Using Dynamic SQL

Oracle Academy Study Materials

Using Dynamic SQL

All SQL statements in the database go through various stages:

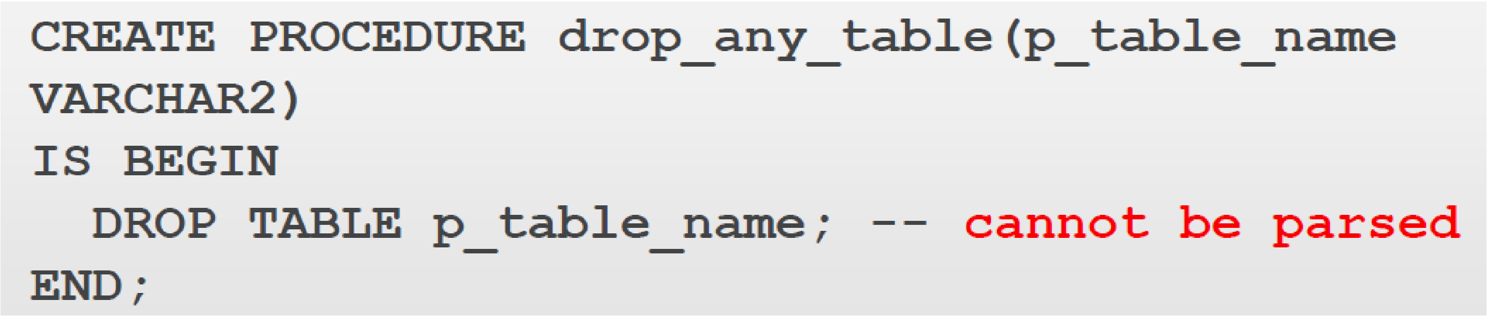
1. Parse: Pre-execution “is this possible?” checks syntax, object existence, privileges, and so on
2. Bind: Getting the actual values of any variables referenced in the statement
3. Execute: The statement is executed.
4. Fetch: Results are returned to the user.

Some stages might not be relevant for all statements:

- The fetch phase is applicable to queries.
- For embedded SQL statements such as SELECT, DML, MERGE, COMMIT, SAVEPOINT, and ROLLBACK, the parse and bind phases are done at compile time.
- For dynamic SQL statements, all phases are performed at run time.

Using Dynamic SQL

- When a SQL statement is included in a PL/SQL subprogram, the parse and bind phases are normally done at compile time, that is, when the procedure or function is created.
- If the text of the SQL statement is not known when the procedure is created, the Oracle server cannot parse it.
- Example:



```
CREATE PROCEDURE drop_any_table(p_table_name
VARCHAR2)
IS BEGIN
  DROP TABLE p_table_name; -- cannot be parsed
END;
```

Using Dynamic SQL

The embedded SQL statements available in PL/SQL are limited to :

- SELECT, INSERT, UPDATE, DELETE, COMMIT, and ROLLBACK,
- All of which are parsed at compile time; that is, they have a fixed structure.

Use Dynamic SQL functionality if you require:

- The structure of a SQL statement to be altered at execution time
- To access to DDL statements and other SQL functionality in PL/SQL
- To create a SQL statement whose text is not completely known in advance.

Dynamic SQL:

- Is constructed and stored as a character string within a subprogram.
- Is a SQL statement with varying column data, or different conditions with or without bind variables.
- Enables data-definition, data-control, or session-control statements to be written and executed from PL/SQL.

Using Dynamic SQL

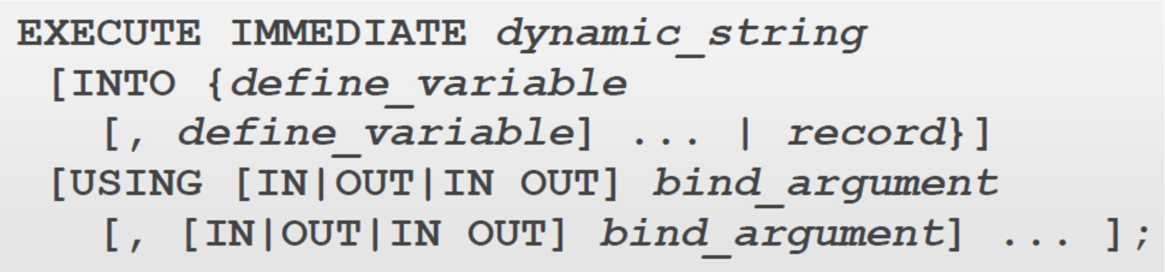
- PL/SQL does not support DDL statements written directly in a program.
- To perform these kinds of tasks in PL/SQL, you must construct SQL statements dynamically in character strings and execute them using either of the following:
  - Native Dynamic SQL statements with EXECUTE IMMEDIATE
  - The DBMS_SQL package
- The process of using SQL statements that are not embedded in your source program and are constructed in strings and executed at run time is known as “dynamic SQL.”

Using Dynamic SQL

- Native Dynamic SQL (NDS) allows you to work around this by constructing and storing SQL as a character string within a subprogram.
- NDS:
  - Provides native support for Dynamic SQL directly in the PL/SQL language.
  - Enables data-definition, data-control, or session-control statements to be written and executed from PL/SQL.
  - Provides the ability to execute SQL statements whose structure is unknown until execution time.
  - Can also use the OPEN-FOR, FETCH, and CLOSE PL/SQL statements.

Using Dynamic SQL

Using the EXECUTE IMMEDIATE Statement:



```
EXECUTE IMMEDIATE dynamic_string
 [INTO {define_variable
   [, define_variable] ... | record}]
 [USING [IN|OUT|IN OUT] bind_argument
   [, [IN|OUT|IN OUT] bind_argument] ... ];
```

- INTO is used for single-row queries and specifies the variables or records into which column values are retrieved.
- USING holds all bind arguments. The default parameter mode is IN, if not specified.
- dynamic_string is a character variable or literal containing the text of a SQL statement.
- define_variable is a PL/SQL variable that stores a selected column value.
- record is a user-defined or %ROWTYPE record that stores a selected row.
- bind_argument is an expression whose value is passed to the dynamic SQL statement at execution time.

Available Methods for Using NDS

Method #	SQL Statement Type	NDS SQL Statements Used
Method 1	***Non-query without*** host variables	`EXECUTE IMMEDIATE` without the `USING` and `INTO` clauses
Method 2	***Non-query with*** known number of input host variables	`EXECUTE IMMEDIATE` with a `USING` clause
Method 3	***Query with known*** number of select-list items and input host variables	`EXECUTE IMMEDIATE` with the `USING` and `INTO` clauses
Method 4	***Query with unknown*** number of select-list items or input host variables	Use the `DBMS_SQL` package

Available Methods for Using NDS

Method 1:

- This method lets the program accept or build a dynamic SQL statement, then immediately execute it using the EXECUTE IMMEDIATE command. The SQL statement must not be a query (SELECT statement) and must not contain any placeholders for input host variables.
- With Method 1, the SQL statement is parsed every time it is executed.
- Examples of non-queries include data definition language (DDLs) statements, UPDATEs, INSERTs, or DELETEs.

Available Methods for Using NDS

Method 2:

- This method lets the program accept or build a dynamic SQL statement, then process it using the PREPARE and EXECUTE commands.
- The SQL statement must not be a query.
- The number of placeholders for input host variables and the data types of the input host variables must be known at precompile time.
- Examples:
  - INSERT INTO EMPLOYEES (FIRST_NAME, LAST_NAME, JOB_ID) VALUES (:emp_first_name, :emp_last_name,:job_id)
  - DELETE FROM EMPLOYEES WHERE EMPLOYEE_ID = :emp_number
- With Method 2, the SQL statement is parsed just once, but can be executed many times with different values for the host variables.
- SQL data definition statements such as CREATE and GRANT are executed when they are prepared.

Available Methods for Using NDS

Method 3:

- This method lets the program accept or build a dynamic query, then process it using the PREPARE command with the DECLARE, OPEN, FETCH, and CLOSE cursor commands.
- The number of select-list items, the number of placeholders for input host variables, and the data types of the input host variables must be known at precompile time.
- Examples:

  -
    ```
    SELECT DEPARTMENT_ID, MIN(SALARY), MAX(SALARY)
    FROM EMPLOYEES
    GROUP BY DEPARTMENT_ID
    ```

  -
    ```
    SELECT LAST_NAME, EMPLOYEE_ID
    FROM EMPLOYEES
    WHERE DEPARTMENT_ID = :dept_number
    ```

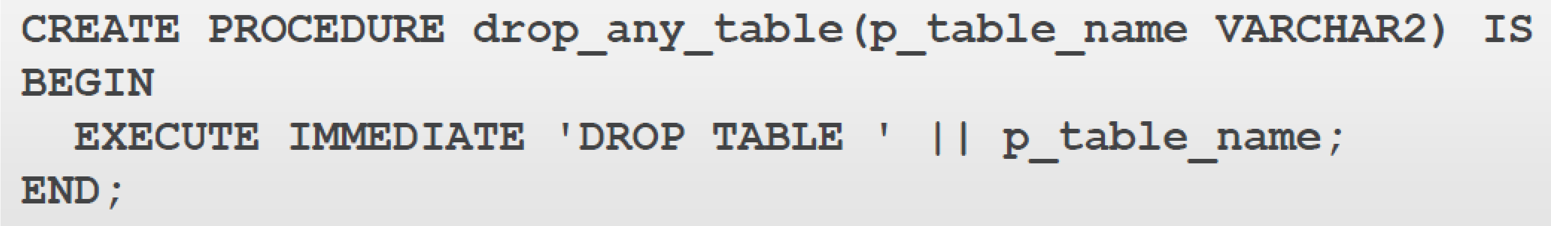
Available Methods for Using NDS

Method 4:

- This method lets your program accept or build a dynamic SQL statement, then process it using descriptors.
- A descriptor is an area of memory used by your program and Oracle to hold a complete description of the variables in a dynamic SQL statement.
- The number of select-list items, the number of placeholders for input host variables, and the data types of the input host variables can be unknown until run time.
- Examples:
  - INSERT INTO EMPLOYEES (<unknown>) VALUES (<unknown>)
  - SELECT <unknown> FROM EMPLOYEES WHERE DEPARTMENT_ID = 20
- Method 4 is required for dynamic SQL statements that contain an unknown number of select-list items or input host variables.
- With this method, you use the DBMS_SQL package.

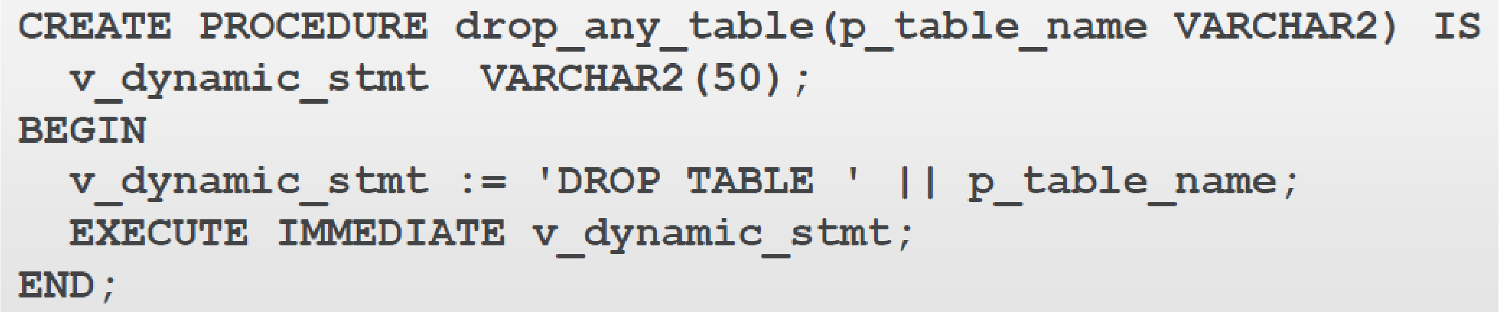
Using Dynamic SQL – Example 1

- Constructing the dynamic statement in-line:

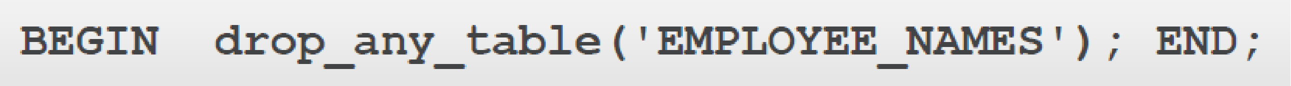


```
CREATE PROCEDURE drop_any_table(p_table_name VARCHAR2) IS
BEGIN
  EXECUTE IMMEDIATE 'DROP TABLE ' || p_table_name;
END;
```

- Constructing the dynamic statement in a variable:



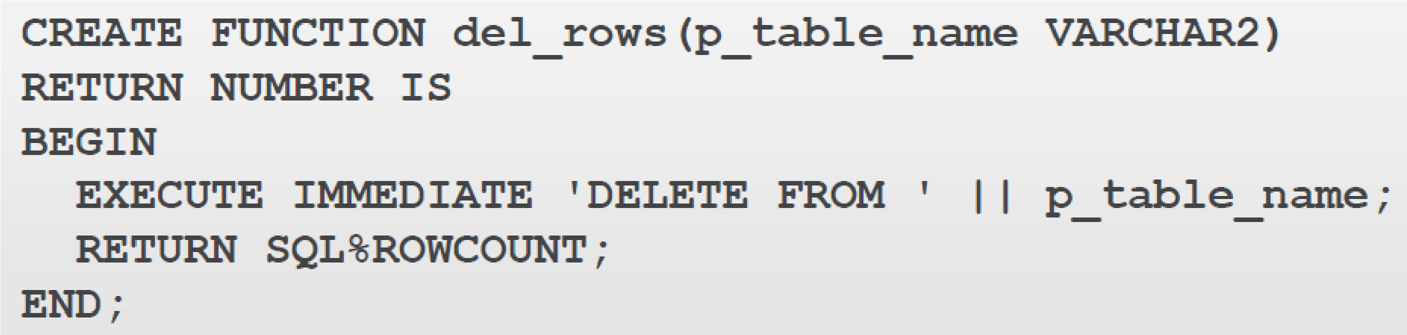
```
CREATE PROCEDURE drop_any_table(p_table_name VARCHAR2) IS
  v_dynamic_stmt  VARCHAR2(50);
BEGIN
  v_dynamic_stmt := 'DROP TABLE ' || p_table_name;
  EXECUTE IMMEDIATE v_dynamic_stmt;
END;
```



```
BEGIN  drop_any_table('EMPLOYEE_NAMES'); END;
```

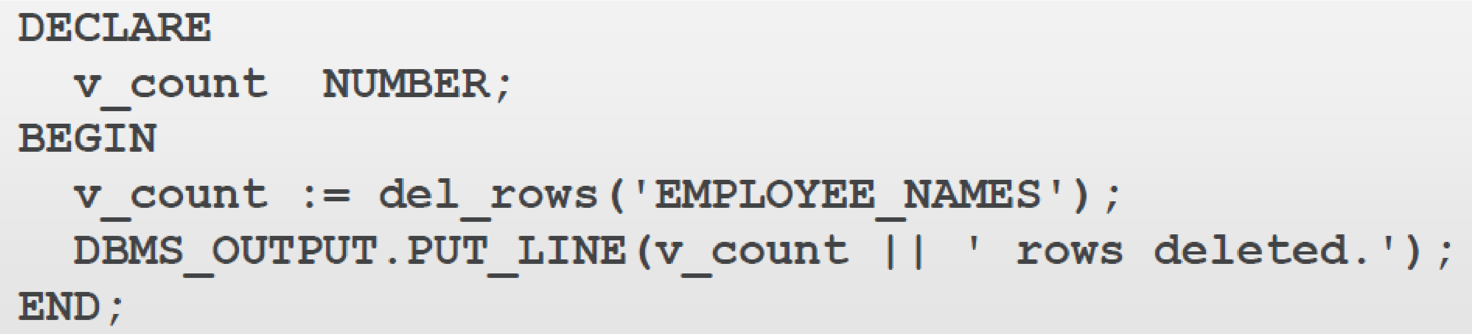
Using Dynamic SQL – Example 2

- Deleting all the rows from any table and returning a count:



```
CREATE FUNCTION del_rows(p_table_name VARCHAR2)
RETURN NUMBER IS
BEGIN
  EXECUTE IMMEDIATE 'DELETE FROM ' || p_table_name;
  RETURN SQL%ROWCOUNT;
END;
```

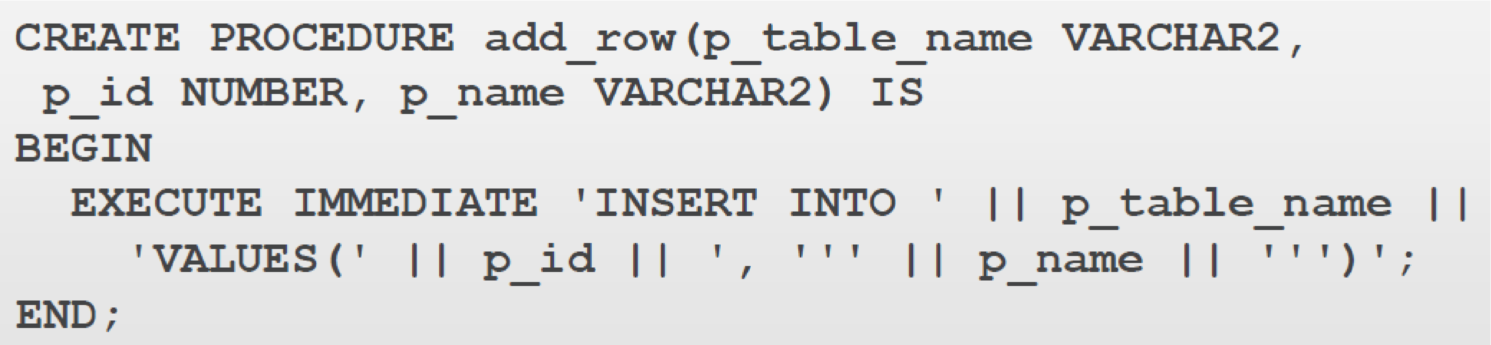
- Invoking the function:



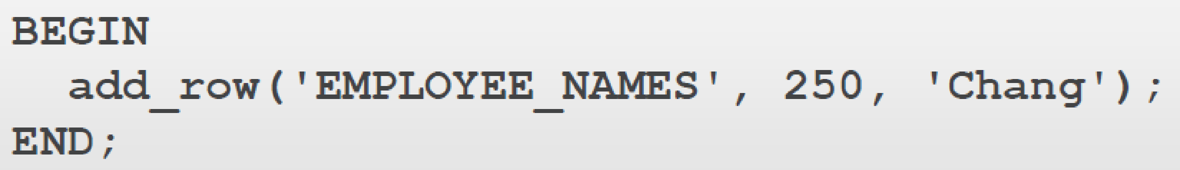
```
DECLARE
  v_count  NUMBER;
BEGIN
  v_count := del_rows('EMPLOYEE_NAMES');
  DBMS_OUTPUT.PUT_LINE(v_count || ' rows deleted.');
END;
```

Using Dynamic SQL – Example 3

– Inserting a row into a table with two columns and invoking the procedure:



```
CREATE PROCEDURE add_row(p_table_name VARCHAR2,
 p_id NUMBER, p_name VARCHAR2) IS
BEGIN
  EXECUTE IMMEDIATE 'INSERT INTO ' || p_table_name ||
     'VALUES(' || p_id || ', ''' || p_name || ''')';
END;
```



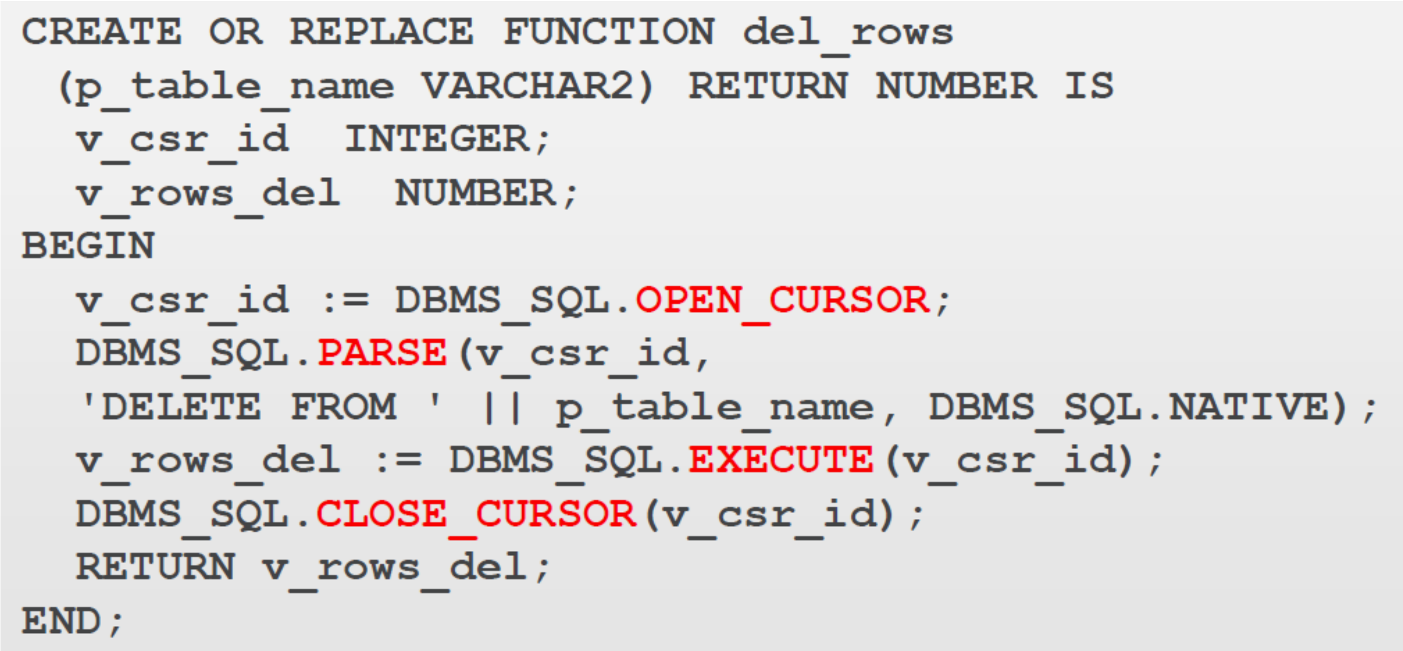
```
BEGIN
  add_row('EMPLOYEE_NAMES', 250, 'Chang');
END;
```

Using the DBMS_SQL Package

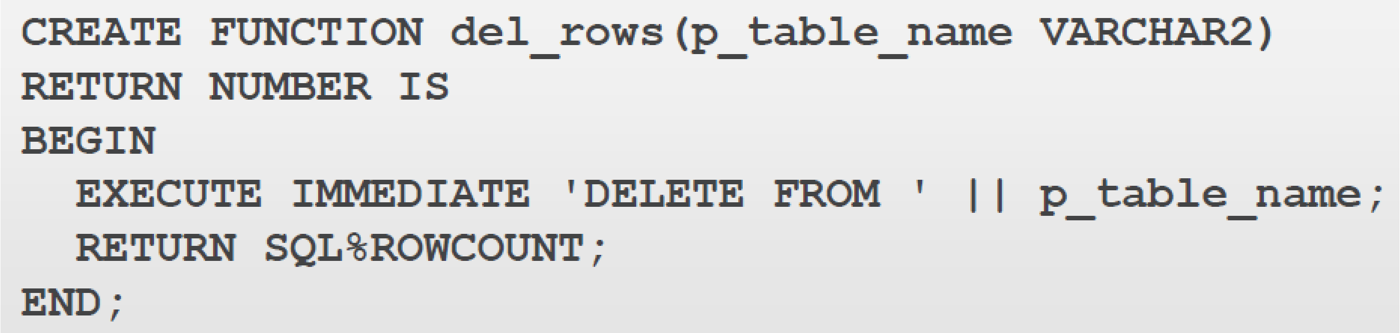
- Some of the procedures and functions of the DBMS_SQL package are:
- OPEN_CURSOR
- PARSE
- BIND_VARIABLE
- EXECUTE
- FETCH_ROWS
- CLOSE_CURSOR

Using DBMS_SQL with a DML Statement

– Example of deleting rows:

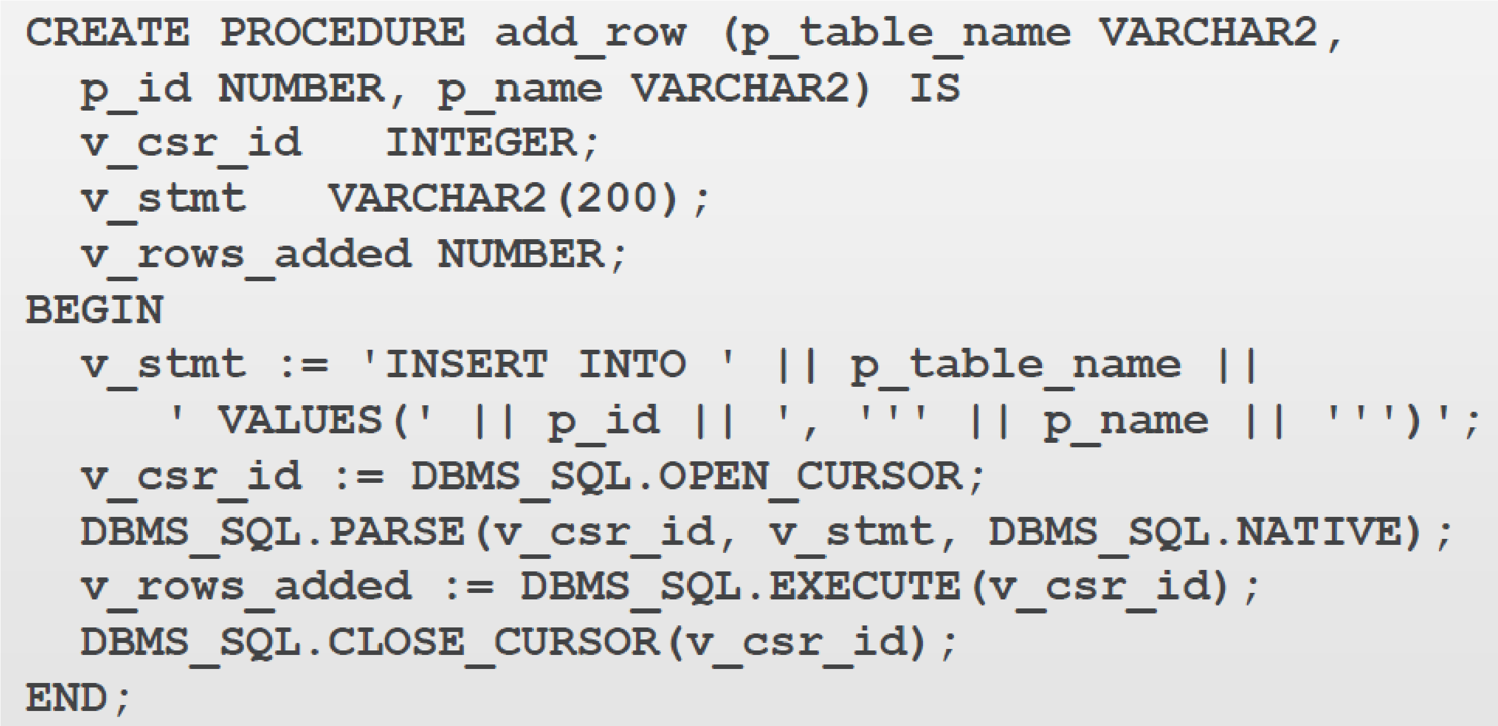


```
CREATE OR REPLACE FUNCTION del_rows
 (p_table_name VARCHAR2) RETURN NUMBER IS
  v_csr_id   INTEGER;
  v_rows_del   NUMBER;
BEGIN
  v_csr_id := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(v_csr_id,
  'DELETE FROM ' || p_table_name, DBMS_SQL.NATIVE);
  v_rows_del := DBMS_SQL.EXECUTE(v_csr_id);
  DBMS_SQL.CLOSE_CURSOR(v_csr_id);
  RETURN v_rows_del;
END;
```

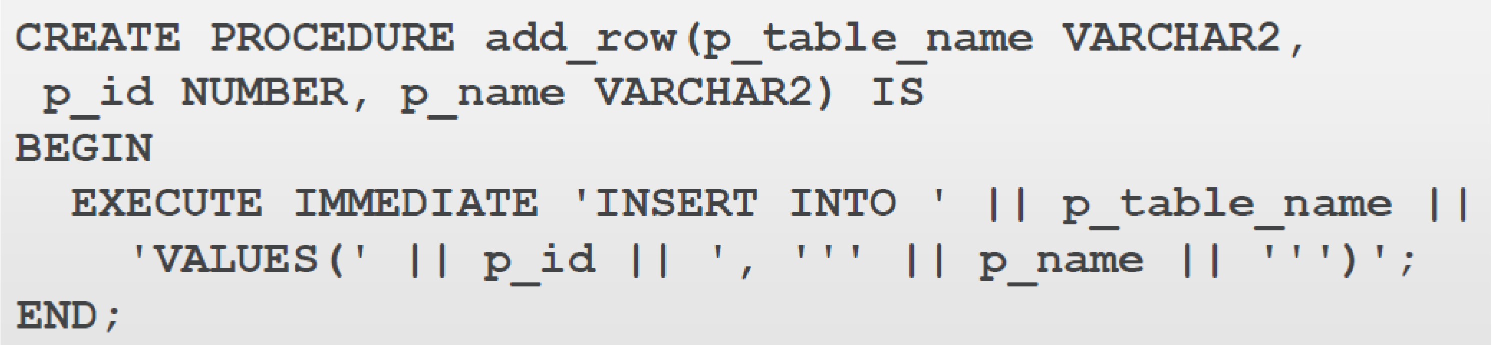


```
CREATE FUNCTION del_rows(p_table_name VARCHAR2)
RETURN NUMBER IS
BEGIN
  EXECUTE IMMEDIATE 'DELETE FROM ' || p_table_name;
  RETURN SQL%ROWCOUNT;
END;
```

Using DBMS_SQL with a Parameterized DML Statement



```
CREATE PROCEDURE add_row (p_table_name VARCHAR2,
  p_id NUMBER, p_name VARCHAR2) IS
  v_csr_id   INTEGER;
  v_stmt   VARCHAR2(200);
  v_rows_added NUMBER;
BEGIN
  v_stmt := 'INSERT INTO ' || p_table_name ||
     ' VALUES(' || p_id || ', ''' || p_name || ''')';
  v_csr_id := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(v_csr_id, v_stmt, DBMS_SQL.NATIVE);
  v_rows_added := DBMS_SQL.EXECUTE(v_csr_id);
  DBMS_SQL.CLOSE_CURSOR(v_csr_id);
END;
```



```
CREATE PROCEDURE add_row(p_table_name VARCHAR2,
 p_id NUMBER, p_name VARCHAR2) IS
BEGIN
  EXECUTE IMMEDIATE 'INSERT INTO ' || p_table_name ||
    'VALUES(' || p_id || ', ''' || p_name || ''')';
END;
```

Using Dynamic SQL

Comparison of Native Dynamic SQL and the DBMS_SQL Package

- Native Dynamic SQL:
  - Is easier to use than DBMS_SQL
  - Requires less code than DBMS_SQL
  - Often executes faster than DBMS_SQL because there are fewer statements to execute.

Improving PL/SQL Performance

ORACLE

Oracle Academy Study Materials

Improving PL/SQL Performance

Purpose:

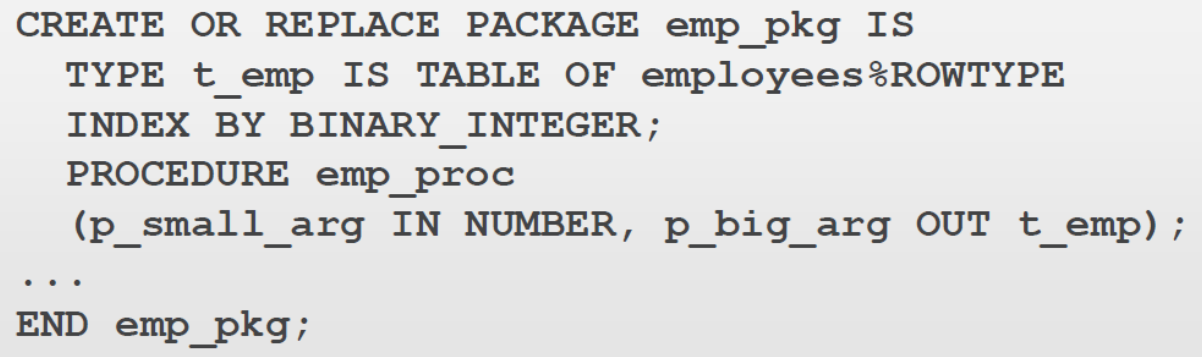
- Until now, you have learned how to write, compile, and execute PL/SQL code without thinking much about how long the execution will take.
- None of the tables in HR schema contain more than a few hundred rows, so the execution is always fast.
- But in real organizations, tables can contain millions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

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,
  - 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.
  - while the subprogram is executing, there are two copies of each argument.
- 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.

Using the NOCOPY Hint

- Example:

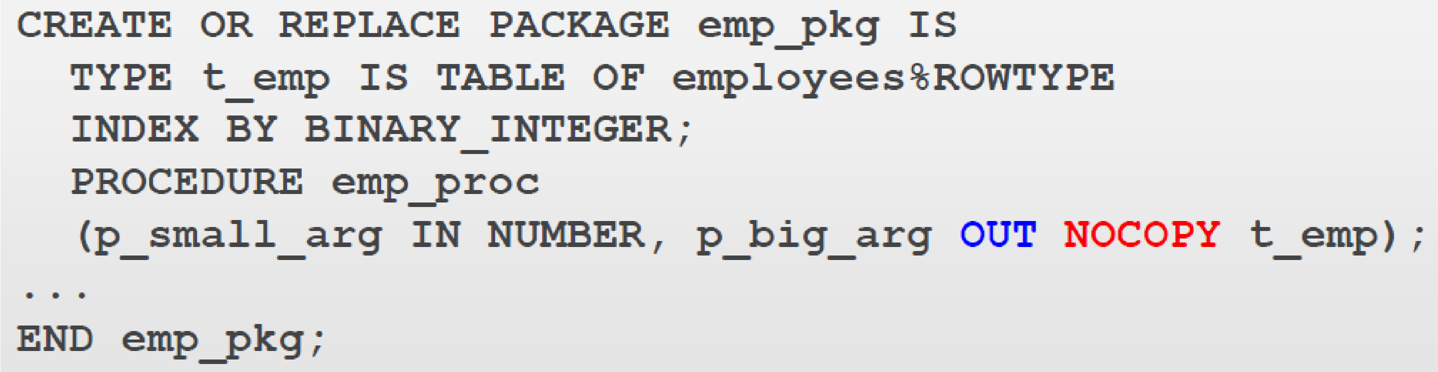


```
CREATE OR REPLACE PACKAGE emp_pkg IS
  TYPE t_emp IS TABLE OF employees%ROWTYPE
  INDEX BY BINARY_INTEGER;
  PROCEDURE emp_proc
  (p_small_arg IN NUMBER, p_big_arg OUT t_emp);
...
END emp_pkg;
```

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

Using the NOCOPY Hint

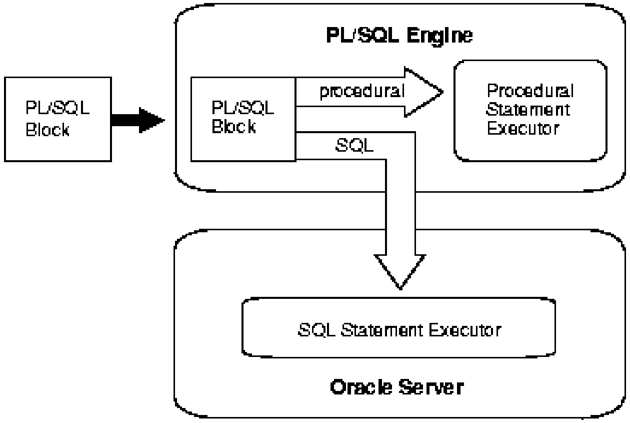
- By default, PL/SQL IN parameter arguments are passed by reference, while OUT and IN OUT arguments are passed by value.
- To pass an OUT or IN OUT argument by reference the NOCOPY hint can be used.



```
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;
```

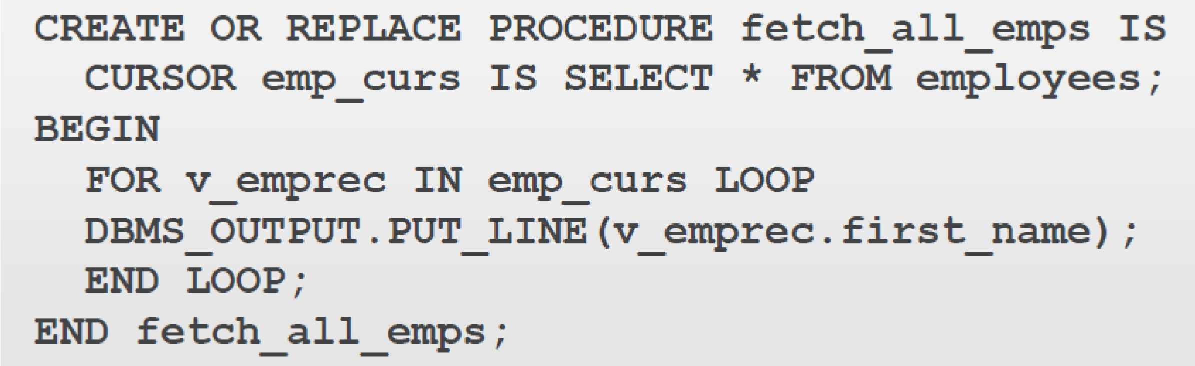
Bulk Binding

- 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, which takes time. For one change, this is at most a few milliseconds, but what if there are millions of changes?



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

Bulk Binding

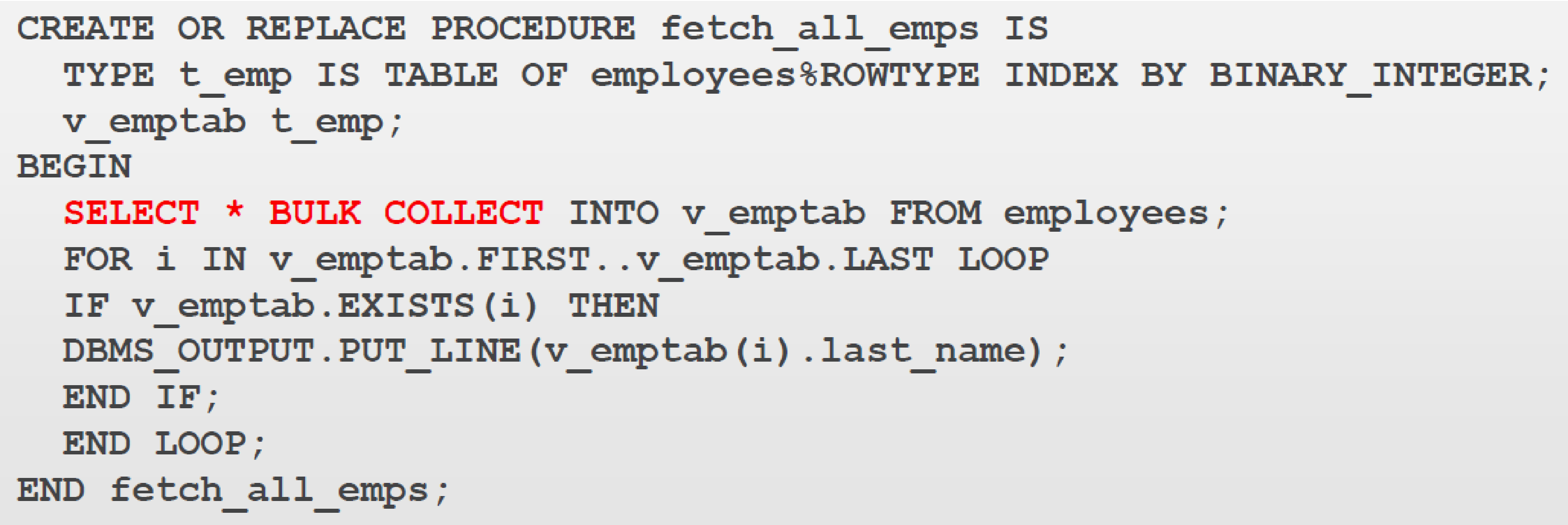


```
CREATE OR REPLACE PROCEDURE fetch_all_emps IS
  CURSOR emp_curs IS SELECT * FROM employees;
BEGIN
  FOR v_emprec IN emp_curs LOOP
  DBMS_OUTPUT.PUT_LINE(v_emprec.first_name);
  END LOOP;
END fetch_all_emps;
```

- 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.
- But: If each row is (on average) 100 bytes in size, storing one million rows will need 100 megabytes of memory.
- So Bulk Binding is a trade-off: more memory required (possibly bad) but faster execution (good).

Bulk Binding

Bulk Binding a SELECT: Using BULK COLLECT:

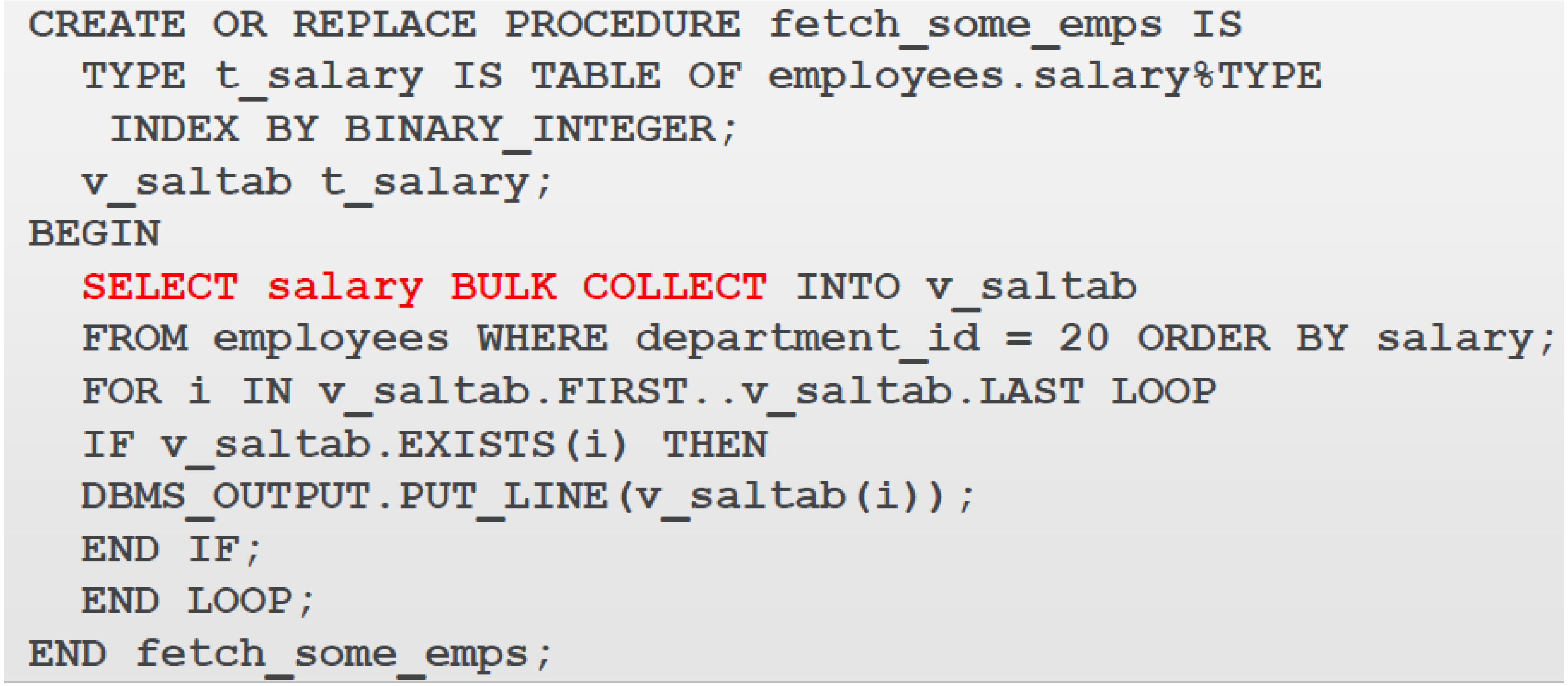


```
CREATE OR REPLACE PROCEDURE fetch_all_emps IS
  TYPE t_emp IS TABLE OF employees%ROWTYPE INDEX BY BINARY_INTEGER;
  v_emptab t_emp;
BEGIN
  SELECT * BULK COLLECT INTO v_emptab FROM employees;
  FOR i IN v_emptab.FIRST..v_emptab.LAST LOOP
  IF v_emptab.EXISTS(i) THEN
  DBMS_OUTPUT.PUT_LINE(v_emptab(i).last_name);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

- 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.

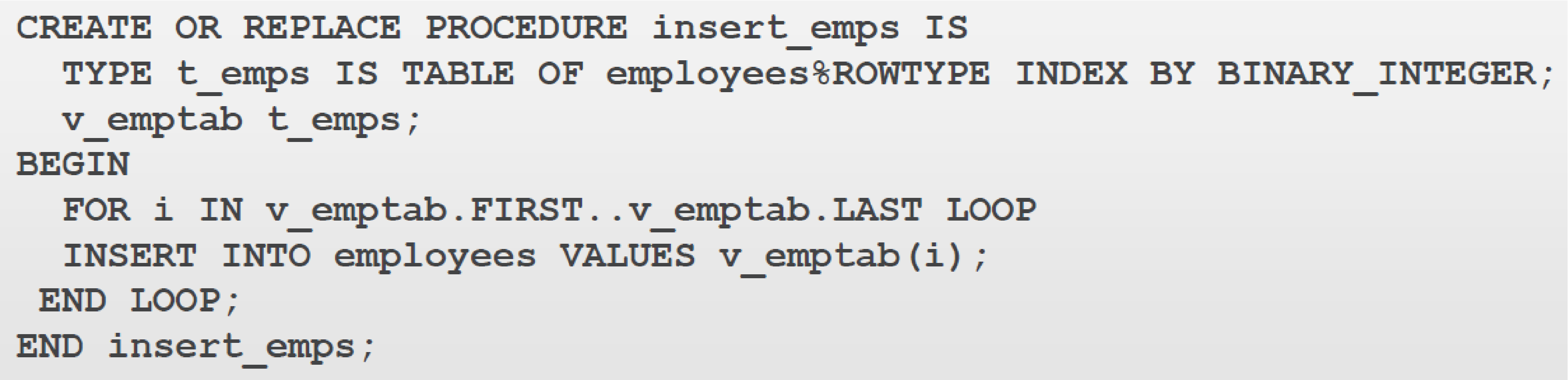
Bulk Binding

Bulk Binding a SELECT: Using BULK COLLECT – Example 2:



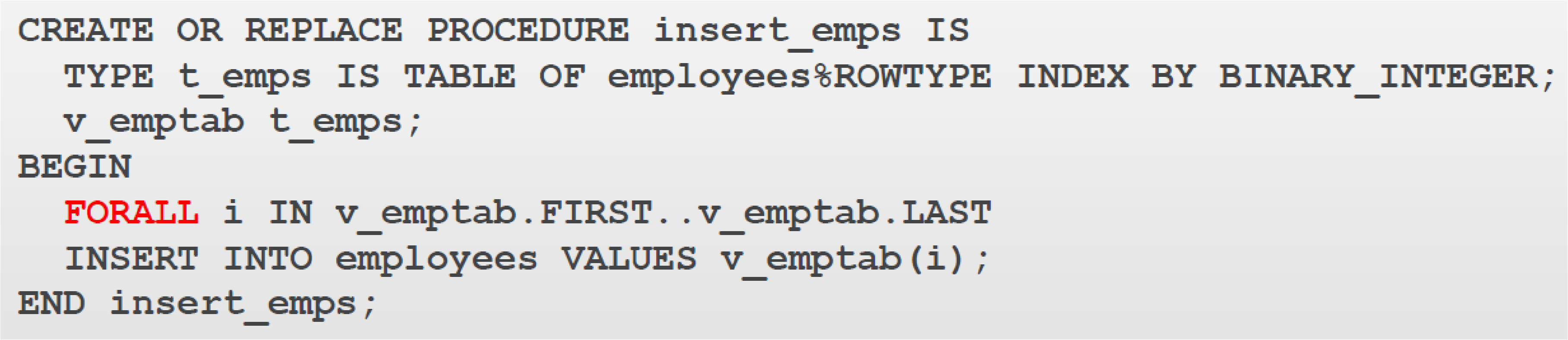
```
CREATE OR REPLACE PROCEDURE fetch_some_emps IS
  TYPE t_salary IS TABLE OF employees.salary%TYPE
   INDEX BY BINARY_INTEGER;
  v_saltab t_salary;
BEGIN
  SELECT salary BULK COLLECT INTO v_saltab
  FROM employees WHERE department_id = 20 ORDER BY salary;
  FOR i IN v_saltab.FIRST..v_saltab.LAST LOOP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

Bulk Binding



```
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;
```

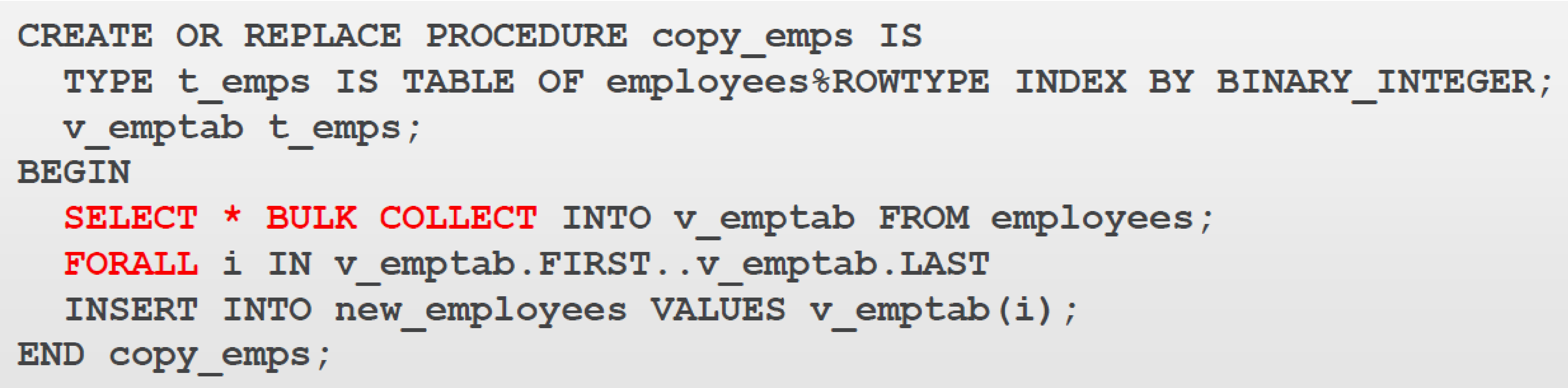
– Bulk Binding with DML: Using FORALL (the code will compile, but will not perform any inserts as the v_emptab table is not populated).



```
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

- We can combine BULK COLLECT and FORALL:

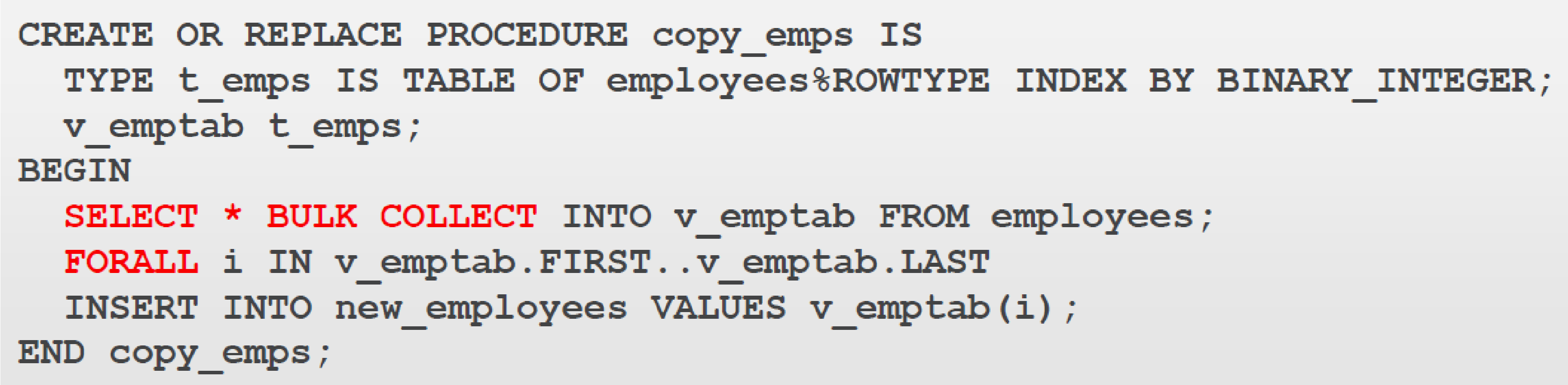


```
CREATE OR REPLACE PROCEDURE copy_emps IS
  TYPE t_emps IS TABLE OF employees%ROWTYPE INDEX BY BINARY_INTEGER;
  v_emptab t_emps;
BEGIN
  SELECT * BULK COLLECT INTO v_emptab FROM employees;
  FORALL i IN v_emptab.FIRST..v_emptab.LAST
  INSERT INTO new_employees VALUES v_emptab(i);
END copy_emps;
```

- Since no columns are specified in the INSERT statement, the record structure of the collection must match the table exactly.
- Bulk binds can also improve the performance when loading collections from queries.

Bulk Binding

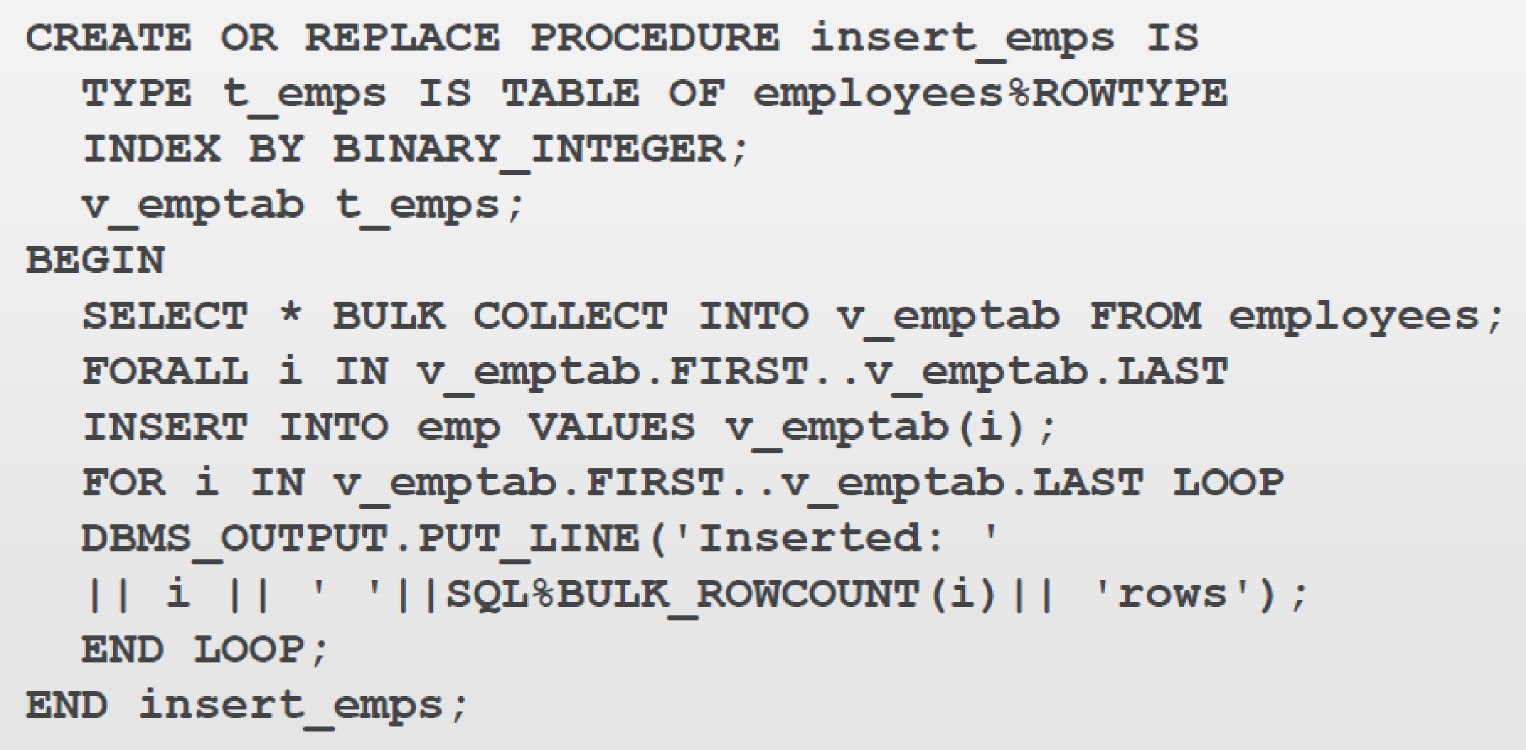
– We can use FORALL with UPDATE and DELETE statements as well as with INSERT:



```
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

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

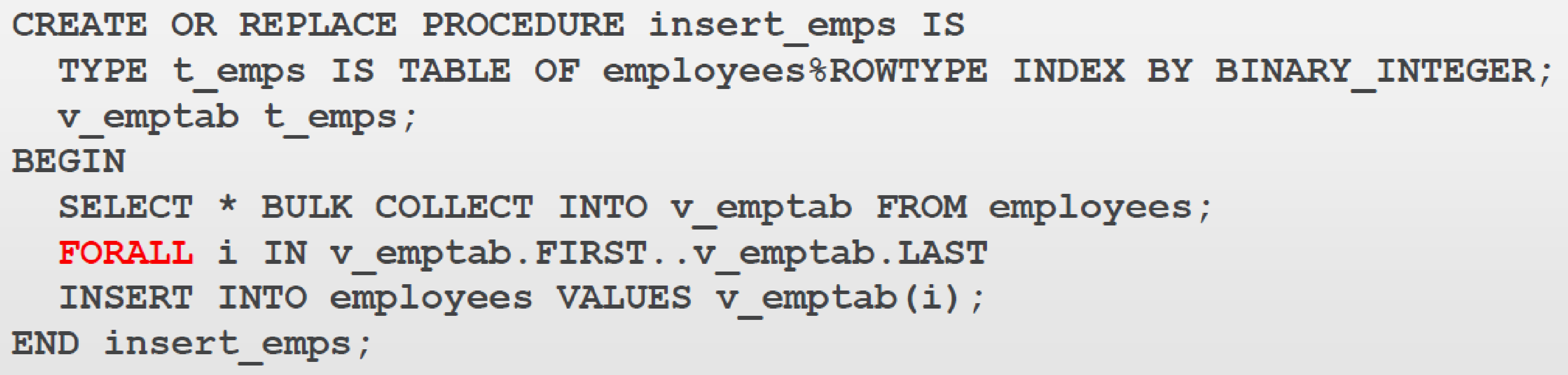


```
CREATE OR REPLACE PROCEDURE insert_emps IS
  TYPE t_emps IS TABLE OF employees%ROWTYPE
  INDEX BY BINARY_INTEGER;
  v_emptab t_emps;
BEGIN
  SELECT * BULK COLLECT INTO v_emptab FROM employees;
  FORALL i IN v_emptab.FIRST..v_emptab.LAST
  INSERT INTO emp VALUES v_emptab(i);
  FOR i IN v_emptab.FIRST..v_emptab.LAST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

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

Bulk Binding

- Bulk Binding Cursor Attributes: SQL%BULK_EXCEPTIONS – example:

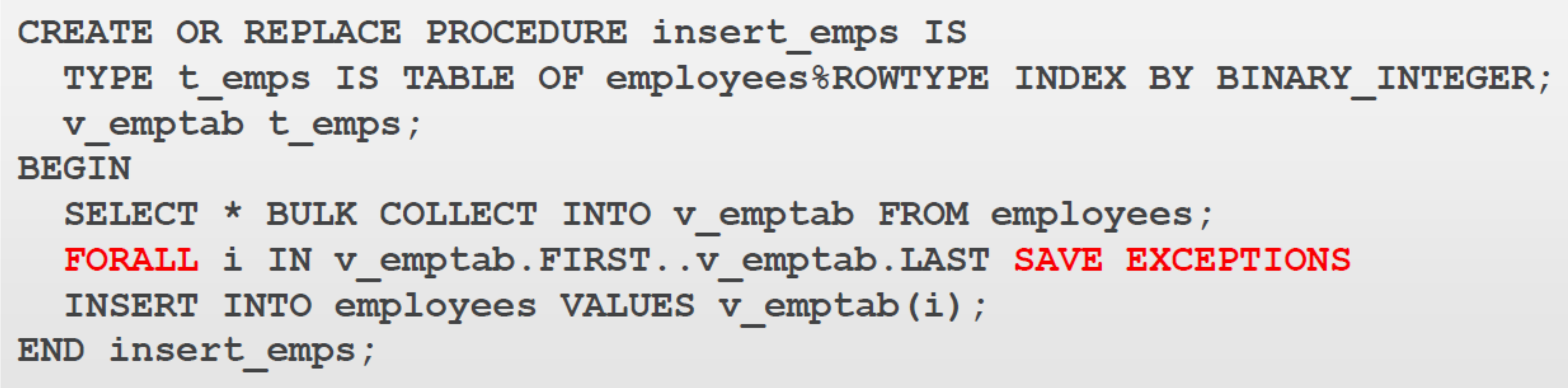


```
CREATE OR REPLACE PROCEDURE insert_emps IS
  TYPE t_emps IS TABLE OF employees%ROWTYPE INDEX BY BINARY_INTEGER;
  v_emptab t_emps;
BEGIN
  SELECT * BULK COLLECT INTO v_emptab FROM employees;
  FORALL i IN v_emptab.FIRST..v_emptab.LAST
  INSERT INTO employees VALUES v_emptab(i);
END insert_emps;
```

- If one of the INSERTs fails, perhaps because a constraint was violated, the whole FORALL statement fails.
- No rows are inserted and there is no information on which row failed to insert.

Bulk Binding

– If SAVE EXCEPTIONS is added to 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;
```

– All the non-violating rows will be inserted.

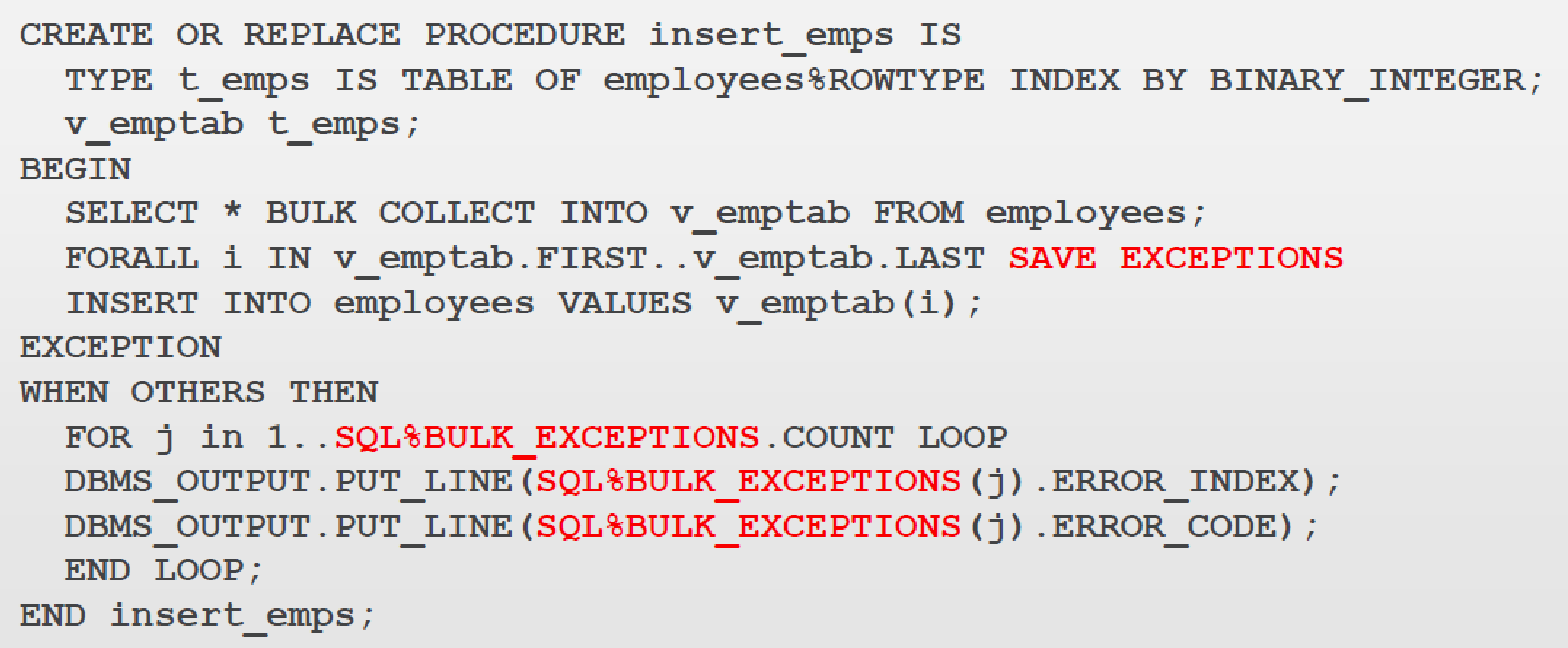
– The violating rows populate an INDEX BY table called SQL%BULK_EXCEPTIONS which has two fields:

  – ERROR_INDEX - shows which inserts failed (first, second, …),

  – ERROR_CODE – shows the Oracle Server predefined error code.

Bulk Binding

– 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;
```

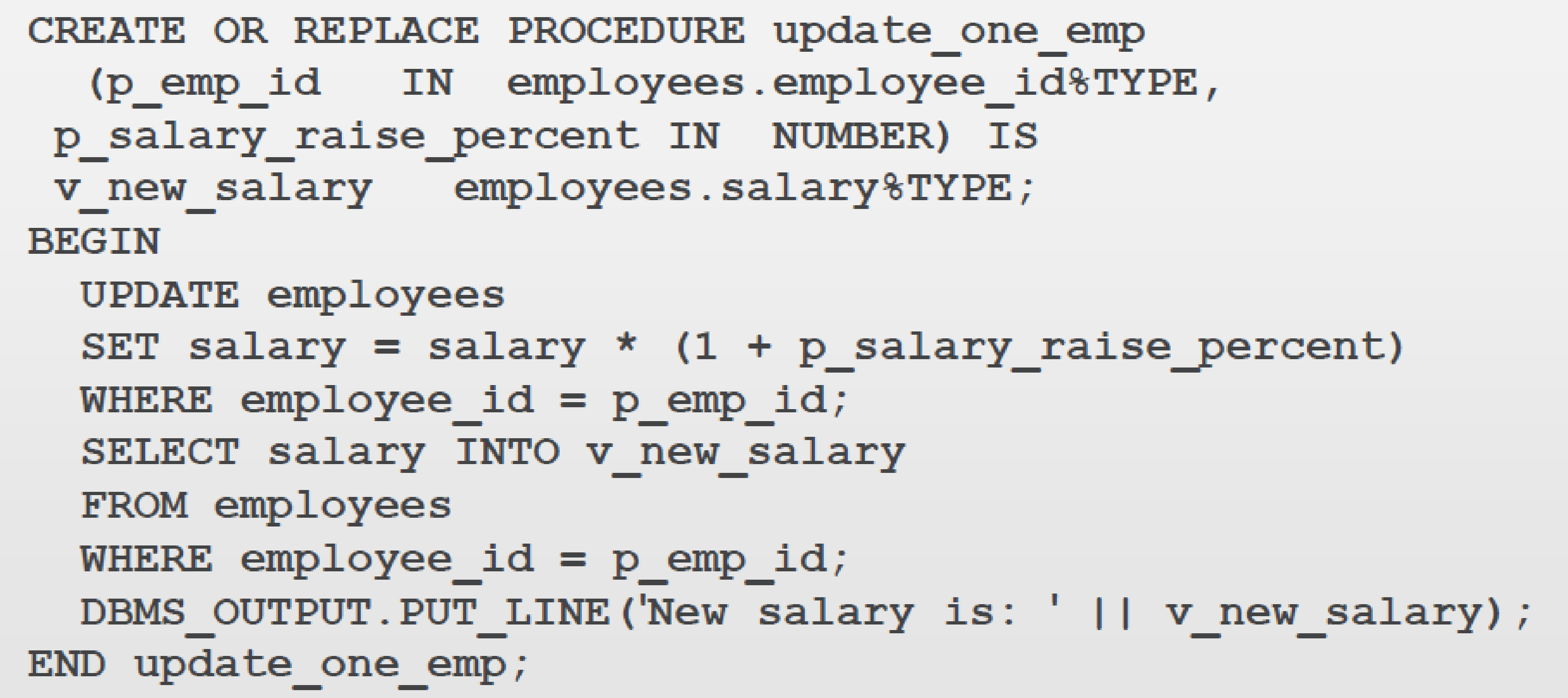
Bulk Binding

SQL%BULK_EXCEPTIONS summary:

- The FORALL statement includes an optional SAVE EXCEPTIONS clause that allows bulk operations to save exception information and continue processing.
- Once the operation is complete, the exception information can be retrieved using the SQL%BULK_EXCEPTIONS attribute.
- This is a collection of exceptions for the most recently executed FORALL statement, with the following two fields for each exception:
  - SQL%BULK_EXCEPTIONS(i).ERROR_INDEX
  - SQL%BULK_EXCEPTIONS(i).ERROR_CODE

Using the RETURNING Clause

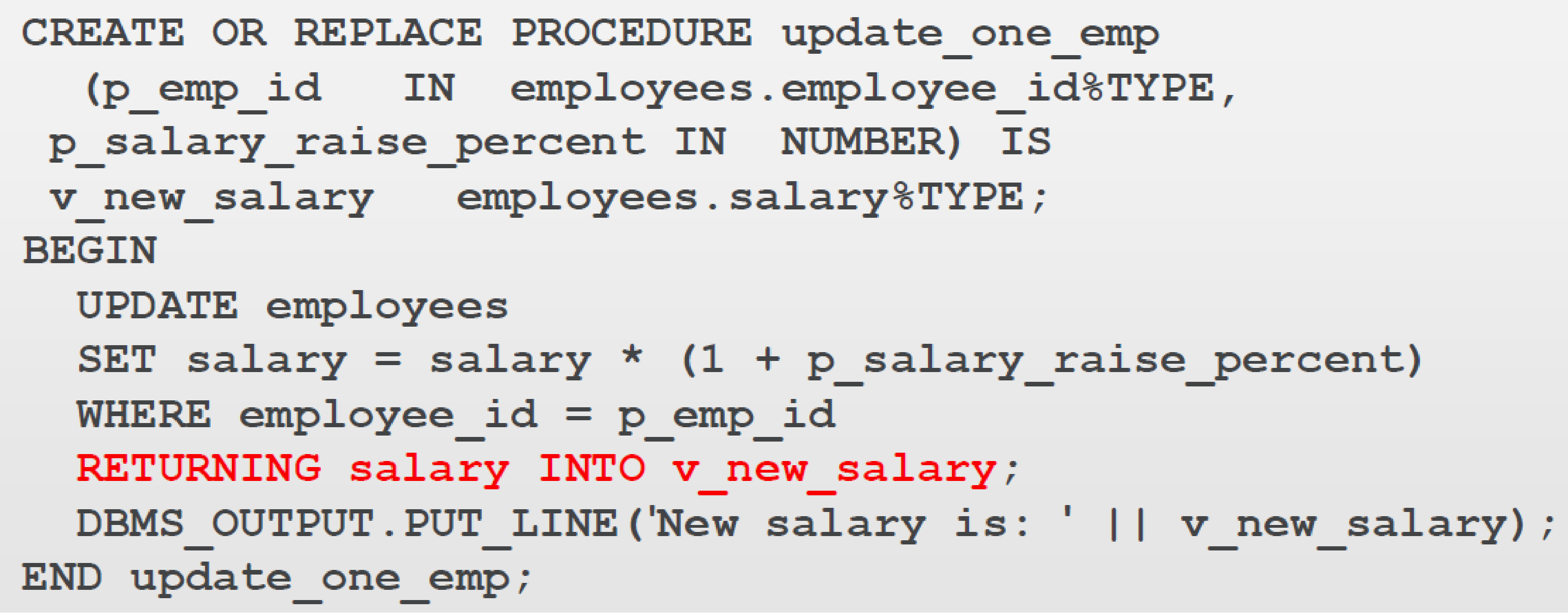
- Sometimes we need to perform a DML operation on a row, and then SELECT column values from the updated row for later use.
- Two SQL statements are required: an UPDATE and a SELECT:



```
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;
```

Using the RETURNING Clause

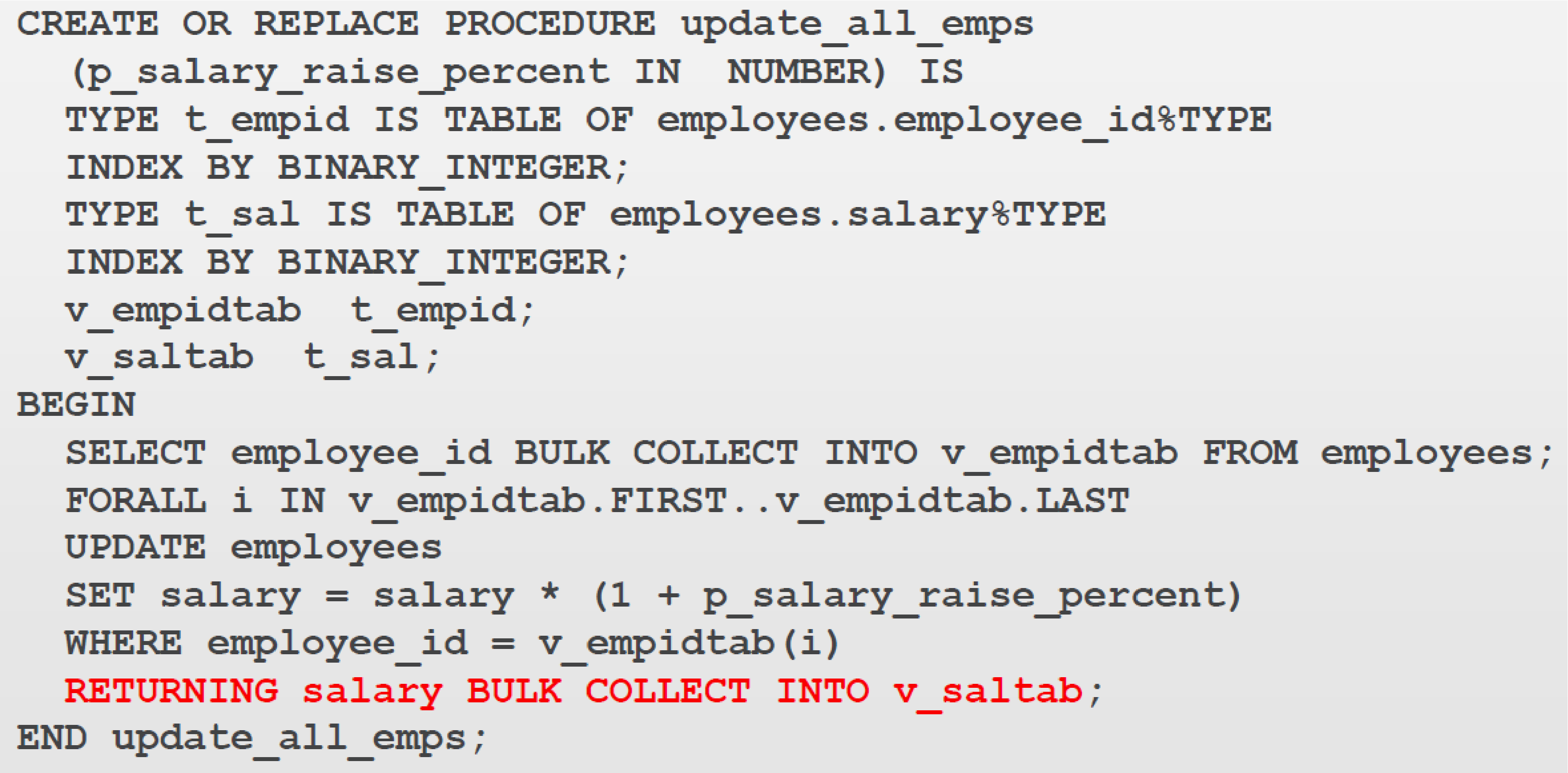
- The SELECT can be done within the UPDATE statement – it is faster because only one call is made to the SQL Engine:



```
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;
```

Using the RETURNING Clause with FORALL

– if we want to update millions of rows and see the updated values, 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;
```

Data Dictionary

ORACLE

Oracle Academy Study Materials

Data Dictionary

- Imagine that you have created many procedures and/or functions, as well as tables and other database objects.

- The **Data Dictionary** remembers this information for you.

- Every Oracle database contains a Data Dictionary – „an automatically-managed master catalog of everything in the database”.
- All database objects, such as tables, views, users and their privileges, procedures or functions are automatically registered in the Data Dictionary when they are created.
- If an object is later altered or dropped, the Dictionary is automatically updated to reflect the change.

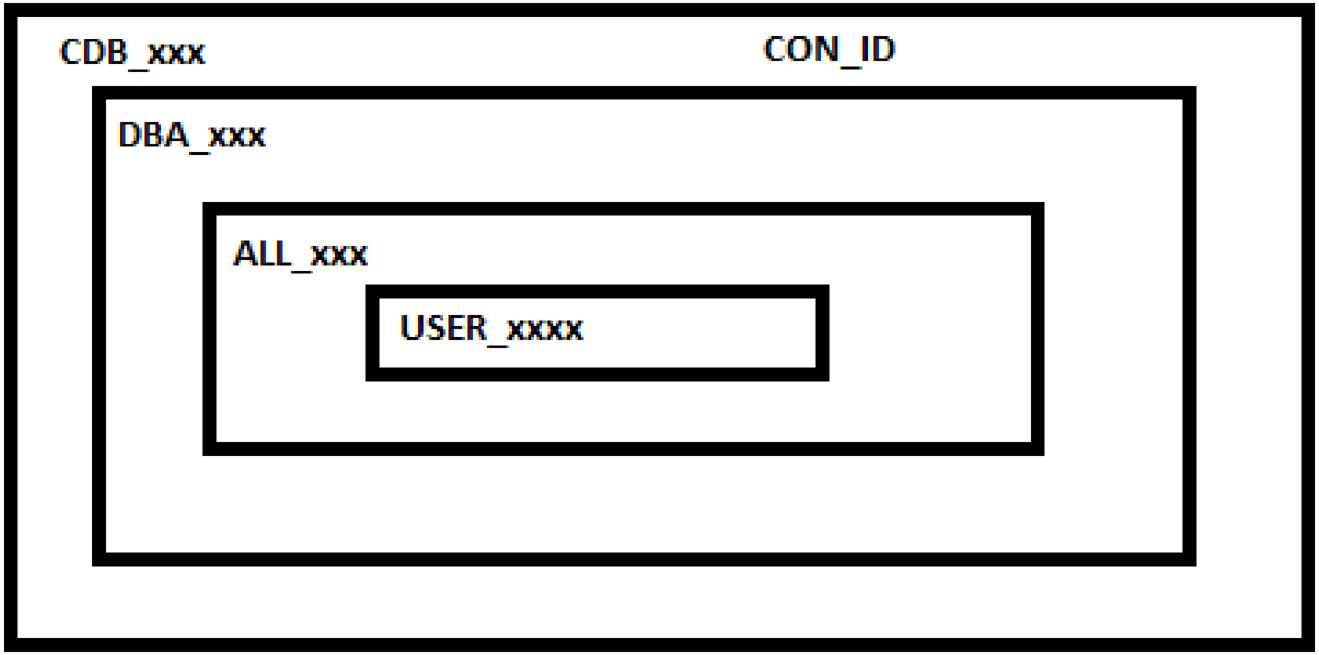
Data Dictionary

There are three classes of dictionary views from which every user can SELECT to view information from the Dictionary:

- The USER_* views contain information about objects that user owns, usually because he created them.
  - Examples: USER_TABLES, USER_INDEXES.
- The ALL_* views contain information about objects that user has privileges to use.
  - These include the USER_* information as a subset, because one always has privileges to use the objects that he owns.
  - Examples: ALL_TABLES, ALL_INDEXES.
- A third class of views one can SELECT to view information from the Dictionary are normally only available to the Database Administrator:
  - The DBA_* tables contain information about everything in the database, no matter who owns them.
  - Examples: DBA_TABLES, DBA_INDEXES.

Data Dictionary

There are three classes of dictionary views from which every user can SELECT to view information from the Dictionary:



Data Dictionary

- Viewing Information in the Dictionary:
- The dictionary should not been modified manually.
- One can DESCRIBE and SELECT from Dictionary tables.

- Example 1: to see information about all the tables that one can use:

```
DESCRIBE ALL_TABLES
```

- Example 2: to see the name and the owner of the tables

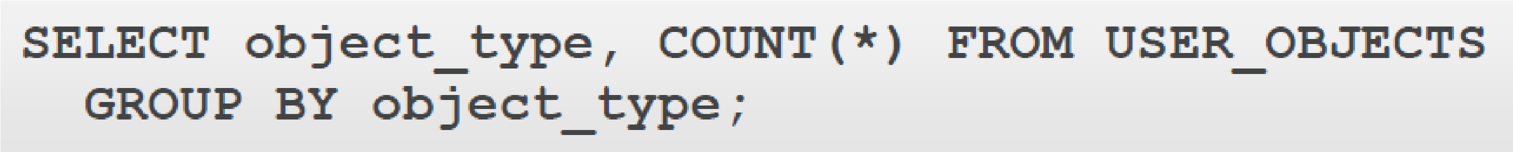
```
SELECT table_name, owner FROM ALL_TABLES;
```

- Example 3:

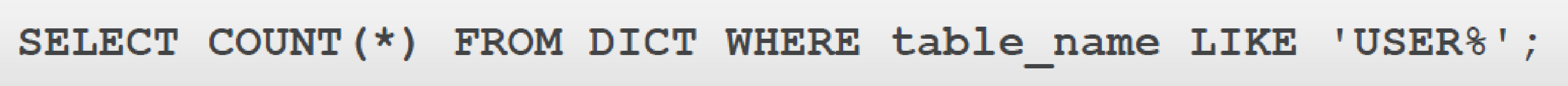
```
SELECT object_type, object_name FROM USER_OBJECTS;
```

Data Dictionary

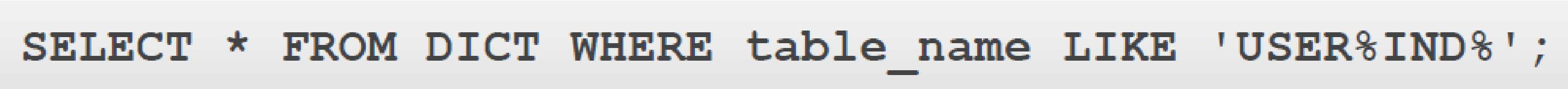
Examples:



```
SELECT object_type, COUNT(*) FROM USER_OBJECTS
  GROUP BY object_type;
```



```
SELECT COUNT(*) FROM DICT WHERE table_name LIKE 'USER%';
```



```
SELECT * FROM DICT WHERE table_name LIKE 'USER%IND%';
```