ORACLE® ACADEMY

Database Programming with PL/SQL

12-2
Improving PL/SQL Performance

ORACLE ACADEMY

Objectives

This lesson covers the following objectives:

- Identify the benefits of the NOCOPY hint and the DETERMINISTIC clause
- Create subprograms which use the NOCOPY hint and the DETERMINISTIC clause
- Use Bulk Binding FORALL in a DML statement
- Use BULK COLLECT in a SELECT or FETCH statement
- Use the Bulk Binding RETURNING clause

Purpose

- Until now, you have learned how to write, compile, and execute PL/SQL code without thinking much about how long the execution will take.
- None of the tables you use in this course contain more than a few hundred rows, so the execution is always fast.
- But in real organizations, tables can contain millions or even billions of rows.
- Obviously, processing two million rows takes much longer than processing twenty rows.
- In this lesson you will learn some ways to speed up the processing of very large sets of data.

Using the NOCOPY Hint

- In PL/SQL and most other programming languages, there are two ways to pass parameter arguments between a calling program and a called subprogram: by *value* and by *reference*.
- Passing by value means that the argument values are *copied* from the calling program's memory to the subprogram's memory, and copied back again when the subprogram is exited.
- So while the subprogram is executing, there are two copies of each argument.

Using the NOCOPY Hint

- Passing by *reference* means that the argument values are not copied.
- The two programs share a single copy of the data.
- While passing by *value* is safer, it can use a lot of memory and execute slowly if the argument value is large.
- Look at this fragment of code:

```
CREATE OR REPLACE PACKAGE emp_pkg IS
  TYPE t_emp IS TABLE OF employees%ROWTYPE
                INDEX BY BINARY_INTEGER;
  PROCEDURE emp_proc
    (p_small_arg IN NUMBER, p_big_arg OUT t_emp);
...
END emp_pkg;
```

Using the NOCOPY Hint

- Suppose EMP_PKG.EMP_PROC fetches one million EMPLOYEES rows into P_BIG_ARG.
- That’s a lot of memory!
- And those one million rows must be copied to the calling environment at the end of the procedure’s execution.
- That’s a lot of time.

Using the NOCOPY Hint

Maybe we should pass P_BIG_ARG by *reference* instead of by value.

```
CREATE OR REPLACE PACKAGE emp_pkg IS
  TYPE t_emp IS TABLE OF employees%ROWTYPE
                INDEX BY BINARY_INTEGER;
  PROCEDURE emp_proc
    (p_small_arg IN NUMBER, p_big_arg OUT t_emp);
...
END emp_pkg;
```

Using the NOCOPY Hint

- By default, PL/SQL IN parameter arguments are passed by reference, while OUT and IN OUT arguments are passed by value.
- We can change this to pass an OUT or IN OUT argument by reference, using the NOCOPY hint.

```
CREATE OR REPLACE PACKAGE emp_pkg IS
  TYPE t_emp IS TABLE OF employees%ROWTYPE
                INDEX BY BINARY_INTEGER;
  PROCEDURE emp_proc
    (p_small_arg IN NUMBER, p_big_arg OUT NOCOPY t_emp);
...
END emp_pkg;
```

Using the NOCOPY Hint

- Notice that NOCOPY must come immediately after the parameter mode (OUT or IN OUT).
- Specify NOCOPY to instruct the database to pass an argument as fast as possible.
- This clause can significantly enhance performance when passing a large value.

```
CREATE OR REPLACE PACKAGE emp_pkg IS
  TYPE t_emp IS TABLE OF employees%ROWTYPE
                INDEX BY BINARY_INTEGER;
  PROCEDURE emp_proc
    (p_small_arg IN NUMBER, p_big_arg OUT NOCOPY t_emp);
...
END emp_pkg;
```

Function Based Indexes

- All of the Function Based Index examples have demonstrated the use of the UPPER and LOWER functions.
- While these two are frequently used in Function Based Indexes, the Oracle database is not limited to just allowing those two functions in an index.
- Any valid Oracle built-in function can be used in a Function-Based Index.
- Also, any database function you write yourself can be used.

Function Based Indexes

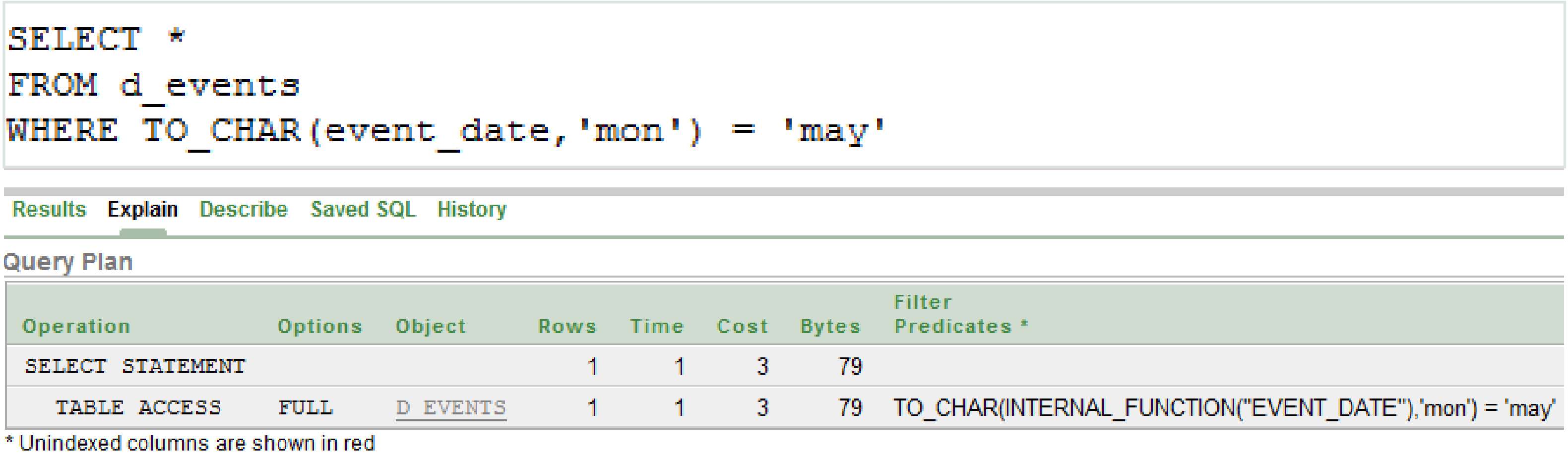
- There is one rule you must remember: if you are writing your own functions to use in a Function Based Index, you must include the key word `DETERMINISTIC` in the function header.
- In mathematics, a deterministic system is a system in which no randomness is involved in the development of future states of the system.
- Deterministic models therefore produce the same output for a given starting condition.

Function Based Indexes

- In Oracle, the term deterministic declares that a function, when given the same inputs, will always return the exact same output.
- You must tell Oracle that the function is DETERMINISTIC and will return a consistent result given the same inputs.
- The built-in SQL functions UPPER, LOWER, and TO_CHAR are already defined as deterministic by Oracle so this is why you can create an index on the UPPER value of a column.

Function Based Indexes

- The results of another example of Function Based Indexes is shown below.
- The d_events table was queried to find any events planned for the month of May.



```
SELECT *
FROM d_events
WHERE TO_CHAR(event_date,'mon') = 'may'
```

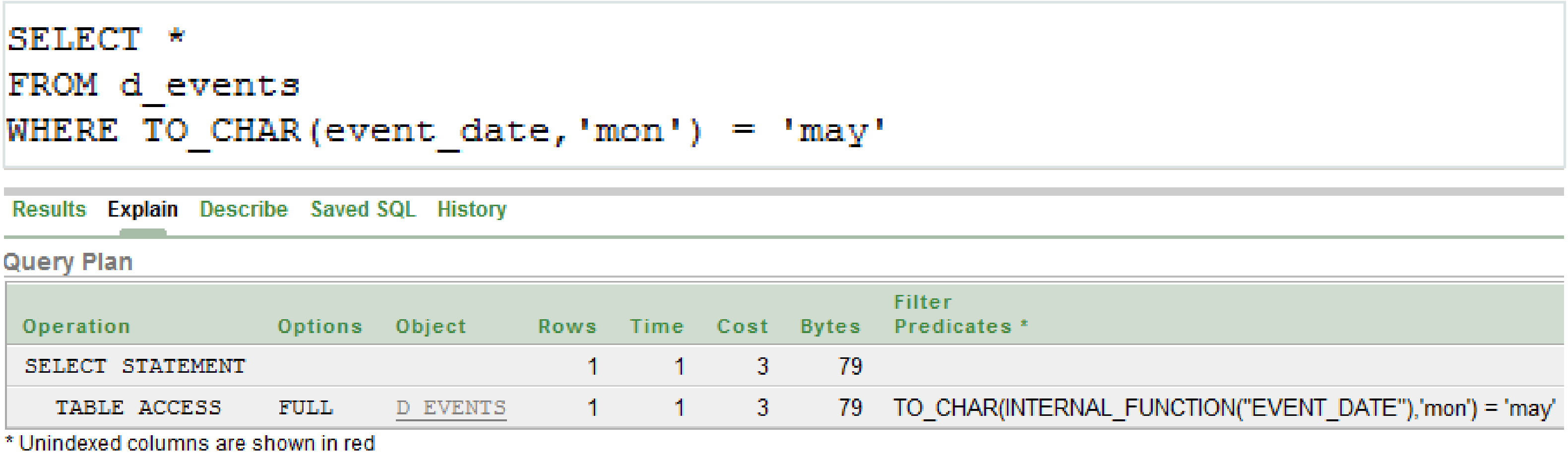
Query Plan

Operation	Options	Object	Rows	Time	Cost	Bytes	Filter Predicates *
SELECT STATEMENT			1	1	3	79	
TABLE ACCESS	FULL	D_EVENTS	1	1	3	79	TO_CHAR(INTERNAL_FUNCTION("EVENT_DATE"),'mon') = 'may'

* Unindexed columns are shown in red

Function Based Indexes

- As the Query Plan results indicate, this query executed a Full Table Scan, which can be a very time-intensive operation when a table has a lot of rows.
- Even though the event_date column is indexed, the index is not used, due to the TO_CHAR expression.



```
SELECT *
FROM d_events
WHERE TO_CHAR(event_date,'mon') = 'may'
```

Query Plan

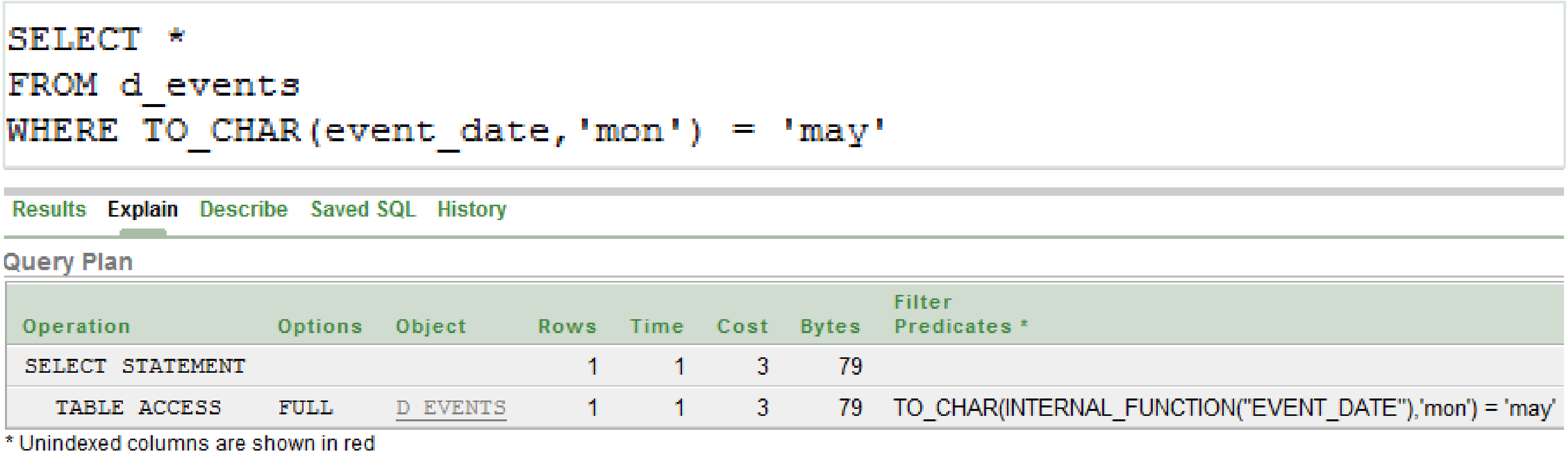
Operation	Options	Object	Rows	Time	Cost	Bytes	Filter Predicates *
SELECT STATEMENT			1	1	3	79	
TABLE ACCESS	FULL	D_EVENTS	1	1	3	79	TO_CHAR(INTERNAL_FUNCTION("EVENT_DATE"),'mon') = 'may'

* Unindexed columns are shown in red

Function Based Indexes

- Once we create the following Function Based Index, we can run the same query, but this time avoid the time-intensive Full Table Scan.
- The index on the event_date column can now be used.

```
CREATE INDEX d_evnt_dt_indx
  ON d_events (TO_CHAR(event_date,'mon'))
```



```
SELECT *
FROM d_events
WHERE TO_CHAR(event_date,'mon') = 'may'
```

Query Plan

Operation	Options	Object	Rows	Time	Cost	Bytes	Filter Predicates *
SELECT STATEMENT			1	1	3	79	
TABLE ACCESS	FULL	D_EVENTS	1	1	3	79	TO_CHAR(INTERNAL_FUNCTION("EVENT_DATE"),'mon') = 'may'

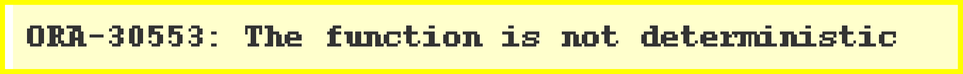
* Unindexed columns are shown in red

Function Based Indexes

Now create your own PL/SQL function and try to create a Function Based Index on it:

```
CREATE OR REPLACE FUNCTION twicenum
  (p_number IN NUMBER)
  RETURN NUMBER IS
BEGIN
  RETURN p_number * 2;
END twicenum;
```

```
CREATE INDEX emp_twicesal_idx
  ON employees(twicenum(salary));
```



```
ORA-30553: The function is not deterministic
```

Using the DETERMINISTIC Clause

- If you want to create a Function Based Index on your own functions (not the built-in functions like MOD) you must create the function using the DETERMINISTIC clause:

```
CREATE OR REPLACE FUNCTION twicenum
  (p_number IN NUMBER)
  RETURN NUMBER DETERMINISTIC IS
BEGIN
  RETURN p_number * 2;
END twicenum;
```

- Now the index can be created successfully:

```
CREATE INDEX emp_twicesal_idx
  ON employees(twicenum(salary));
```

Using the DETERMINISTIC Clause

- Be careful!
- The word “deterministic” means that the same input value will always produce the same output value.
- Look at this function:

```
CREATE OR REPLACE FUNCTION total_sal
  (p_dept_id IN employees.department_id%TYPE)
  RETURN NUMBER DETERMINISTIC IS
  v_total_sal  NUMBER;
BEGIN
  SELECT SUM(salary) INTO v_total_sal
    FROM employees WHERE department_id =
p_dept_id;
  RETURN v_total_sal;
END total_sal;
```

Using the DETERMINISTIC Clause

- The function on the previous slide is not really deterministic, but the Oracle server still allowed you to create it.
- What if we give everyone a salary increase?

```
UPDATE employees SET salary = salary * 1.10;
COMMIT;
```

- Now the `SUM(salary)` values stored in the index are out-of-date, and the index will not be used unless you DROP and CREATE it again.
- This will take a long time on a very large table.
- Do *NOT* create a deterministic function which contains a SELECT statement on data which may be modified in the future.

What is Bulk Binding?

- Many PL/SQL blocks contain both PL/SQL statements and SQL statements, each of which is executed by a different part of the Oracle software called the *PL/SQL Engine* and the *SQL Engine*.
- A change from one engine to the other is called a *context switch*, and takes time.
- For one change, this is at most a few milliseconds.
- But what if there are millions of changes?

What is Bulk Binding?

- If we FETCH (in a cursor) and process millions of rows one at a time, that's millions of context switches.
- And that will really slow down the execution.
- FETCH is a SQL statement because it accesses database tables, but the processing is done by PL/SQL statements.

What is Bulk Binding?

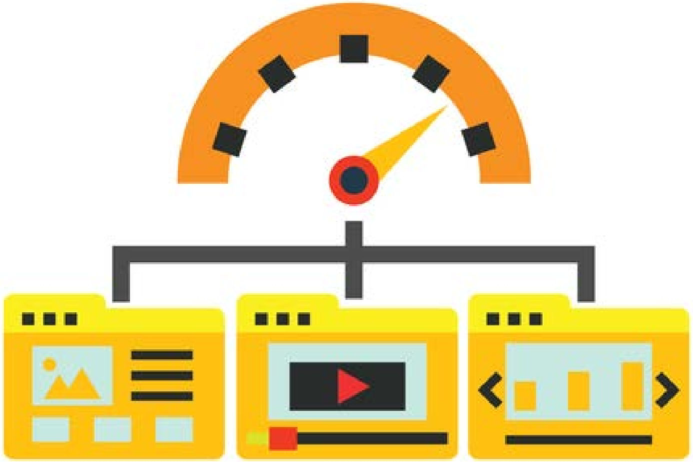
- Look at this code, and imagine that our EMPLOYEES table has one million rows.
- How many context switches occur during one execution of the procedure?

```
CREATE OR REPLACE PROCEDURE fetch_all_emps IS
  CURSOR emp_curs IS SELECT * FROM employees;
BEGIN
  FOR v_emprec IN emp_curs LOOP
    DBMS_OUTPUT.PUT_LINE(v_emprec.first_name);
  END LOOP;
END fetch_all_emps;
```

- Remember that in a cursor FOR loop, all the fetches are still executed even though we do not explicitly code a FETCH statement.

What is Bulk Binding?

- It would be much quicker to fetch all the rows in just one context switch within the SQL Engine.
- This is what Bulk Binding does.
- Of course, if all the rows are fetched in one statement, we will need an `INDEX BY` table of records to store all the fetched rows.



What is Bulk Binding?

- If each row is (on average) 100 bytes in size, storing one million rows will need 100 megabytes of memory.
- When you think about many users accessing a database, you can see how memory usage could become an issue.
- So Bulk Binding is a trade-off: more memory required (possibly bad) but faster execution (good).

Bulk Binding a SELECT: Using BULK COLLECT

- Here is the one million row table from the earlier slide, this time using Bulk Binding to fetch all the rows in a single call to the SQL Engine.

```
CREATE OR REPLACE PROCEDURE fetch_all_emps IS
  TYPE t_emp IS TABLE OF employees%ROWTYPE INDEX BY BINARY_INTEGER;
  v_emptab       t_emp;
BEGIN
  SELECT * BULK COLLECT INTO v_emptab FROM employees;
  FOR i IN v_emptab.FIRST..v_emptab.LAST LOOP
    IF v_emptab.EXISTS(i) THEN
      DBMS_OUTPUT.PUT_LINE(v_emptab(i).last_name);
    END IF;
  END LOOP;
END fetch_all_emps;
```

- Now how many context switches are there?

Bulk Binding a SELECT: Using BULK COLLECT

- When using BULK COLLECT, we do not declare a cursor because we do not fetch individual rows one at a time.
- Instead, we SELECT the whole database table into the PL/SQL INDEX BY table in a single SQL statement.
- Here is another example:

```
CREATE OR REPLACE PROCEDURE fetch_some_emps IS
  TYPE t_salary IS TABLE OF employees.salary%TYPE
                   INDEX BY BINARY_INTEGER;
  v_saltab       t_salary;
BEGIN
  SELECT salary BULK COLLECT INTO v_saltab
    FROM employees WHERE department_id = 20 ORDER BY salary;
  FOR i IN v_saltab.FIRST..v_saltab.LAST LOOP
    IF v_saltab.EXISTS(i) THEN
      DBMS_OUTPUT.PUT_LINE(v_saltab(i));
    END IF;
  END LOOP;
END fetch_some_emps;
```

Bulk Binding with DML: Using FORALL

- We may also want to speed up DML statements which process many rows.
- Look at this code:

```
CREATE OR REPLACE PROCEDURE insert_emps IS
  TYPE t_emps IS TABLE OF employees%ROWTYPE INDEX BY BINARY_INTEGER;
  v_emptab       t_emps;
BEGIN
  FOR i IN v_emptab.FIRST..v_emptab.LAST LOOP
    INSERT INTO employees VALUES v_emptab(i);
 END LOOP;
END insert_emps;
```

- Again, if we are inserting one million rows, this is one million executions of an INSERT SQL statement.
- How many context switches?

Bulk Binding with DML: Using FORALL

- Just like `BULK COLLECT`, there is no `LOOP...END LOOP` code because all the rows are inserted with a single call to the SQL Engine.
- The example on the slide will compile, but will not perform any inserts as the v_emptab table is not populated in this code example.

```
CREATE OR REPLACE PROCEDURE insert_emps IS
  TYPE t_emps IS TABLE OF employees%ROWTYPE INDEX BY BINARY_INTEGER;
  v_emptab       t_emps;
BEGIN
  FORALL i IN v_emptab.FIRST..v_emptab.LAST
    INSERT INTO employees VALUES v_emptab(i);
END insert_emps;
```

Bulk Binding with DML: Using FORALL

- We can combine BULK COLLECT and FORALL.
- Suppose we want to copy millions of rows from one table to another:

```
CREATE OR REPLACE PROCEDURE copy_emps IS
  TYPE t_emps IS TABLE OF employees%ROWTYPE INDEX BY BINARY_INTEGER;
  v_emptab       t_emps;
BEGIN
  SELECT * BULK COLLECT INTO v_emptab FROM employees;
  FORALL i IN v_emptab.FIRST..v_emptab.LAST
    INSERT INTO new_employees VALUES v_emptab(i);
END copy_emps;
```

Bulk Binding with DML: Using FORALL

We can use FORALL with UPDATE and DELETE statements as well as with INSERT:

```
CREATE OR REPLACE PROCEDURE update_emps IS
  TYPE t_emp_id IS TABLE OF employees.employee_id%TYPE
                   INDEX BY BINARY_INTEGER;
  v_emp_id_tab     t_emp_id;
BEGIN
  SELECT employee_id BULK COLLECT INTO v_emp_id_tab FROM employees;
  FORALL i IN v_emp_id_tab.FIRST..v_emp_id_tab.LAST
    UPDATE new_employees
      SET salary = salary * 1.05
      WHERE employee_id = v_emp_id_tab(i);
END update_emps;
```

Bulk Binding Cursor Attributes: SQL%BULK_ROWCOUNT

In addition to implicit cursor attributes such as SQL%ROWCOUNT, Bulk Binding uses two extra cursor attributes, which are both INDEX BY tables.

```
CREATE OR REPLACE PROCEDURE insert_emps IS
  TYPE t_emps IS TABLE OF employees%ROWTYPE
  INDEX BY BINARY_INTEGER;
  v_emptab       t_emps;
BEGIN
  SELECT * BULK COLLECT INTO v_emptab FROM employees;
  FORALL i IN v_emptab.FIRST..v_emptab.LAST
    INSERT INTO emp VALUES v_emptab(i);
  FOR i IN v_emptab.FIRST..v_emptab.LAST LOOP
    DBMS_OUTPUT.PUT_LINE('Inserted: '
        || i || ' '||SQL%BULK_ROWCOUNT(i)|| 'rows');
  END LOOP;
END insert_emps;
```

Bulk Binding Cursor Attributes: SQL%BULK_ROWCOUNT

SQL%BULK_ROWCOUNT(i) shows the number of rows processed by the i^th execution of a DML statement when using FORALL:

```
CREATE OR REPLACE PROCEDURE insert_emps IS
  TYPE t_emps IS TABLE OF employees%ROWTYPE
  INDEX BY BINARY_INTEGER;
  v_emptab       t_emps;
BEGIN
  SELECT * BULK COLLECT INTO v_emptab FROM employees;
  FORALL i IN v_emptab.FIRST..v_emptab.LAST
    INSERT INTO emp VALUES v_emptab(i);
  FOR i IN v_emptab.FIRST..v_emptab.LAST LOOP
    DBMS_OUTPUT.PUT_LINE('Inserted: '
        || i || ' '||SQL%BULK_ROWCOUNT(i)|| 'rows');
  END LOOP;
END insert_emps;
```

Bulk Binding Cursor Attributes: SQL%BULK_EXCEPTIONS

- Look again at our first example of using FORALL:

```
CREATE OR REPLACE PROCEDURE insert_emps IS
  TYPE t_emps IS TABLE OF employees%ROWTYPE INDEX BY BINARY_INTEGER;
  v_emptab       t_emps;
BEGIN
  SELECT * BULK COLLECT INTO v_emptab FROM employees;
  FORALL i IN v_emptab.FIRST..v_emptab.LAST
    INSERT INTO employees VALUES v_emptab(i);
END insert_emps;
```

- What if one of the INSERTs fails, perhaps because a constraint was violated?

Bulk Binding Cursor Attributes: SQL%BULK_EXCEPTIONS

- The whole FORALL statement fails, so no rows are inserted. And you don’t even know which row failed to insert!
- That has wasted a lot of time.

```
CREATE OR REPLACE PROCEDURE insert_emps IS
  TYPE t_emps IS TABLE OF employees%ROWTYPE INDEX BY BINARY_INTEGER;
  v_emptab       t_emps;
BEGIN
  SELECT * BULK COLLECT INTO v_emptab FROM employees;
  FORALL i IN v_emptab.FIRST..v_emptab.LAST
    INSERT INTO employees VALUES v_emptab(i);
END insert_emps;
```

Bulk Binding Cursor Attributes: SQL%BULK_EXCEPTIONS

We add SAVE EXCEPTIONS to our FORALL statement:

```
CREATE OR REPLACE PROCEDURE insert_emps IS
  TYPE t_emps IS TABLE OF employees%ROWTYPE INDEX BY BINARY_INTEGER;
  v_emptab       t_emps;
BEGIN
  SELECT * BULK COLLECT INTO v_emptab FROM employees;
  FORALL i IN v_emptab.FIRST..v_emptab.LAST SAVE EXCEPTIONS
    INSERT INTO employees VALUES v_emptab(i);
END insert_emps;
```

Bulk Binding Cursor Attributes: SQL%BULK_EXCEPTIONS

- Now, all the non-violating rows will be inserted.
- The violating rows populate an INDEX BY table called SQL%BULK_EXCEPTIONS which has two fields: ERROR_INDEX shows which inserts failed (first, second, …) and ERROR_CODE shows the Oracle Server predefined error code.

```
CREATE OR REPLACE PROCEDURE insert_emps IS
  TYPE t_emps IS TABLE OF employees%ROWTYPE INDEX BY BINARY_INTEGER;
  v_emptab       t_emps;
BEGIN
  SELECT * BULK COLLECT INTO v_emptab FROM employees;
  FORALL i IN v_emptab.FIRST..v_emptab.LAST SAVE EXCEPTIONS
    INSERT INTO employees VALUES v_emptab(i);
END insert_emps;
```

Bulk Binding Cursor Attributes: SQL%BULK_EXCEPTIONS

An exception has been raised (at least one row failed to insert) so we must code the display of SQL%BULK_EXCEPTIONS in the EXCEPTION section.

```
CREATE OR REPLACE PROCEDURE insert_emps IS
  TYPE t_emps IS TABLE OF employees%ROWTYPE INDEX BY BINARY_INTEGER;
  v_emptab       t_emps;
BEGIN
  SELECT * BULK COLLECT INTO v_emptab FROM employees;
  FORALL i IN v_emptab.FIRST..v_emptab.LAST SAVE EXCEPTIONS
    INSERT INTO employees VALUES v_emptab(i);
EXCEPTION
WHEN OTHERS THEN
  FOR j in 1..SQL%BULK_EXCEPTIONS.COUNT LOOP
    DBMS_OUTPUT.PUT_LINE(SQL%BULK_EXCEPTIONS(j).ERROR_INDEX);
    DBMS_OUTPUT.PUT_LINE(SQL%BULK_EXCEPTIONS(j).ERROR_CODE);
  END LOOP;
END insert_emps;
```

Using the RETURNING Clause

- Sometimes we need to DML a row, then SELECT column values from the updated row for later use:

```
CREATE OR REPLACE PROCEDURE update_one_emp
  (p_emp_id                 IN  employees.employee_id%TYPE,
   p_salary_raise_percent IN  NUMBER) IS
   v_new_salary           employees.salary%TYPE;
BEGIN
  UPDATE employees
    SET salary = salary * (1 + p_salary_raise_percent)
    WHERE employee_id = p_emp_id;
  SELECT salary INTO v_new_salary
    FROM employees
    WHERE employee_id = p_emp_id;
  DBMS_OUTPUT.PUT_LINE('New salary is: ' || v_new_salary);
END update_one_emp;
```

- Two SQL statements are required: an UPDATE and a SELECT.

Using the RETURNING Clause

- However, we can do the SELECT within the UPDATE statement:

```
CREATE OR REPLACE PROCEDURE update_one_emp
  (p_emp_id               IN  employees.employee_id%TYPE,
   p_salary_raise_percent IN  NUMBER) IS
   v_new_salary           employees.salary%TYPE;
BEGIN
  UPDATE employees
    SET salary = salary * (1 + p_salary_raise_percent)
    WHERE employee_id = p_emp_id
    RETURNING salary INTO v_new_salary;
  DBMS_OUTPUT.PUT_LINE('New salary is: ' || v_new_salary);
END update_one_emp;
```

- This is faster because it makes only one call to the SQL Engine.

Using the RETURNING Clause with FORALL

What if we want to update millions of rows and see the updated values?

```
CREATE OR REPLACE PROCEDURE update_all_emps
  (p_salary_raise_percent IN  NUMBER) IS
  TYPE t_empid IS TABLE OF employees.employee_id%TYPE
                  INDEX BY BINARY_INTEGER;
  TYPE t_sal IS   TABLE OF employees.salary%TYPE
                  INDEX BY BINARY_INTEGER;
  v_empidtab      t_empid;
  v_saltab        t_sal;
BEGIN
  SELECT employee_id BULK COLLECT INTO v_empidtab FROM employees;
  FORALL i IN v_empidtab.FIRST..v_empidtab.LAST
    UPDATE employees
      SET salary = salary * (1 + p_salary_raise_percent)
      WHERE employee_id = v_empidtab(i);
  SELECT salary BULK COLLECT INTO v_saltab FROM employees;
END update_all_emps;
```

Using the RETURNING Clause with FORALL

We can use RETURNING with a Bulk Binding FORALL clause:

```
CREATE OR REPLACE PROCEDURE update_all_emps
  (p_salary_raise_percent IN  NUMBER) IS
  TYPE t_empid IS TABLE OF employees.employee_id%TYPE
                  INDEX BY BINARY_INTEGER;
  TYPE t_sal IS   TABLE OF employees.salary%TYPE
                  INDEX BY BINARY_INTEGER;
  v_empidtab      t_empid;
  v_saltab        t_sal;
BEGIN
  SELECT employee_id BULK COLLECT INTO v_empidtab FROM employees;
  FORALL i IN v_empidtab.FIRST..v_empidtab.LAST
    UPDATE employees
      SET salary = salary * (1 + p_salary_raise_percent)
      WHERE employee_id = v_empidtab(i)
      RETURNING salary BULK COLLECT INTO v_saltab;
END update_all_emps;
```

Terminology

Key terms used in this lesson included:

- Bulk Binding
- `BULK COLLECT` Clause
- `DETERMINISTIC` Clause
- `FORALL`
- `NOCOPY` hint
- `RETURNING` Clause

Summary

In this lesson, you should have learned how to:

- Identify the benefits of the NOCOPY hint and the DETERMINISTIC clause
- Create subprograms which use the NOCOPY hint and the DETERMINISTIC clause
- Use Bulk Binding FORALL in a DML statement
- Use BULK COLLECT in a SELECT or FETCH statement
- Use the Bulk Binding RETURNING clause

ORACLE® ACADEMY