




# Lesson 5

## New PL/SQL Features

ANSI CASE Support  
Record-Based DML  
New Bulk Collect  
Associative Arrays  
Multi-Level Collections

Pipelined Functions  
Native Compilation  
LOB Support  
UTL\_FILE  
DBMS\_METADATA

**SKILLBUILDERS**



5.2

## New PL/SQL features

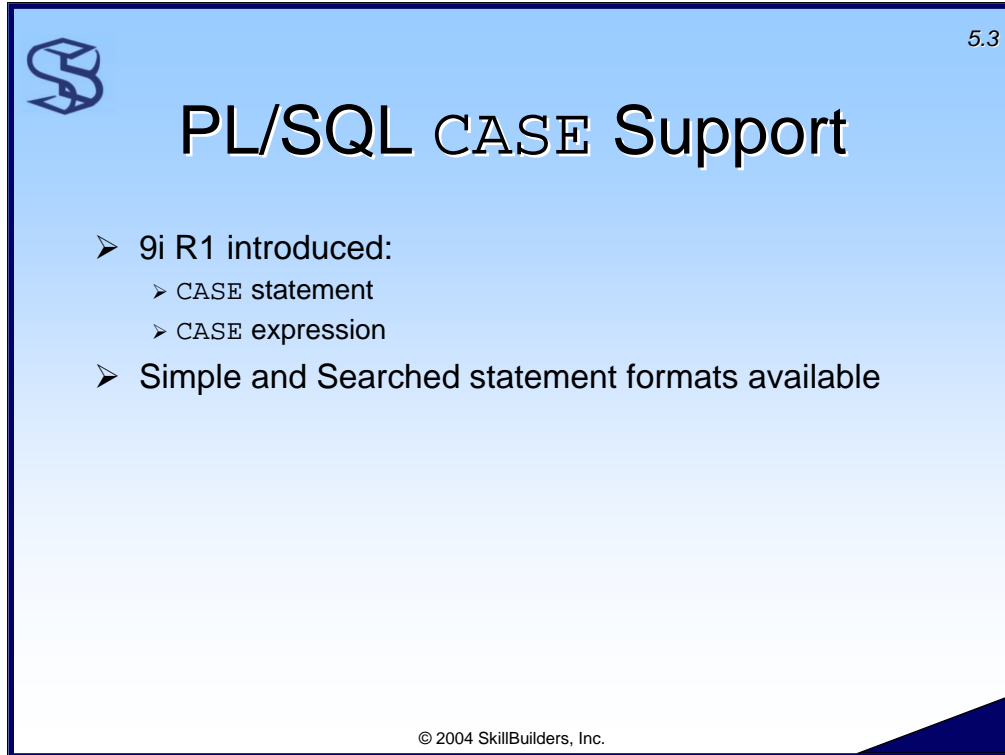
- Oracle9i adds:
  - ANSI compliant `CASE` statement
  - Record-Based DML
  - Bulk Collect on DML Returning clause
  - `VARCHAR` index on Arrays
  - Multi-level collections
  - Pipelined Functions
  - Native Compilation
  - LOB support enhancements
  - Metadata Access
  - `UTL_FILE` enhancements
- Let's look at each in turn

© 2004 SkillBuilders, Inc.


#### Additional Notes:

In addition to the new features listed above, Oracle9i also adds:

- PL/SQL and SQL now uses a common SQL parser so any valid SQL can now be used in PL/SQL.



5.3

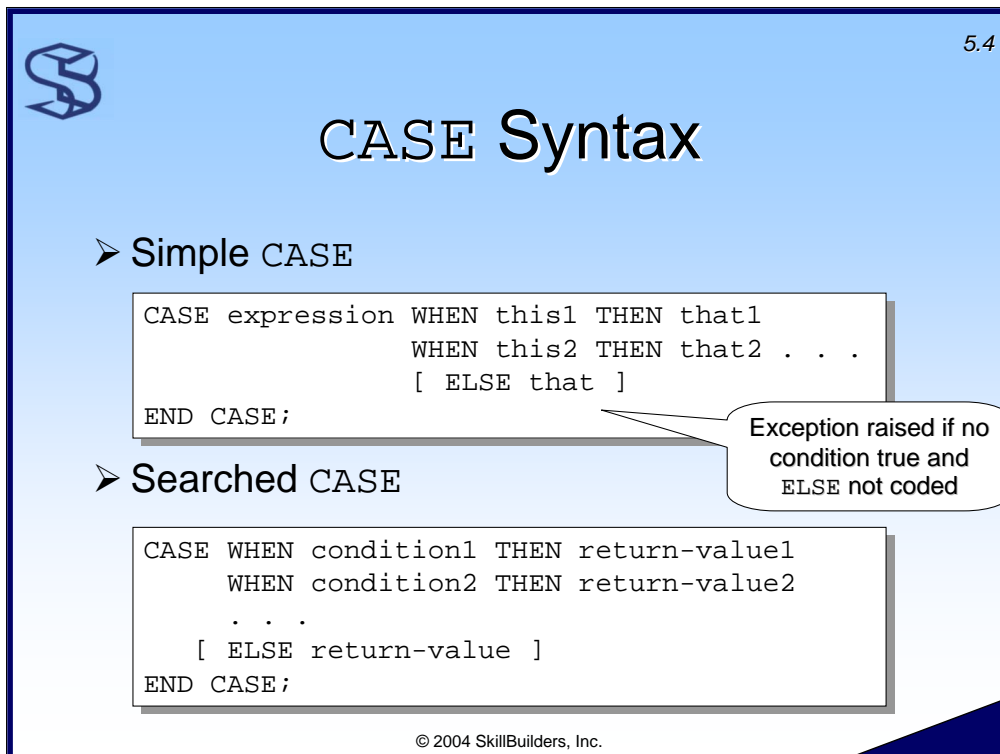


## PL/SQL CASE Support

- 9i R1 introduced:
  - CASE statement
  - CASE expression
- Simple and Searched statement formats available

© 2004 SkillBuilders, Inc.

Oracle9i Release 1 introduced a PL/SQL `CASE` statement (Oracle8i added `CASE` for SQL statements). `CASE` can also be coded as an expression if it is desired to populate a variable with the result of the `CASE` logic. Examples will follow.



5.4

## CASE Syntax

➤ Simple CASE

```
CASE expression WHEN this1 THEN that1
                 WHEN this2 THEN that2 . . .
                 [ ELSE that ]
END CASE;
```

Exception raised if no condition true and ELSE not coded

➤ Searched CASE

```
CASE WHEN condition1 THEN return-value1
      WHEN condition2 THEN return-value2
      . . .
      [ ELSE return-value ]
END CASE;
```

© 2004 SkillBuilders, Inc.

Oracle supports two flavors of `CASE`, simple and searched.

The simple case expression tests for an equal condition on the supplied value or expression. The first `WHEN` value that is equal causes Oracle to return the corresponding `THEN` value. If none of the `WHEN` values match the supplied expression, the `ELSE` value is returned. If the `ELSE` is not coded, `NULL` is returned.

The searched case (as seen in the previous example) allows multiple comparison expressions (`<`, `>`, `<=`, `>=`, `BETWEEN`, `LIKE`, `IN`, `IS NULL`, etc.). The first `TRUE` expression causes Oracle to return the corresponding `THEN` value. If none of the `WHEN` values match the supplied expression, the `ELSE` value is returned. If the `ELSE` is not coded, `NULL` is returned.


In all versions of the `CASE` statement the `WHEN` clause can appear any number of times. The `WHEN` clauses are evaluated sequentially. The 1<sup>st</sup> `TRUE` `WHEN` causes the associated statement(s) to be executed; The `CASE` statement then ends (execution continues after the `END CASE` clause). If none of the `WHEN` expressions is true the `ELSE` statement (if any) will execute.

Notes continue on the next page...

The `CASE` statement raises a `CASE_NOT_FOUND` exception if an `ELSE` clause is not provided and none of the `WHEN`'s are `TRUE`.

Only one `THEN` statement (or `ELSE` statement) is executed for each `CASE` statement. There is no "fall-through" as in the C language 'switch' statement.

`CASE` is limited to 128 `WHEN/THEN` pairs (255 total values). This limit can be overcome by nesting `CASE` within `CASE`.

5.6

## Simple CASE

```
<<salary_test>>
CASE v_sal
  WHEN 12 THEN
    dbms_output.put_line('Salary is '||v_sal);
    v_sal := v_sal * 1.2 ;
    dbms_output.put_line('Salary is '||v_sal);
  WHEN 14 THEN
    dbms_output.put_line('Salary is '||v_sal);
    v_sal := v_sal * 1.15 ;
    dbms_output.put_line('Salary is '||v_sal);
  ELSE
    v_sal := v_sal * 1.1 ;
END CASE salary_test;
```

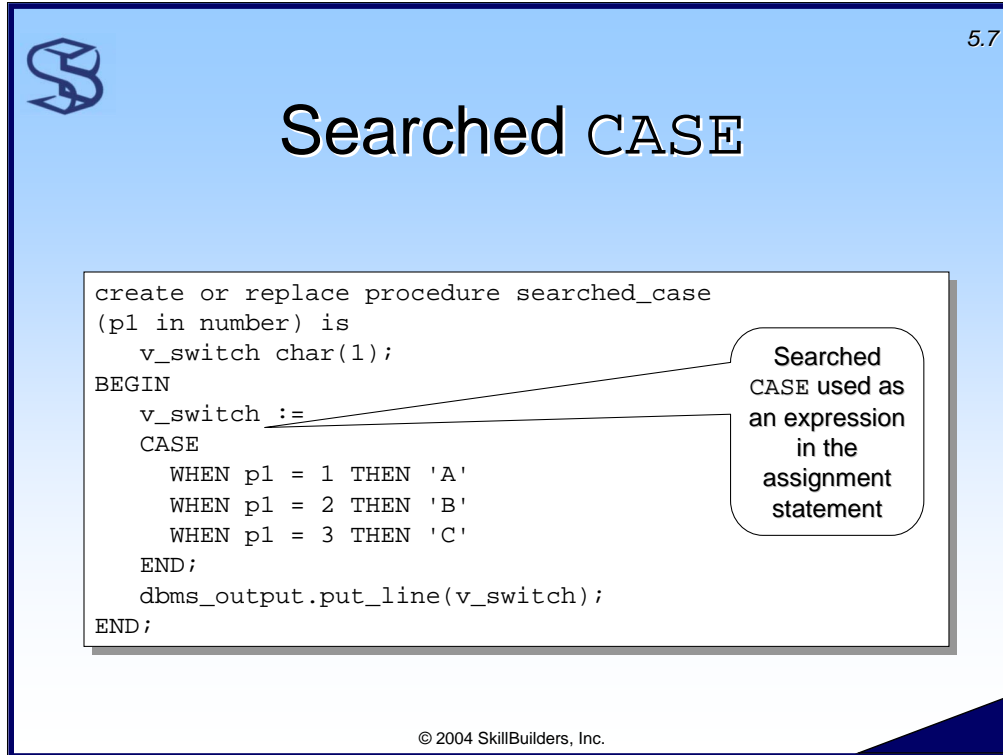
© 2004 SkillBuilders, Inc.

Here is an example of a simple CASE statement. Notes:

- The label is optional but provides good documentation.
- Each THEN can have any number of statements, each terminated with a semicolon.
- Only the 1st TRUE THEN is executed. Control is transferred to the END CASE after the 1st TRUE THEN is executed.
- If the ELSE is not provided and none of the THEN's are TRUE, a CASE\_NOT\_FOUND exception is raised and control is automatically transferred to the EXCEPTION block, if coded.

Restriction: The case-operand and the when-operands can be any datatype except BLOB, BFILE, an object type, a PL/SQL record, an index-by-table, a varray, or a nested table.

See supplied script `case_plsql_examples` for working examples of PL/SQL CASE.



The slide features a blue background with a white box containing PL/SQL code. A callout box points to the assignment statement line in the code. The Oracle logo is in the top left corner, and the slide number '5.7' is in the top right corner.

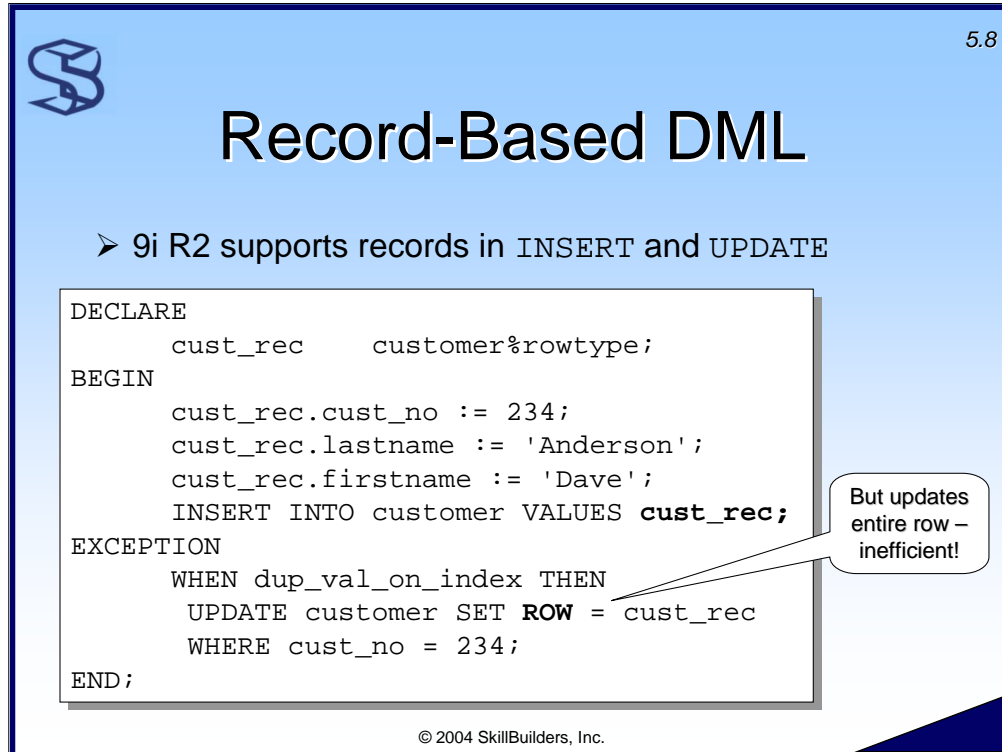
```
create or replace procedure searched_case
(p1 in number) is
  v_switch char(1);
BEGIN
  v_switch :=
  CASE
    WHEN p1 = 1 THEN 'A'
    WHEN p1 = 2 THEN 'B'
    WHEN p1 = 3 THEN 'C'
  END;
  dbms_output.put_line(v_switch);
END;
```

Searched CASE used as an expression in the assignment statement

© 2004 SkillBuilders, Inc.

Here is an example of a Searched CASE expression. Note the absence of a CASE selector and the use of Boolean expressions in each WHEN clause.

See supplied script `case_plsql_examples.sql` for a working example of this CASE expression.



The slide features a blue background with a white box containing PL/SQL code. A callout bubble points to the `UPDATE` statement. The slide number '5.8' is in the top right corner, and a logo is in the top left.

```
DECLARE
    cust_rec    customer%rowtype;
BEGIN
    cust_rec.cust_no := 234;
    cust_rec.lastname := 'Anderson';
    cust_rec.firstname := 'Dave';
    INSERT INTO customer VALUES cust_rec;
EXCEPTION
    WHEN dup_val_on_index THEN
        UPDATE customer SET ROW = cust_rec
        WHERE cust_no = 234;
END;
```

But updates entire row – inefficient!

© 2004 SkillBuilders, Inc.


Oracle9i R2 supports PL/SQL records in the `INSERT` and `UPDATE` DML statements.

Note that in the `INSERT`, “`cust_rec`” is NOT enclosed in parenthesis. This is required.

Note the use of the new keyword “`ROW`” in the `UPDATE` statement. This allows us to update the entire row. Unfortunately, it is not possible (yet?) to update a subset of a row using the `ROW` keyword. This is inefficient, especially because primary and foreign keys may be unnecessarily updated, causing unnecessary referential integrity checks.

You can define your own PL/SQL record (as opposed to using `%ROWTYPE`), but it must be completely compatible with the table row. I.e. You cannot define only a subset of columns.

See the supplied script `record_based_dml.sql` for working code examples of record-based DML.

5.9

## BULK COLLECT

```
SQL> DECLARE
2   Type cust_rec_t is RECORD
3   (cust_no number, lastname varchar2(25),
4     firstname varchar2(25) );
5   Type cust_tab_t is table of cust_rec_t;
6   cust_tab cust_tab_t;
7 BEGIN
8   UPDATE customer SET area_code = '917'
9   WHERE area_code IS NULL
10  RETURNING cust_no, lastname, firstname
11  BULK COLLECT INTO cust_tab;
12  dbms_output.put_line(cust_tab(1).lastname);
13 END;
14 /
Anderson


PL/SQL procedure successfully completed.
```

© 2004 SkillBuilders, Inc.

Another new supported use of PL/SQL records is in the `BULK COLLECT INTO` clause. This is an extension to the bulk select/fetch and `FORALL` statement introduced in earlier releases of Oracle..

We see in this example that we return (using bulk collect) all effected customers (customers who are updated) into the collection "cust\_tab" which is based on the record "cust\_rec\_t". Note that it is still required to code all the individual column names in the `RETURNING` clause, even if you desire to return all columns. "`RETURNING *`" is not supported.

See the supplied script `bulk_collect.sql` for working code example of bulk collect.



5.10

## Associative Arrays...

- Formerly called Index-By tables
- An unbounded array of variables
- Use any number or character string as subscript
- Useful for lookups
- Can be passed as Procedure or Function parameter
- Only define-able at PL/SQL level

```
TYPE table_type_name IS
  TABLE OF datatype [NOT NULL]
  INDEX BY
  [BINARY_INTEGER | VARCHAR2(size)];
```

© 2004 SkillBuilders, Inc.

Associative Arrays are a PL/SQL array, used to store lists of data in a PL/SQL program. This might be helpful for storing data that is repeatedly scanned – eliminating repetitive access to a database table. It can also be a useful structure for passing sets of data between PL/SQL programs. Associative Arrays were introduced with Oracle9i Release 2.

Associative arrays are arrays of variables where the variable can be a scalar type, a variable defined with %TYPE or a record defined with %ROWTYPE.

Associative arrays are unbounded, meaning that they have is no limit to the number of elements in the array. (actually, the limit is -2,147,483,647 to +2,147,483,647, or 4.3 billion rows. However, we consider them to be unbounded because you'll run out of memory before you'll ever reach the limit.)

Associate arrays are also considered sparse, in that they do not require a sequential number of rows. I.e. there can be gaps between element 1 and the second element.

Associative arrays require an index. The index can be BINARY\_INTEGER (a number) or, with Oracle9i, a VARCHAR2 field.

The elements in a PL/SQL array are not in any particular order and are not necessarily stored contiguously in memory. The keys used for a PL/SQL table do not have to be sequential and can be an expression as well as a constant or variable.

5.11

## ...Associative Arrays...

```

DECLARE
  TYPE sales_type IS TABLE OF NUMBER(17,2)
                      INDEX BY VARCHAR2(12);
  sales_region      SALES_TYPE;
  sales_figure      NUMBER;
  first_index_value VARCHAR2(12);
  last_index_value  VARCHAR2(12);
BEGIN
  sales_region('West') := 200000.34;
  sales_region('East') := 100000.99;
  sales_region('MidWest') := 30000.45;
  sales_figure := sales_region('East');
  first_index_value := sales_region.FIRST;
  last_index_value := sales_region.LAST;
  sales_figure := sales_region(sales_region.LAST);
  sales_region('West') := 150000.55;
END;

```

Assign value to element of array

Access index values

Update value

© 2004 SkillBuilders, Inc.

This example shows the usage of an associative array using a VARCHAR2 index.

The following statements declare the associative array type and then assign the type to a variable.

```

TYPE sales_type IS TABLE OF NUMBER(17,2)
                  INDEX BY VARCHAR2(12);
sales_region sales_type;

```

To assign a value to the array, specify a character string index.

```
sales_region('West') := 200000.34;
```

Notes continue on the next page...

There are a number of collection methods that can be used with associative arrays.

The `FIRST` and `LAST` collection methods can be used to access the first and last index values or the first and last actual values in the associative array.

```
first_index_value:=sales_region.FIRST;-- value will be East
last_index_value:= sales_region.LAST; -- value will be West
sales_figure:=sales_region(sales_region.LAST); -- 200000.34
```

`COUNT` is another collection method that can be used with an associative array. It returns the number of index items in the array.

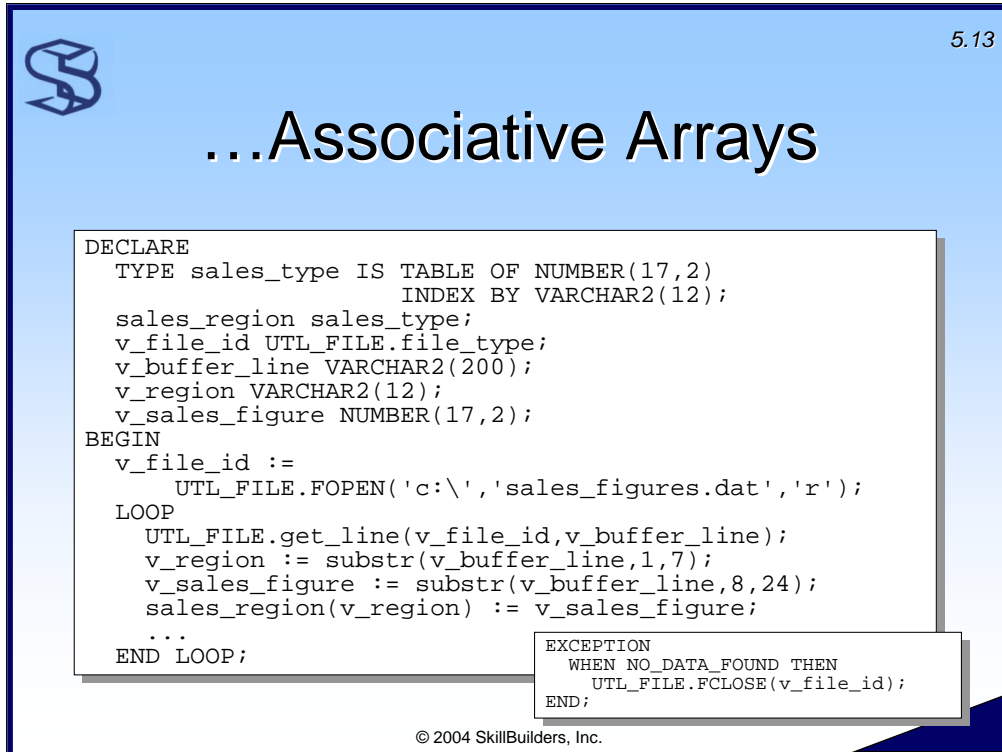
```
sales_region.COUNT
```

The `NEXT` and `PRIOR` collection methods allow you to traverse through the array.

To replace a value for an index value that has already been used, execute an assignment statement reusing the index value.

```
sales_region('West') := 150000.55;
```

See the supplied script `ASSOC_ARRAY1.SQL` for a working example of this code.



5.13

## ...Associative Arrays

```
DECLARE
  TYPE sales_type IS TABLE OF NUMBER(17,2)
                      INDEX BY VARCHAR2(12);
  sales_region sales_type;
  v_file_id UTL_FILE.file_type;
  v_buffer_line VARCHAR2(200);
  v_region VARCHAR2(12);
  v_sales_figure NUMBER(17,2);
BEGIN
  v_file_id :=
    UTL_FILE.FOPEN('c:\', 'sales_figures.dat', 'r');
  LOOP
    UTL_FILE.get_line(v_file_id, v_buffer_line);
    v_region := substr(v_buffer_line, 1, 7);
    v_sales_figure := substr(v_buffer_line, 8, 24);
    sales_region(v_region) := v_sales_figure;
    ...
  END LOOP;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    UTL_FILE.FCLOSE(v_file_id);
END;
```

© 2004 SkillBuilders, Inc.

This example demonstrates reading data from an operating system file into an associative array. The array is declared and assigned to a variable just as in the example on the previous page. One line at a time is retrieved from the file and is assigned to the array.

To retrieve a line from the file execute the `get_line` procedure.

```
UTL_FILE.get_line(v_file_id, v_buffer_line);
```

The data in the file is in a fixed format.

```
West    34000.30
```

```
East    28900.15
```

```
MidWest44890.25
```

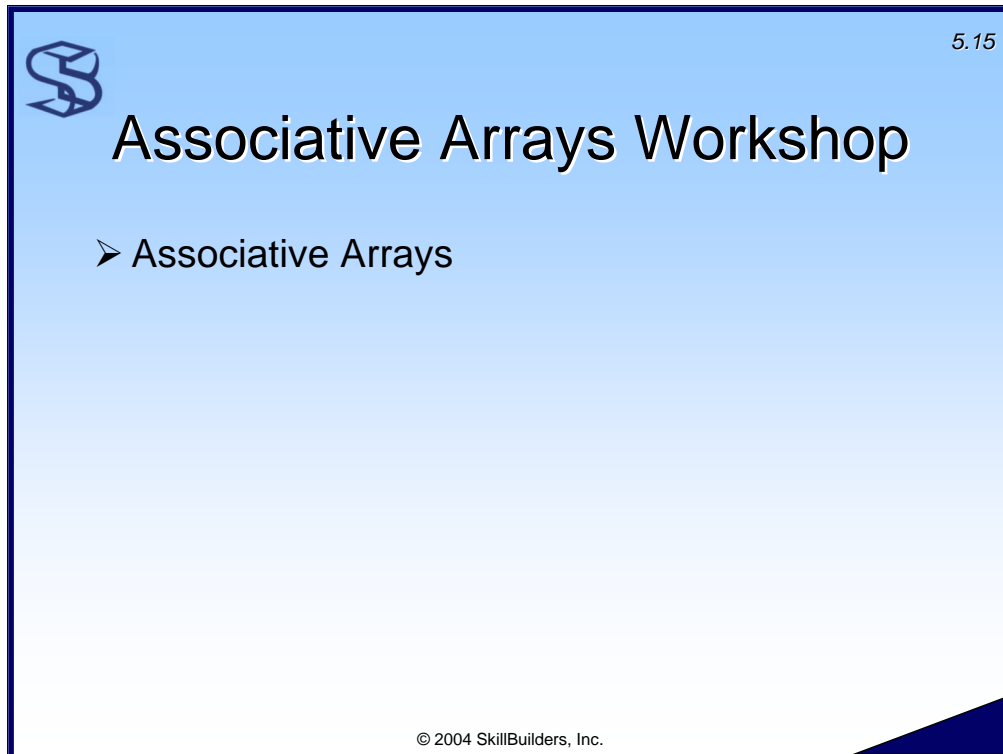
Notes continue on the next page...

To separate the region value from the sales figure value that was read from the file, the `SUBSTR` function can be used. Once separated, the individual values can be used to assign a value to the array.

```
v_region := substr(v_buffer_line,1,7);  
v_sales_figure := substr(v_buffer_line,8,24);  
sales_region(v_region) := v_sales_figure;
```

The `get_line` procedure raises a `no_data_found` exception when there is no data to be accessed, at which point the file can be closed.

See the supplied script `ASSOC_ARRAY2.SQL` for a working example of this code. The external operating system file is also supplied (`sales_figures.dat`).



### Workshop - Associate Arrays

1. Complete the code supplied on the next page. It can also be found in the supplied file `ASSOC_ARRAY_WS_TEMPLATE.SQL`. Use an associative array to store information about products and how many times they have been ordered. Use the product name as the index into the array. For each product name store a count of the number of times that product has been ordered. Include products that have not been ordered with a count of 0.

To verify the results, display to the screen the number of items in the array, the first index value, and the last index value along with all product names and their order counts.

Workshop continues on the next page...

```
DECLARE
/* define the associative array TYPE */

/* declare a variable of type table */

v_product_name product.description%TYPE;
v_product_count NUMBER(20);
v_index VARCHAR2(75);

CURSOR read_counts IS
SELECT description, count(ord_item.product_id)
FROM product, ord_item
WHERE product.product_id = ord_item.product_id (+)
GROUP BY description;

BEGIN
OPEN read_counts;
LOOP
FETCH read_counts INTO v_product_name, v_product_count;
EXIT WHEN read_counts%NOTFOUND;

/* Provide the code to assign the product_count to an element of the array
*/

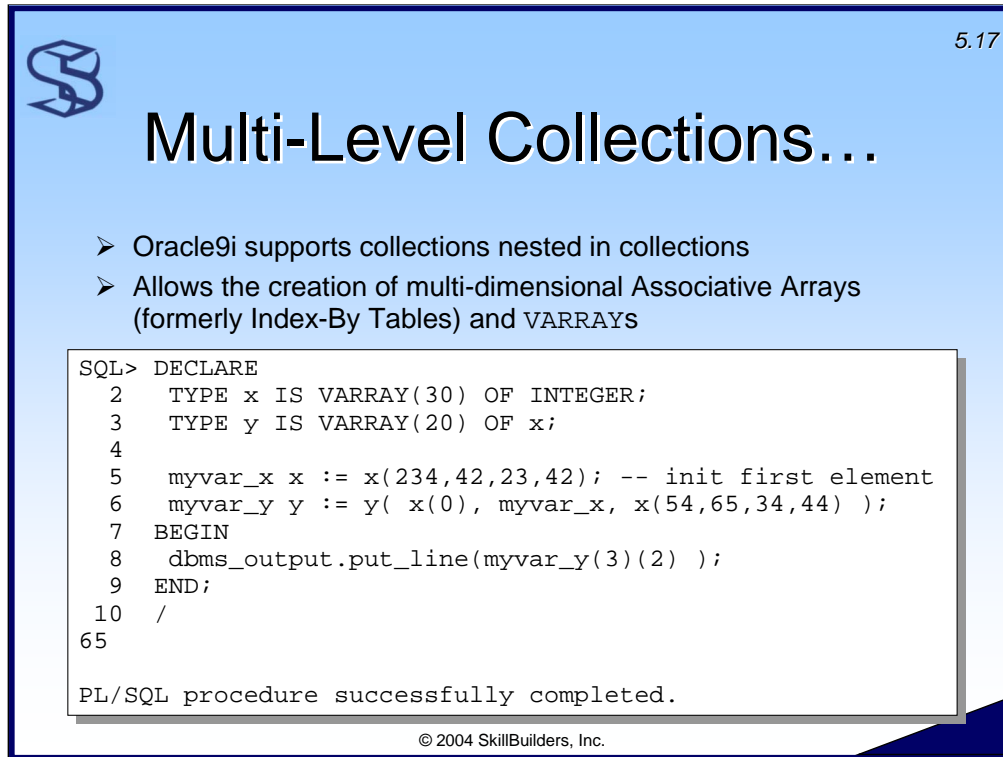
END LOOP;
CLOSE read_counts;

/* Use put_line to display:
number of items in the array,
the first index value,
the last index value
all product names their order counts.
*/

DBMS_OUTPUT.PUT_LINE ( );
DBMS_OUTPUT.PUT_LINE ( );
DBMS_OUTPUT.PUT_LINE ( );

/* get subscript of first element */
v_index := product_count.FIRST;

WHILE v_index IS NOT NULL LOOP
DBMS_OUTPUT.PUT_LINE
(v_index||' '||product_count(v_index));
/*get subscript of next element */
v_index := product_count.NEXT(v_index);
END LOOP;
END;
/
```



5.17

## Multi-Level Collections...

- Oracle9i supports collections nested in collections
- Allows the creation of multi-dimensional Associative Arrays (formerly Index-By Tables) and VARRAYS

```

SQL> DECLARE
  2   TYPE x IS VARRAY(30) OF INTEGER;
  3   TYPE y IS VARRAY(20) OF x;
  4
  5   myvar_x x := x(234,42,23,42); -- init first element
  6   myvar_y y := y( x(0), myvar_x, x(54,65,34,44) );
  7 BEGIN
  8   dbms_output.put_line(myvar_y(3)(2) );
  9 END;
 10 /
65

PL/SQL procedure successfully completed.

```

© 2004 SkillBuilders, Inc.

Oracle9i supports multi-level (multi-dimensional) arrays through the support of collections within collections.

In this example, “y” is now an array of 20 “x”.

To use the multi-dimensional `varray` simply define a variable of that type. You must initialize each value of the `varray` before you can access them.

The following statement declares `myvar` and initializes one `y` element containing only null `x` elements:

```
myvar y := y();
```


The following declares and initializes one `y` element containing one `x` element which only contains nulls:

```
myvar y := y( x(null, null, null) );
```

To access `myvar` as a two dimensional `varray` use the following syntax:

```
myvar(1)(1) := 123;
```

See the provided script ‘`multi_level_collections.sql`’ for a working example.



5.18

## ...Multi-Level Collections

```
DECLARE
  TYPE district_type is VARRAY(2) of NUMBER(17,2);
  TYPE region_type is VARRAY(3) of district_type;
  district_sales_figures district_type := district_type(0,0);
  region_sales_figures region_type :=
    region_type(district_sales_figures, district_sales_figures,
               district_sales_figures);
BEGIN
  region_sales_figures(1)(1) := 20000.00;
  region_sales_figures(1)(2) := 10000.00;
  region_sales_figures(2)(1) := 40000.00;
  region_sales_figures(2)(2) := 17000.00;
  ...
end;
```

© 2004 SkillBuilders, Inc.

This example shows the usage of a multi-dimensional array. The example is a portion of code created to support the sales organizations within a company. The sales organizations are broken into 3 regions that contain 2 districts each.

The following statements declare the array types and then assign the types to variables.

```
TYPE district_type is VARRAY(2) of NUMBER(17,2);
TYPE region_type is VARRAY(3) of district_type;


district_sales_figures district_type := district_type(0,0);
region_sales_figures region_type :=
  region_type(district_sales_figures,
              district_sales_figures,
              district_sales_figures);
```

Note: Collections must be initialized before they can be used. In this example the arrays are initialized with values of 0;

The following statement assigns a sales figure of 40000.00 to region 2, district 1.

```
region_sales_figures(2)(1) := 40000.00;
```

See the supplied script `MULTI-LEVEL_COLLECTIONS2.SQL` for a working example of this code.



5.19

## Pipelined Functions


- Extraction, Transformation & Load (ETL) feature
- Extension to 8i *Table Functions* feature
  - Function is *row source*
- Let's review Table Functions
  - Then we will look at Pipelined Table Functions

© 2004 SkillBuilders, Inc.

Oracle9i provided an enhancement to table functions called “pipelined functions.”

Table functions were introduced with Oracle8i and allow a function to “act like a table”. I.e. The function can be coded in the `FROM` clause of a `SELECT` statement.

We will first review table functions, then turn to pipelining.



5.20

## Review: 8i Table Functions

- Oracle8i feature
- Allows function to “act like a table”
- The function is a row source – instead of a table
- Use in FROM clause of SQL query

```
select x.column_value as month
from TABLE( CAST (month_generator(12)
                 AS sqlMONTH_TABLEtype) ) x
```

MONTH
-----
23-OCT-02
23-SEP-02
23-AUG-02
23-JUL-02
...
23-DEC-01

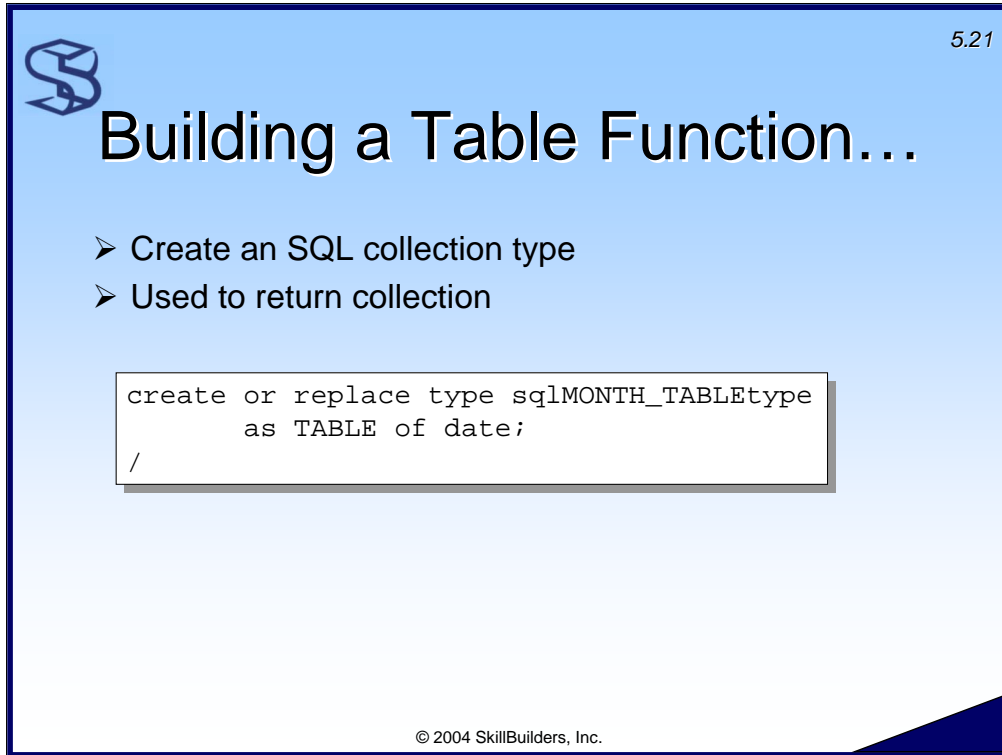
© 2004 SkillBuilders, Inc.

Table functions are an Oracle8i feature that allows a function to be called in the FROM clause of a SELECT statement. The table function returns a result set, in the form of a SQL collection type.

The purpose of any table function is to return a set of rows to the caller – thus acting like a normal Oracle table. In this example, we see a table function called `month_generator` called in the FROM clause of a SELECT statement. The `month_generator` function will (as we will see in subsequent examples) return “n” number of previous months.

### Supplemental Notes:

- `COLUMN_VALUE` is a reserved word and is a generic reference to the unnamed column produced by the Table Function. `SELECT *` would also work.
- The `TABLE` keyword is the required syntax when coding a table function in the FROM clause.
- `CAST` is required in Oracle8i to tell Oracle the datatype of the returned values. It is not required in Oracle9i.



5.21

## Building a Table Function...

- Create an SQL collection type
- Used to return collection

```
create or replace type sqlMONTH_TABLEtype
  as TABLE of date;
/
```

© 2004 SkillBuilders, Inc.

A table function returns a set of rows in the form of a collection. Therefore, we first need to define an SQL type of type `TABLE`, as shown above. This definition is permanent; like the creation of any Oracle object with the `CREATE` command, this type is now permanently defined to this schema in the database. It can be removed with the `DROP TYPE` command.

5.22

## ...Building a Table Function

```
create or replace function month_generator
(p_num_months in number)
RETURN sqlMONTH_TABLEtype
AS
    month_table      sqlMONTH_TABLEtype
                    := sqlMONTH_TABLEtype();
BEGIN
    for i in 1..p_num_months loop
        month_table.extend(1);
        month_table(i) :=
            add_months(sysdate, -i);
    end loop;
    return(month_table);
END;
/
```

Return type

Define PL/SQL collection

Populate collection

Return collection


© 2004 SkillBuilders, Inc.

This example shows the definition of the Table Function called `month_generator`.

### Notables:

- The function returns a collection. The collection “`sqlMONTH_TABLEtype`” was defined in the previous example.
- The PL/SQL collection (`month_table`) is populated with dates in the `LOOP`.
- The `RETURN` instruction passes the collection back to the calling query.

See supplied script `TABLEFUNC1.SQL` for a working example.



5.23

## Using a Table Function...

- Problem:
  - Create an average sales report
  - Include ALL the previous “n” months, even if zero sales
- Solution:
  - Create a “virtual table” with a Table Function
    - Contains all months
  - Code an outer join to the virtual table

© 2004 SkillBuilders, Inc.

Let's present a problem that we can solve with a table function...

Perhaps you are asked to create a monthly average sales report for all of the previous *n* months, where *n* is *any* number of months. If you simply `GROUP BY` the `ORD` table, you will miss months for which there are not sales:

```
SQL> select to_Char(order_date, 'mm/yyyy'), avg(total_order_price)
  2  from ord
  3  group by to_Char(order_date, 'mm/yyyy');
```

TO_CHAR	AVG(TOTAL_ORDER_PRICE)
02/1999	35.64
03/1999	10.95
04/1999	22.95
06/1999	110.95
11/2002	23.4625
12/1999	55.95

The table function `month_generator` will help. We will use it to create all of the months we need to report on, then `OUTER JOIN` to it.



5.24

## ...Using a Table Function

- Outer join to the table produced by Table Function:

```
SELECT to_Char(x.column_value, 'mm/yyyy') ,
       nvl(avg(total_order_price),0) as avg_Sales
FROM TABLE( month_generator(12) ) x
   LEFT OUTER JOIN ord
ON to_Char(x.column_value, 'mm/yyyy') =
   to_Char(order_date, 'mm/yyyy')
GROUP BY to_Char(x.column_value, 'mm/yyyy') ;
```


© 2004 SkillBuilders, Inc.

Here is the solution. Note the outer join to the table created by the Table Function "month\_generator".

The example above uses the new Oracle9i ANSI outer join. An equivalent 8i solution would be:

```
select to_Char(x.column_value, 'mm/yyyy') ,
       nvl(avg(total_order_price),0) as avg_Sales
from TABLE( month_generator(12) ) x, ord
where to_Char(x.column_value, 'mm/yyyy') =
       to_Char(order_date(+), 'mm/yyyy')
group by to_Char(x.column_value, 'mm/yyyy') ;
```

See supplied script TABLEFUNC1.SQL for a working example.


5.25

## Pipelined Functions

- 9i feature
- Return rows before function completes
- Caller can resume; start consuming rows
  - Can be more efficient
- Intermediate collection storage not required
  - Uses less memory

© 2004 SkillBuilders, Inc.

Oracle9i provided an enhancement to table functions called “pipelined functions.” The purpose of pipelining is efficiency. The pipelined function starts returning rows (piping rows) back to the caller before the function even completes. (Remember that in the previous example, the function passed back a complete, populated collection on the `RETURN` instruction.) In addition to passing rows back to the caller as soon as possible, the pipelined function does not require potentially large memory area to hold a populated collection – as the previous example did.



5.26

## Pipelined Example

```
create or replace function month_generator
(p_num_months in number)
RETURN sqlMONTH_TABLEtype
PIPELINED
AS
  month_table      sqlMONTH_TABLEtype
                  := sqlMONTH_TABLEtype();
BEGIN
  for i in 1..p_num_months loop
    PIPE ROW ( add_months(sysdate, -i) );
  end loop;
  return;
END;
/
```

© 2004 SkillBuilders, Inc.


Here is an example of a pipelined function. This function performs the same function as the previous `month_generator` function, but is more efficient because:

1. the `PIPE ROW` instruction “pipes” rows back to the caller immediately – before the function fully completes, and
2. the function requires less memory because it does not have to fully populate the collection and pass the collection back to the caller, it simply passes the rows back as it creates them (with the `PIPE ROW` statement).

Notables:

- Use the `PIPELINED` keyword in the function header to define a pipelined function.
- Use the `PIPE ROW` statement to send the rows back to the caller.
- Do NOT code any return value on the `RETURN` instruction.

See supplied script `TABLEFUNC2.SQL` for a working example.



5.27

## Test Data Generator...

```

create or replace function ord_generator
  (p_num_rows in number)
  RETURN ord_table
AS
  v_ord_table      ord_TABLE := ord_table();
BEGIN
  for i in 1..p_num_rows loop
    v_ord_table.EXTEND;
    v_ord_table(i) := ( ord_type(
      i ,i ,sysdate - i ,0 ,sysdate - i
      ,null , 'CA' ,1 ,null ,null ));
  end loop;
  return v_ord_table;
END;
/

```

© 2004 SkillBuilders, Inc.

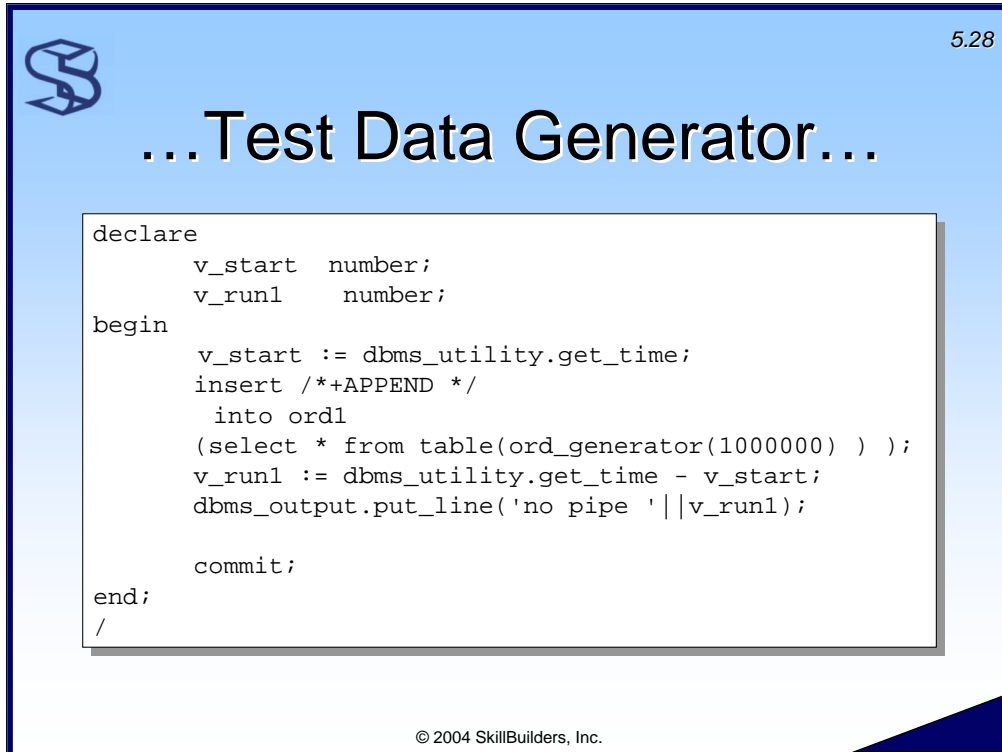
This table function, called `ord_generator`, generates test data for my ORD table. It requires these structures:

```

create or replace type ord_type as object
  ( ORDER_NO NUMBER,
    CUST_NO NUMBER,
    ORDER_DATE DATE ,
    TOTAL_ORDER_PRICE NUMBER(7,2),
    DELIVER_DATE DATE,
    DELIVER_TIME VARCHAR2(7),
    PAYMENT_METHOD VARCHAR2(2) ,
    EMP_NO NUMBER(3,0),
    DELIVER_NAME VARCHAR2(35),
    GIFT_MESSAGE VARCHAR2(100) );
/
create or replace type ord_table as table of ord_type;
/

```

See supplied script `TABLEFUNC3.SQL` for a working example of this code.



5.28

## ...Test Data Generator...


```
declare
    v_start  number;
    v_run1   number;
begin
    v_start := dbms_utility.get_time;
    insert /*+APPEND */
        into ord1
        (select * from table(ord_generator(1000000) ) );
    v_run1 := dbms_utility.get_time - v_start;
    dbms_output.put_line('no pipe '||v_run1);

    commit;
end;
/
```

© 2004 SkillBuilders, Inc.

However, it is very likely that this function will fail on a memory error because it attempts to generate one million rows, and the `ORD_GENERATOR` function will attempt to create a memory-based nested table with all one million rows, then pass that back to the `INSERT` statement.

A more efficient method will be a pipelined function. Turn to the next page for an example...

5.29

## ...Test Data Generator

```
create or replace function ord_generator_piped
  (p_num_rows in number)
  RETURN ord_table
  PIPELINED
AS
BEGIN
  for i in 1..p_num_rows loop

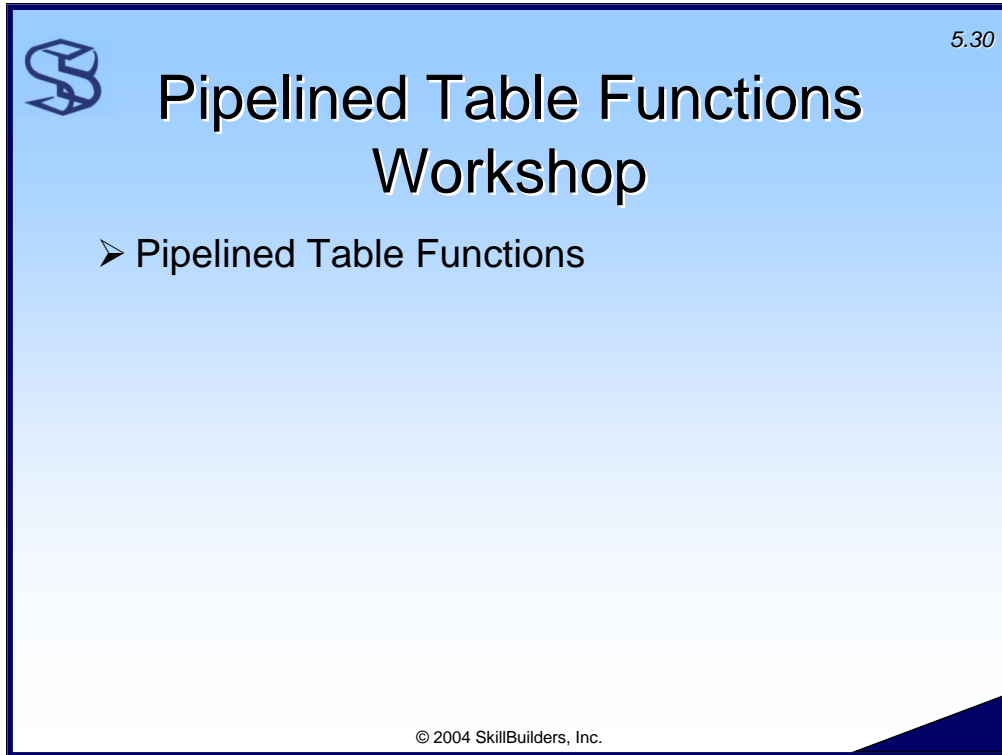
    PIPE ROW ( ord_type(
      i ,i ,sysdate - i ,0 ,sysdate - i
      ,null , 'CA' ,1 ,null ,null ));

  end loop;
  return;
END;
/
```

© 2004 SkillBuilders, Inc.

This version of the `ORD_GENERATOR` function is `PIPELINED`. Rather than constructing a memory-based one million row nested table, and returning that to the caller, this function pipes the rows back to the `INSERT` statement as they are generated. Consuming too much memory is not an issue!

See supplied script `TABLEFUNC4.SQL` for a working example of this code.



The slide features a blue gradient background with a dark blue border. In the top left corner is the Oracle logo. The title 'Pipelined Table Functions Workshop' is centered in a large, black, sans-serif font. Below the title is a bullet point with a right-pointing arrowhead, followed by the text 'Pipelined Table Functions'. In the top right corner, the number '5.30' is displayed. At the bottom center, there is a small copyright notice: '© 2004 SkillBuilders, Inc.'.

5.30

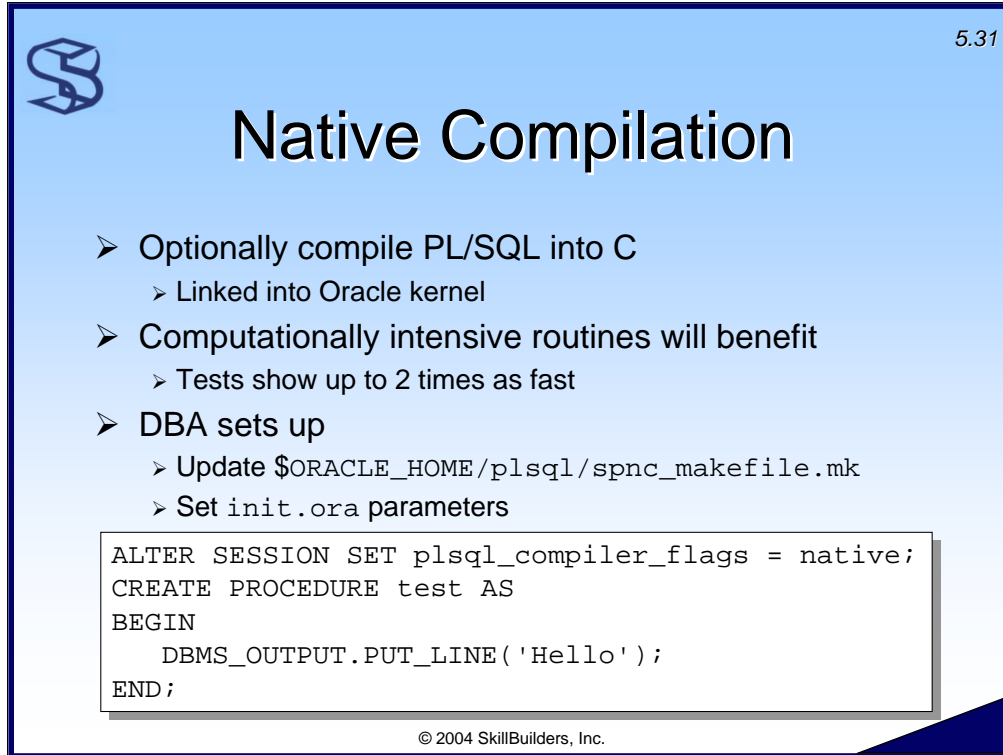
# Pipelined Table Functions Workshop

- Pipelined Table Functions

© 2004 SkillBuilders, Inc.

## Workshop – Pipelined Table Functions

1. Convert the supplied table function `tablefunc1.sql` to a pipelined function. Test by running the queries supplied at the end of `tablefunc1.sql`. The queries should all run successfully and produce 12 months (rows) of output. Note that most months will have zero sales; this is not an error.



5.31

## Native Compilation

- Optionally compile PL/SQL into C
  - Linked into Oracle kernel
- Computationally intensive routines will benefit
  - Tests show up to 2 times as fast
- DBA sets up
  - Update `$ORACLE_HOME/plsql/spnc_makefile.mk`
  - Set `init.ora` parameters

```
ALTER SESSION SET plsql_compiler_flags = native;
CREATE PROCEDURE test AS
BEGIN
    DBMS_OUTPUT.PUT_LINE('Hello');
END;
```

© 2004 SkillBuilders, Inc.

Oracle9i provides optional native compilation for PL/SQL programs. When native compilation is used, Oracle will convert the PL/SQL into C, compile the C into object code, then linked into the Oracle executable (kernel). When the routine is invoked, Oracle simply calls the linked subroutine.

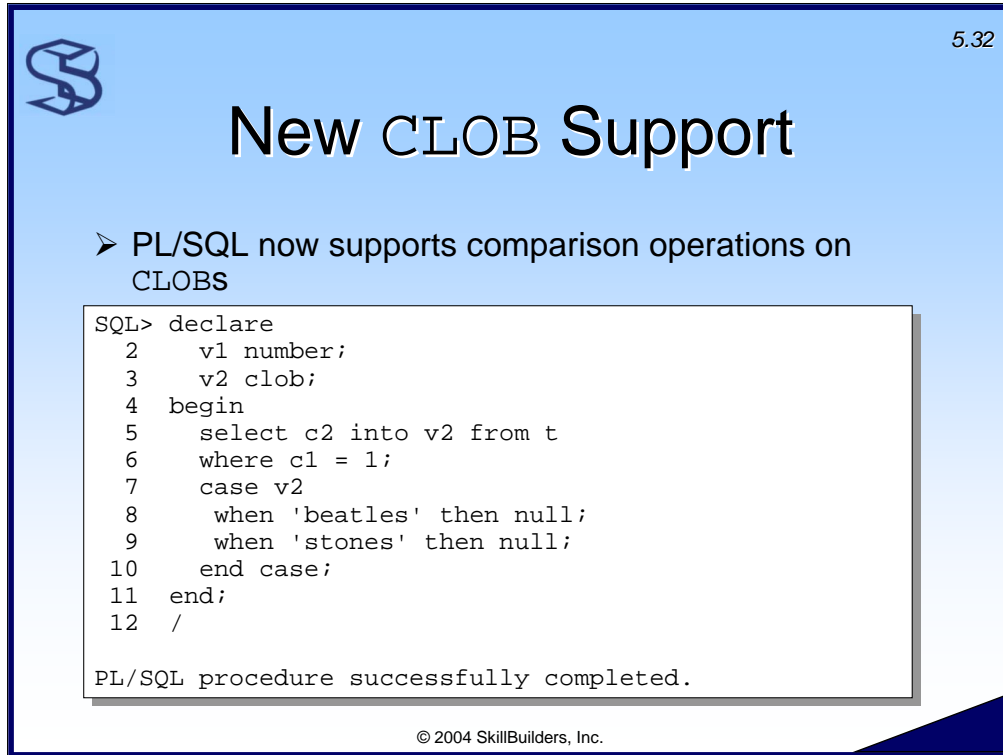
Programs that are reliant on lots of computations will run faster when compiled natively. Tests show up to 2x as fast as interpreted routines. (See [asktom.oracle.com](http://asktom.oracle.com) and search on “oracle9i pl/sql native compile” for some good examples.)

The DBA will need to setup the server for native compilation before it can be used.

### Supplemental Notes

Refer to the `init.ora` parameters `plsql_native_make_utility` and `plsql_native_make_file_name` when preparing your server for native compilation.

Query the data dictionary view `USER_STORED_SETTINGS` to determine the compilation method. The `PARAM_VALUE` column will contain `NATIVE` if native compilation was used to create the object.



5.32

## New CLOB Support

- PL/SQL now supports comparison operations on CLOBs

```
SQL> declare
  2   v1 number;
  3   v2 clob;
  4   begin
  5     select c2 into v2 from t
  6     where c1 = 1;
  7     case v2
  8       when 'beatles' then null;
  9       when 'stones' then null;
 10    end case;
 11 end;
 12 /
```

PL/SQL procedure successfully completed.

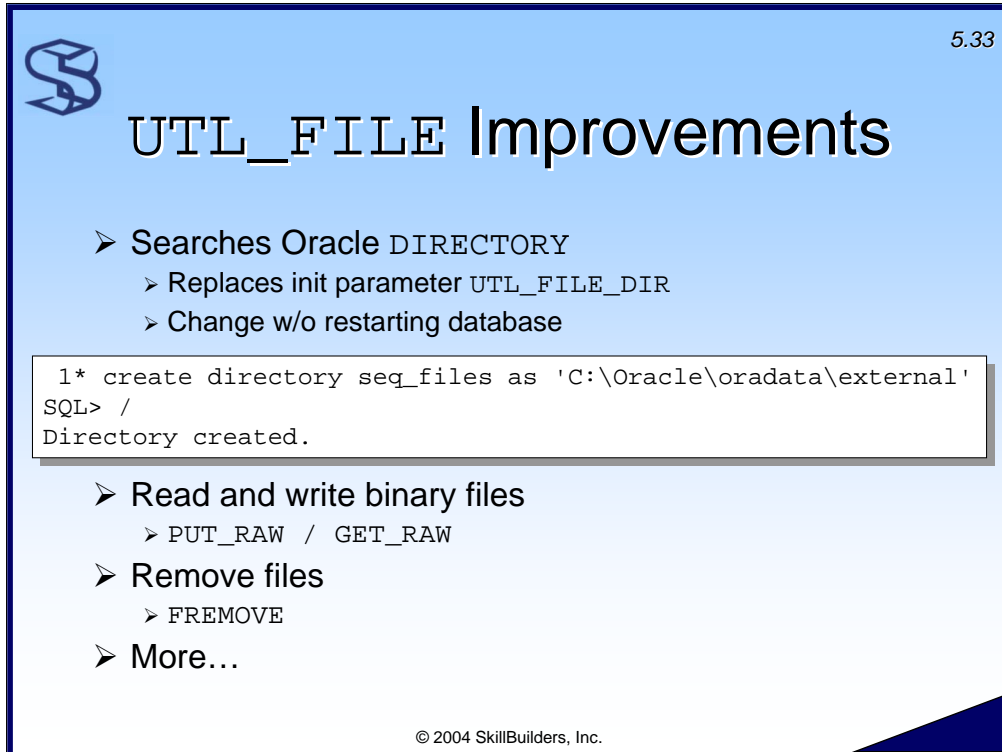
© 2004 SkillBuilders, Inc.

PL/SQL now supports comparison operations on CLOB columns. This includes CASE (as shown above) and standard operators such as =, <, >, IN, BETWEEN, LIKE.

```
SQL> declare
  2   v1 clob := 'abc';
  3   begin
  4     If v1 = 'abc' then null; end if;
  5     If v1 >= 'abc' then null; end if;
  6     If v1 between 'a' and 'z' then null; end if;
  7     If 'abc' IN (v1) then null; end if;
  8     If v1 like 'a%' then null; end if;
  9   end;
 10 /
```

PL/SQL procedure successfully completed.

See the supplied script `clob.sql` for a working example of the code shown here.



5.33

## UTL\_FILE Improvements

- Searches Oracle DIRECTORY
  - Replaces init parameter UTL\_FILE\_DIR
  - Change w/o restarting database

```
1* create directory seq_files as 'C:\Oracle\oradata\external'  
SQL> /  
Directory created.
```

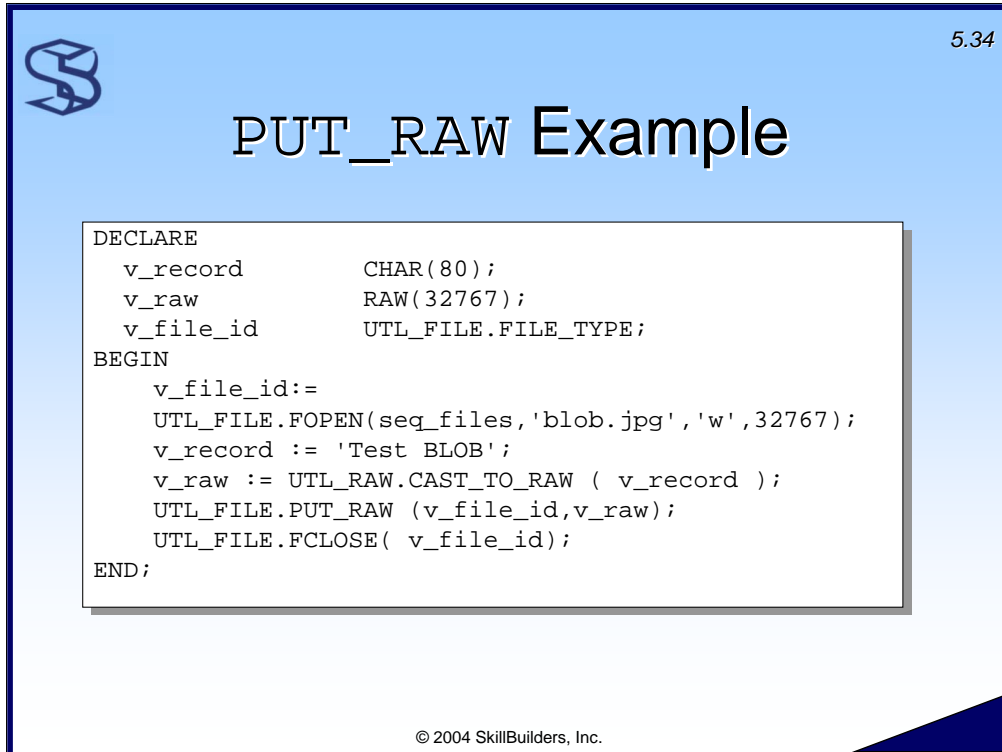
- Read and write binary files
  - PUT\_RAW / GET\_RAW
- Remove files
  - FREMOVE
- More...

© 2004 SkillBuilders, Inc.

Improvements have been made to the UTL\_FILE package in Oracle9i. Before UTL\_FILE can be used, accessible directories need to be defined. In the past this was done with the UTL\_FILE\_DIR init parameter. Enabling this parameter required a database shutdown and startup. Defining accessible directories can also be done with the CREATE DIRECTORY command. The CREATE DIRECTORY command requires additional privileges and would normally be executed by a DBA. Once the directory has been created, the DBA grants the necessary read / write privileges on the directory to the appropriate users and then you are ready to use the UTL\_FILE package.

Some new functions have been added to the UTL\_FILE package. Here are three very useful ones.

- PUT\_RAW – Accepts RAW (binary) data as input and writes it out to an operating system file.
- GET\_RAW – Reads RAW (binary) data from an operating system file.
- FREMOVE – Deletes an operating system file. The Oracle OS is must have the necessary operating system privileges.



The slide features a blue background with a white box containing PL/SQL code. In the top left corner of the slide is a stylized 'S' logo. The title 'PUT\_RAW Example' is centered at the top. The code is as follows:

```
DECLARE
  v_record      CHAR(80);
  v_raw         RAW(32767);
  v_file_id     UTL_FILE.FILE_TYPE;
BEGIN
  v_file_id:=
  UTL_FILE.FOPEN(seq_files,'blob.jpg','w',32767);
  v_record := 'Test BLOB';
  v_raw := UTL_RAW.CAST_TO_RAW ( v_record );
  UTL_FILE.PUT_RAW (v_file_id,v_raw);
  UTL_FILE.FCLOSE( v_file_id);
END;
```

© 2004 SkillBuilders, Inc.

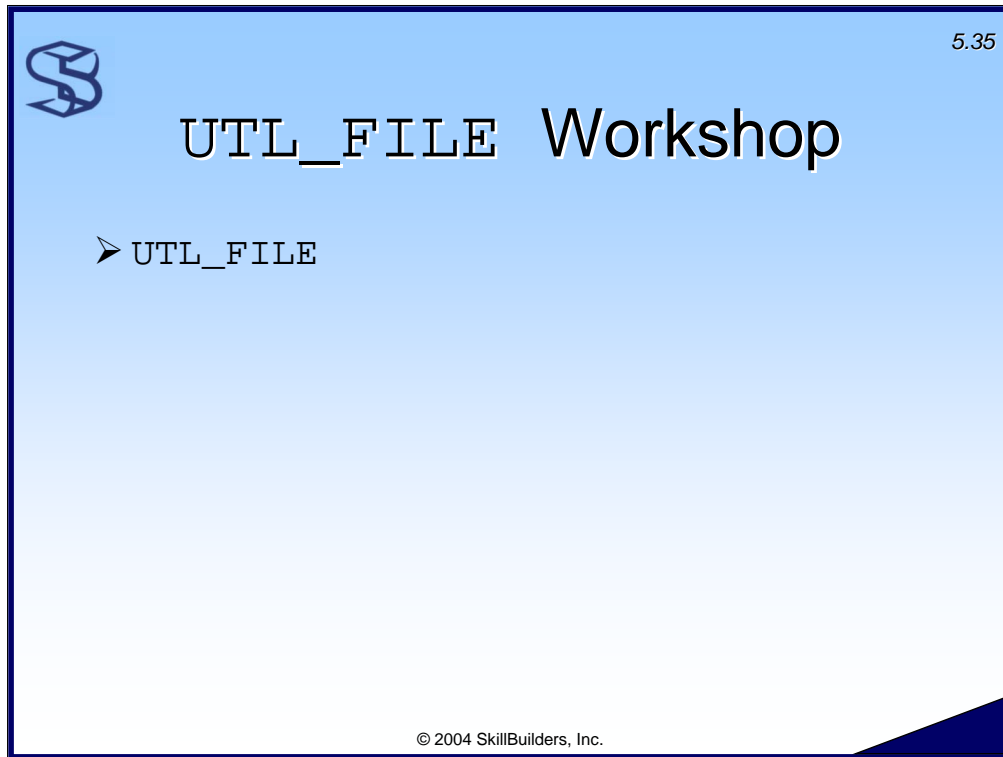
This example shows the usage of the `put_raw` function. It simulates a BLOB datatype by converting a character string to a BLOB.

The following statement writes the BLOB data out to a operating system file.

```
UTL_FILE.PUT_RAW (v_file_id,v_raw);
```


In using the `UTL_FILE` package and the `put_raw` function, all of the typical steps need to be executed. Those include getting a file id, opening the file, and closing the file.

See supplied script `PUT_RAW.SQL` for a working example of this code.



### Workshop - UTL\_FILE

1. Write a PL/SQL block to read an operating system file called `BLOB.JPG` that contains 1 row of binary data. Load the data into table `blob_table`. You may use any valid number for column `c1`'s data.



5.36

## Metadata Access...

- DBMS\_METADATA package allows access to object information

```
SELECT
  DBMS_METADATA.GET_DDL('TABLE',
                        'CUSTOMER', 'SYSTEM') FROM DUAL;
SELECT
  DBMS_METADATA.GET_XML('TABLE',
                        'CUSTOMER', 'SCOTT') FROM DUAL;
```


- See next page for output...

Case sensitive; use upper case

© 2004 SkillBuilders, Inc.

The first query allows one to retrieve the DDL that can create the `Customer` table. The second query retrieves the `Customer` table metadata in XML format.

See the supplied script `METADATA.SQL` for a working example.



5.37

## ...Metadata Access

```

SQL> set linesize 4000
SQL> Set long 4000
SQL> set heading off
SQL> SELECT
  2   DBMS_METADATA.GET_DDL('TABLE', 'CUSTOMER', 'DAVE')
  3   FROM DUAL;

CREATE TABLE "DAVE"."CUSTOMER"
(
  "CUST_NO" NUMBER(*,0),
  "LASTNAME" VARCHAR2(20) NOT NULL ENABLE,
  "FIRSTNAME" VARCHAR2(15) NOT NULL ENABLE,
  "MIDINIT" VARCHAR2(1),
  "CITY" VARCHAR2(20),
  "STATE" VARCHAR2(2),
  . . .

```

© 2004 SkillBuilders, Inc.

### The complete example:

```

LOCAL> set heading off
LOCAL> 1
1* SELECT DBMS_METADATA.GET_DDL('TABLE', 'CUSTOMER', 'DAVE') FROM DUAL
LOCAL> /

CREATE TABLE "DAVE"."CUSTOMER"
(
  "CUST_NO" NUMBER(*,0),
  "LASTNAME" VARCHAR2(20) NOT NULL ENABLE,
  "FIRSTNAME" VARCHAR2(15) NOT NULL ENABLE,
  "MIDINIT" VARCHAR2(1),
  "STREET" VARCHAR2(30),
  "CITY" VARCHAR2(20),
  "STATE" VARCHAR2(2),
  "ZIP" VARCHAR2(5),
  "ZIP_4" VARCHAR2(4),
  "AREA_CODE" VARCHAR2(3),
  "PHONE" VARCHAR2(8),
  "COMPANY_NAME" VARCHAR2(50),
  PRIMARY KEY ("CUST_NO")
  USING INDEX PCTFREE 10 INITRANS 2 MAXTRANS 255
  STORAGE(INITIAL 65536 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS 2147483645
  PCTINCREASE 0
  FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL DEFAULT) TABLESPACE "TOOLS" ENABLE )
  PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255 LOGGING
  STORAGE(INITIAL 65536 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS 2147483645
  PCTINCREASE 0
  FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL DEFAULT) TABLESPACE "TOOLS"

```



5.38

## Summary of PL/SQL Features

- ANSI compliant CASE statement
- Bulk Collect on DML Returning
- Associate Arrays
- Multi-level collections
- Record-Based DML
- Pipelined Functions
- Native Compilation
- Enhanced LOB support
- UTL\_FILE Enhancements
- Metadata Access

© 2004 SkillBuilders, Inc.