

Beyond Basic Datatypes

SQL Objects and PL/SQL

Who am I ?

- ž Gary Myers
- ž Oracle developer since 1994
- ž Database Consultant with SMS M&T
- ž Blogger since 2004
 - Now at blog.sydoracle.com
- ž Twitter at [syd_oracle](#)
- ž Guest editor at Log Buffer
- ž "Oracle" badge at Stackoverflow forums
- ž Find me at LinkedIn...

Did a bit of COBOL and IDMS-X
Then Ingres for a couple of years
Found Oracle around the Forms 3 Era, used a lot of Pro*Cobol
Came to Australia about 12 years ago from England.
Forms work in Perth and Adelaide.
Arrived in Sydney in '99

Contracted with the Office of State Revenue, P&O Nedlloyd, MBF Health Insurance...
Consultant with DWS for a year, contracted for a bit more.
Been consulting with SMS for about 18 months
Currently at SiCorp on a very small data warehouse.

What is PL/SQL

- ž PL/SQL is an **EXTENSION** to SQL
- ž PL/SQL is procedural
- ž Closely tied to the database
- ž Runs under the oracle shadow process
- ž Mostly in the PGA
 - BEWARE Shared Server (eg APEX)

SQL is very good at set-based operations.

When the task isn't set based, it is worth considering a procedural approach.

Processing of large data volumes are more likely to be amenable to set-based solutions.

Because PL/SQL runs in the database, and as the same process performing SQL operations, there is a smaller detrimental impact from context switching than you may get from a client language containing SQL (e.g. Perl, Python)

In Shared Server, session persistent data is stored in the UGA which is in the SGA, but data that is only required for the duration of the call may still be in the PGA.

PL/SQL Datatypes

- ž As an extension, PL/SQL recognizes all SQL datatypes (**except LONG**)
- ž That includes user-defined datatypes
- ž User-defined datatypes includes Oracle supplied ones such as Spatial (MDSYS) and Multimedia (ORDSYS).

By not requiring a translation between the SQL types and PL/SQL types, any impedance is reduced.

PL/SQL specific datatypes

- ž Boolean, PLS_INTEGER etc
- ž Only within PL/SQL
- ž Cannot be used to 'talk' to SQL.
 - **Avoid as function parameters**
- ž Some potential for better performance and/or memory consumption

PL/SQL has its own particular datatypes that are geared towards PROCESSING of data, as opposed to STORAGE. Mostly they are beneficial in terms of performance and/or memory consumption.

Unless you do heavy processing in PL/SQL, the benefits are likely to be small

Plus they cannot be used in embedded SQL statements, and have to be converted to a standard datatype.

PL/SQL Record Types

```
CREATE PROCEDURE TEST
  TYPE typ_rec IS RECORD
    (id NUMBER,
     val VARCHAR2(20));
  t_rec1 typ_rec;
  t_rec2 typ_rec;
BEGIN
  t_rec1.id := 5;
  t_rec1.val := 'TEST';
  t_rec2 := t_rec1;
  t_rec1 := null;
END;
```

You can have type declarations in PL/SQL which are local to the code block. If declared in a package specification, they can be used more widely, but (generally) only within PL/SQL.

You can assign variables of the same type to each other. You can assign NULL to a record type.

Comparison isn't as simple (and don't forget NULLs in record items when you do comparisons).

Sometimes worth creating comparison functions.

SQL Object Type

```
CREATE TYPE typ_rec is OBJECT
  (id NUMBER,
   val VARCHAR2(20));
/
SELECT typ_rec(1, 'test')
FROM DUAL;
```

An SQL Object type has attributes (a cross between variables and columns) and optionally member functions/procedures
These can be used in SQL and PL/SQL

Constructor Methods

```
CREATE TYPE holt AS OBJECT
    (record_type varchar2(1),
      record_column varchar2(10) ,
      CONSTRUCTOR FUNCTION holt
      ( i_record_type in varchar2 default null,
        i_record_column in varchar2 default null)
      return self as result);
```

Objects come with a default constructor method, where you can create an INSTANCE of the object by stating the object name then, in brackets, values for every one of the attributes.

You can create your own constructor functions too. They MUST have the same name as the type, but can differ in their parameters. They can have different numbers of parameters, parameter names, datatypes and default values.

I like constructors that have parameters that default to null, so I don't have to supply them all

Constructor Method Body

```
CREATE OR REPLACE TYPE body holt is
  CONSTRUCTOR FUNCTION holt
    (i_record_type in varchar2 default null,
     i_record_column in varchar2 default null)
    return self as result is
  begin
    self.record_type := i_record_type;
    self.record_column := i_record_column;
    RETURN;
  end holt;
end;
```

To implement the method, you need a BODY, in the same way as a package body. Unlike packages, a type body cannot contain functions not declared in the TYPE specification. [But you can declare private functions / procedures in the declaration part of the member functions.

```
CREATE OR REPLACE TYPE body holt is
  CONSTRUCTOR FUNCTION holt
    (i_record_type in varchar2 default null,
     i_record_column in varchar2 default null)
  return self as result
  is
    v_number number;
    procedure p_test is
    begin
      v_number := 1;
    end;
  begin
    self.record_type := i_record_type;
    self.record_column := i_record_column;
    RETURN;
  end holt;
end;
```

PL/SQL using SQL Type

```
declare
    v_holt    holt;
begin
    v_holt := holt(
        i_record_type => 'a',
        i_record_column => 'B');
end;
/
```

You can use user-defined types as easily as the built-in types.

PL/SQL Collection Types

```
CREATE PROCEDURE TEST  
  TYPE typ_coll_vc10 is TABLE  
    OF VARCHAR2(10) INDEX BY  
    PLS_INTEGER;  
  tc_val typ_coll_vc10;  
BEGIN  
  tc_val (1) := 'TEST' ;  
END;
```

A collection type, in PL/SQL, can be of an ARRAY or TABLE. I find tables more useful as I don't need to consider size when I declare them.

A collection is of a single data type. However that datatype may be a simple SQL datatype [such as NUMBER or VARCHAR2(n)], a user-defined SQL datatype (such as a Spatial co-ordinate)) a PL/SQL type such as PLS_INTEGER or a user-defined PL/SQL record type. You can even define collections of collections, though that would get complicated.

SQL Collection Types

- ǻ SQL Collection types cannot have member functions.
- ǻ They do have functionality equivalent to a default constructor though
- ǻ You can create an object that includes a collection

So you can have a collection like

```
CREATE TYPE tab_varchar_4000 IS TABLE OF VARCHAR2(4000);
```

```
CREATE TYPE strings_coll IS OBJECT (coll tab_varchar_4000);
```

SQL collection as Table

- ž You can select from a collection using the syntax

```
SELECT column_value  
from TABLE(  
    tab_varchar_4000(  
        'a', 'b', 'c'));
```

You are limited to a default constructor if you use the type directly.

FREE

There's a default collection type in the database, granted to public

SYS.DBMS_DEBUG_VC2COLL

```
select * from  
  table(sys.dbms_debug_vc2coll  
        ('a', 'b', 'c'))
```

There's no public synonym for it, so you need to use the SYS schema declaration

Functions returning collections

```
CREATE FUNCTION gen_dates
  (i_start in date, i_end in date)
RETURN tab_varchar_4000 IS
  v_tab tab_varchar_4000 :=
                                tab_varchar_4000();
BEGIN
  v_tab.extend(1 + i_end - i_start );
  FOR i in 0..( i_end - i_start ) LOOP
    v_tab(i+1) := i_start + i;
  END LOOP;
  RETURN v_tab;
END;
```

You need to initialize the collection.
Then extend it to the size necessary.
Load up the rows and return it.

You can then just select from it

```
select * from table(gen_dates(sysdate, sysdate+10));
```

Pipelined functions

```
CREATE FUNCTION gen_dates
  RETURN tab_varchar_4000 PIPELINED IS
  CURSOR c_1 is
    select sysdate dt from dual
    connect by level < 100;
BEGIN
  FOR i in c_1 LOOP
    PIPE ROW (to_char(i.dt));
  END LOOP;
  RETURN;
END;
```

Pipelined functions don't need to assemble the whole dataset into memory before returning it.

You MUST have the RETURN at the end, and it returns nothing.

Wouldn't it be nice....

```
procedure p_test is
Begin
    dbms_output.put_line(' point 1');
    . . .
    dbms_output.put_line(' point 2');
    . . .
End;
```

You've probably seen code like this. Lots of debug statements to report on what the code is doing.
Or rather what the code has done, as dbms_output doesn't get reported until the end.

Can we use PIPE ROW to report on debugging during processing ?

Does not allow DML

```
Function p_test  
return tab_varchar_4000 pipelined is  
Begin  
  pipe row(' point 1' );  
  update....  
  pipe row(' point 2' );  
End;
```

You don't want to try wrapping it all up in autonomous transactions and ugly stuff like that

Must be at the top level

```
Procedure process_records is
  procedure debug (p_text in varchar2)
  is
    begin
      pipe row (p_text);
    end;
  Begin
    . . .
  End;
```

You've probably seen code like this. Lots of debug statements to report on what the code is doing.

Or rather what the code has done, as dbms_output doesn't get reported until the end.

Can we use PIPE ROW to report on debugging during processing ?

The PIPE ROW has to be in the top level (ie can't be delegated to a called subroutine).

```
create function process_records return tab_varchar_4000 pipelined is
  v_str varchar2(10);
  procedure put (p_text in varchar2) is
  begin
    pipe row (p_text);
  end;
begin
  for i in 1 .. 10 loop
    select id into v_str from recs where id = i;
    put ('update ' || i);
  end loop;
  return;
end;
/
```

PLS-00629: PIPE statement cannot be used in non-pipelined functions

Warning – No Data Found

TABLE() functions
complete with a
NO_DATA_FOUND

Warning - NO_DATA_FOUND is a PL/SQL exception
It is NOT an SQL error.

When you select from a function returning a table, if the PL/SQL encounters an unhandled NO_DATA_FOUND exception, it acts like a RETURN. The function ends, and as far as the SQL is concerned, it completed successfully.

```
create function gen_nums
  return tab_varchar_4000 pipelined
is
  cursor c_1 is
    select rownum rn from dual
    connect by level < 10;
  v_num number;
begin
  for i in c_1 loop
    select 1 into v_num from dual where i.rn < 3;
    pipe row (i.rn);
  end loop;
  return;
end;
/
select * from table(gen_nums);
```

Only returns 2 rows.

ALWAYS use a NO_DATA_FOUND exception handler and raise it as a different error.

It gets worse

Non-existent collection elements
also return a no_data_found

```
drop function gen_char;
create function gen_char
  return tab_varchar_4000 pipelined
is
  type tab_val is table of varchar2(10) index by pls_integer;
  t_val tab_val;
begin
  t_val(1) := 'test 1';
  t_val(10) := 'test 10';
  pipe row (t_val(10));
  pipe row (t_val(5));
  pipe row (t_val(1));
  return;
--exception
-- when no_data_found then
--   raise_application_error(-20001,'NDF',false);
end;
/
select * from table(gen_char);
```

Pipelined Functions and PL/SQL Collections

A PL/SQL collection type can also be used in a pipelined function

It implicitly creates three SQL level-types (starting with **SYS_PLSQL_object_id**)

These types can be amended with **ALTER TYPE** (but that's not a good idea)

```
create or replace package test_pipe_pkg is
  type rec_pipe is record (id number, val varchar2(20));
  type tab_pipe is table of rec_pipe;
  function ret_pipe return tab_pipe pipelined;
end;
/
```

```
create or replace package body test_pipe_pkg is
  function ret_pipe return tab_pipe pipelined is
    v_rec rec_pipe;
  begin
    v_rec.id := 1;
    v_rec.val := 'test';
    pipe row (v_rec);
    return;
  end;
end;
/
```

```
select * from table(test_pipe_pkg.ret_pipe);
```

```
select type_name from user_types where type_name like 'SYS_PLSQL%';
alter type "GARY"."SYS_PLSQL_13672_9_1" add attribute gid number;
```

INDEX BY VARCHAR2

Added in 9iR2

Allows a degree of caching in PL/SQL

Useful if you have an expensive calculation that you can reuse (much like the function result cache in 11g).

```
create table t_sub_postcodes (postcode varchar2(4), suburb_name varchar2(20));
insert into t_sub_postcodes values ('2000','Sydney');
insert into t_sub_postcodes values ('3000','Melbourne');
```

```
create or replace procedure demo_cache is
type tab_postcode is table of varchar2(4) index by varchar2(40);
t_postcode tab_postcode;
cursor c_1 is
select decode(mod(rownum,2),1,'Sydney','Melbourne') sub
from dual connect by level <10;
function get_postcode (i_suburb in varchar2) return varchar2 is
v_postcode varchar2(4);
begin
if t_postcode.exists (i_suburb) then
return t_postcode(i_suburb);
else
dbms_output.put_line('Fetching');
select postcode into v_postcode from t_sub_postcodes
where suburb_name = i_suburb;
t_postcode(i_suburb) := v_postcode;
return t_postcode(i_suburb);
end if;
end;
begin
for c_rec in c_1 loop
dbms_output.put_line(get_postcode(c_rec.sub));
end loop;
end;
/
show errors procedure demo_cache

exec demo_cache;
```

SQL collection operators

COLLECT will aggregate up a set of data items into an on-the-fly collection

```
select collect('test')  
from dual connect by level < 10;
```

You can CAST that collection to your own collection type, if compatible.

Beware of CHAR / VARCHAR2 incompatibilities

```
create table test (val varchar2(10));
```

```
insert into test  
select sysdate+level from dual  
connect by level < 10;
```

```
select cast(collect(val) as tab_varchar_4000) a from test;
```

Like any Aggregate Function

DISTINCT

```
select collect(distinct  
              to_char(sysdate))  
from dual connect by level < 5;
```

GROUP BY

```
select cast(collect(object_name) as  
          tab_varchar_4000) t  
from user_objects  
group by object_type;
```

Collections and Sets

Definition :

A collection is a set if it doesn't have duplicates

```
create type mset is table of varchar2(4000);  
/
```

```
create type mset_mset is table of mset;  
/
```

SET testing

```
select 1 a from dual
where mset('a', 'b', 'a') is a set;
no rows selected
```

```
select 1 a from dual
where mset('a', 'b') is a set;
```

A

1

```
create type mset is table of varchar2(4000);
/
```

```
create type mset_mset is table of mset;
/
```

MULTISET operators

Turn a collection into a set.

```
select
  mset('z', 'a', 'a', 'c', 'd', 'c')
multiset union distinct
  mset()
from dual;
```

There are also INTERSECT and EXCEPT
(not MINUS)

SET comparison

```
select 1 from dual  
where 'a' member mset('a','b');  
1
```

```
select 1 from dual  
where mset('a') submultiset  
      mset('a','b');  
1
```

```
select 1 a from dual  
where mset('a','c') submultiset  
      mset('a','b');  
no rows selected
```

First one checks if an ELEMENT is in the set

Second/third ones check if a set is contained within a set

Collection comparison

You can compare simple collections in PL/SQL with an equals operator

```
declare
  x1 mset := mset(' a', ' b');
  x2 mset := mset(' b', ' a');
begin
  if x1 = x2 then
    dbms_output.put_line(' Match');
  end if;
end;
```

```
set serveroutput on
declare
  x1 tab_varchar_4000;
  x2 tab_varchar_4000;
begin
  select cast(collect(to_char(sysdate)) as tab_varchar_4000) into x1
  from dual connect by level < 10;
  --
  select cast(collect(to_char(sysdate)) as tab_varchar_4000) into x2
  from dual connect by level < 5;
  --
  if x1 = x2 then
    dbms_output.put_line('Match');
  --elsif x1 < x2 then
  -- dbms_output.put_line('Less');
  else
    dbms_output.put_line('No Match');
  end if;
end;
.
/
```

Multiset operators in PL/SQL

```
declare
    x1 mset := mset(' a', ' b' );
    x2 mset := mset(' b' );
    x3 mset;
begin
    x3 := x1 multiset except x2;
    dbms_output.put_line(x3.count
        || ' . ' || x3(1));
end;
```

Also works with locally defined collections

```
declare
    TYPE lset is table of varchar2(30);
    x1 lset := lset('a','b');
    x2 lset := lset('b');
    x3 lset;
begin
    x3 := x1 multiset except x2;
    dbms_output.put_line(x3.count||'. '||x3(1));
end;
```

But not complex ones (SQL or PL/SQL)

```
declare
    TYPE lrc is record (id number, val varchar2(10));
    TYPE lset is table of lrc;
    x1 lset;
    x2 lset;
    x3 lset;
    function x (p_id in number, p_val in varchar2) return lrc is
        v_x lrc;
    begin
        v_x.id := p_id;
        v_x.val := p_val;
        return v_x;
    end;
begin
    x1 := lset(x(1,'a'), x(2,'b'));
    x2 := lset(x(2,'b'));
    x3 := x1 multiset except x2;
    dbms_output.put_line(x3.count);
end;
```

Final Thought

For complex logic
SQL*Plus embedded
in shell scripts is

NOT

a good idea

SQL*Plus is a poor tool for procedural logic (loops and conditional flows)

Embedding calls to sqlplus in a shell script is not a good idea.

You generally won't have a persistent connection, so will be continually logging on and off the database.

You can't maintain transactions

There is poor error/exception handling as you can really only return a generic error/warning to the shell script.

Use Perl (which gets installed with the databases, except XE) or Python
or....PL/SQL