_Stack.Dynamic_Depth();
begin
  for j in reverse 2..Depth loop
    DBMS_Output.Put_Line(
      (j - 1)||
      To_Char(UTL_Call_Stack.Unit_Line(j), '99')||
      UTL_Call_Stack.Concatenate_Subprogram(UTL_Call_Stack.Subprogram(j)));
  end loop;
end;
```

```
5   1 __anonymous_block
4  16 PKG.P
3  13 PKG.P.Q
2  10 PKG.P.Q.R
1   7 PKG.P.Q.R.P
```

Improved call stack introspection

- Symmetrical subprograms for error stack and backtrace
- Plus
  - Owner(Depth)
  - Current_Edition(Depth)
  - Lexical_Depth(Depth)

Agenda

- Improved client <> PL/SQL <> SQL interoperability
- A new security capability
- Improved programmer usability
- **Miscellaneous**

Other enhancements brought by 12.1

- You can now result-cache an invoker's rights function (the current user becomes part of the cache lookup key)
- Safe callouts (implemented via *extproc*) are faster (motivated by Oracle R Enterprise – which saw a 20x speedup)
- Edition-based redefinition can now be adopted without needing to change how objects are disposed among schemas – so no reason at all for you not to use EBR for every patch that changes only PL/SQL, views, or synonyms

Other enhancements brought by 12.1

- *pga_aggregate_limit* – exceeding, e.g. by allowing a collection to become too big, it causes a fatal error
- *DBMS_Scheduler* has new *Job_Types*:
  - *Sql_Script*
  - *Backup_Script*
- Controlled by a new use of a credential
  - encapsulates database username, password, and role – e.g. AS SYSDBA, AS SYSBACKUP

Hardware and Software

ORACLE

Engineered to Work Together


ORACLE®