13 Using Dynamic SQL

✅ Describe the execution flow of SQL statements

SQL Processing 19c

DBMS_SQL execution flow

SQL Processing Oracle7: READ ONLY

SQL execution Burleson: READ ONLY

SQL processing 19c

- SQL processing is the parsing, optimization, row source generation, and execution of a SQL statement

Parsing

- The first stage of SQL processing is parsing
- The parsing stage involves separating the pieces of a SQL statement into a data structure that other routines can process
- The database parses a statement when instructed by the application, which means that only the application, and not the database itself, can reduce the number of parses
- When an application issues a SQL statement, the application makes a parse call to the database to prepare the statement for execution
- The parse call opens or creates a cursor, which is a handle for the session-specific private SQL area that holds a parsed SQL statement and other processing information. The cursor and private SQL area are in the program global area (PGA)
- During the parse call, the database performs checks that identify the errors that can be found before statement execution
- Some errors cannot be caught by parsing. For example, the database can encounter deadlocks or errors in data conversion only during statement execution
- **Syntax check**
  - Oracle Database must check each SQL statement for syntactic validity
  - A statement that breaks a rule for well-formed SQL syntax fails the check. For example, statement fails because the keyword `FROM` is misspelled as FORM
- **Semantic check**
  - The semantics of a statement are its meaning
  - A semantic check determines whether a statement is meaningful, for example, whether the objects and columns in the statement exist

    - A syntactically correct statement can fail a semantic check, e.g. when querying a nonexistent table

- **Shared pool check**
    - During the parse, the database performs a shared pool check to determine whether it can skip resource-intensive steps of statement processing
    - To this end, the database uses a hashing algorithm to generate a hash value for every SQL statement. The statement hash value is the SQL ID shown in V$SQL.SQL_ID
    - This hash value is deterministic within a version of Oracle Database, so the same statement in a single instance or in different instances has the same SQL ID
    - When a user submits a SQL statement, the database searches the shared SQL area to see if an existing parsed statement has the same hash value
    - The hash value of a SQL statement is distinct from the following values:
        - Memory address for the statement
        - Hash value of an execution plan for the statement
    - Parse operations fall into the following categories, depending on the type of statement submitted and the result of the hash check:
        - **Hard parse**
            - If Oracle Database cannot reuse existing code, then it must build a new executable version of the application code. This operation is known as a hard parse, or a library cache miss
            - The database always performs a hard parse of DDL
            - During the hard parse, the database accesses the library cache and data dictionary cache numerous times to check the data dictionary
            - When the database accesses these areas, it uses a serialization device called a latch on required objects so that their definition does not change
            - Latch contention increases statement execution time and decreases concurrency
        - **Soft parse**
            - A soft parse is any parse that is not a hard parse
            - If the submitted statement is the same as a reusable SQL statement in the shared pool, then Oracle Database reuses the existing code. This reuse of code is also called a library cache hit
            - Soft parses can vary in how much work they perform. For example, configuring the session shared SQL area can sometimes reduce the amount of latching in the soft parses, making them "softer."
            - In general, a soft parse is preferable to a hard parse because the database skips the optimization and row source generation steps, proceeding straight to execution
    - If a check determines that a statement in the shared pool has the same hash value, then the database performs semantic and environment checks to determine whether the statements have the same meaning. Identical syntax is not sufficient
    - Even if two statements are semantically identical, an environmental difference can force a hard parse. In this context, the optimizer environment is the totality of session settings that can affect execution plan generation, such as the work area size or optimizer settings (for example, the optimizer mode)

- **SQL optimization**

  - During optimization, Oracle Database must perform a hard parse at least once for every unique DML statement and performs the optimization during this parse
  - The database does not optimize DDL. The only exception is when the DDL includes a DML component such as a subquery that requires optimization

- **SQL row source generation**

  - The row source generator is software that receives the optimal execution plan from the optimizer and produces an iterative execution plan that is usable by the rest of the database
  - The iterative plan is a binary program that, when executed by the SQL engine, produces the result set. The plan takes the form of a combination of steps. Each step returns a row set. The next step either uses the rows in this set, or the last step returns the rows to the application issuing the SQL statement
  - A row source is a row set returned by a step in the execution plan along with a control structure that can iteratively process the rows. The row source can be a table, view, or result of a join or grouping operation
  - The row source generator produces a row source tree, which is a collection of row sources. The row source tree shows the following information:
    - An ordering of the tables referenced by the statement
    - An access method for each table mentioned in the statement
    - A join method for tables affected by join operations in the statement
    - Data operations such as filter, sort, or aggregation

- **SQL execution**

  - During execution, the SQL engine executes each row source in the tree produced by the row source generator. This step is the only mandatory step in DML processing
  - In general, the order of the steps in execution is the reverse of the order in the plan, so you read the plan from the bottom up
  - Each step in an execution plan has an ID number
  - During execution, the database reads the data from disk into memory if the data is not in memory
  - The database also takes out any locks and latches necessary to ensure data integrity and logs any changes made during the SQL execution
  - The final stage of processing a SQL statement is closing the cursor

How oracle database processes DML

- Most DML statements have a query component. In a query, execution of a cursor places the results of the query into a set of rows called the result set
- How Row Sets Are Fetched
  - Result set rows can be fetched either a row at a time or in groups
  - In the fetch stage, the database selects rows and, if requested by the query, orders the rows
  - Each successive fetch retrieves another row of the result until the last row has been fetched
  - In general, the database cannot determine for certain the number of rows to be retrieved by a query until the last row is fetched
  - Oracle Database retrieves the data in response to fetch calls, so that the more rows the database reads, the more work it performs

  - For some queries the database returns the first row as quickly as possible, whereas for others it creates the entire result set before returning the first row
- Read Consistency
  - In general, a query retrieves data by using the Oracle Database read consistency mechanism, which guarantees that all data blocks read by a query are consistent to a single point in time
  - Read consistency uses undo data to show past versions of data. For an example, suppose a query must read 100 data blocks in a full table scan. The query processes the first 10 blocks while DML in a different session modifies block 75. When the first session reaches block 75, it realizes the change and uses undo data to retrieve the old, unmodified version of the data and construct a noncurrent version of block 75 in memory
- Data Changes
  - DML statements that must change data use read consistency to retrieve only the data that matched the search criteria when the modification began
  - Afterward, these statements retrieve the data blocks as they exist in their current state and make the required modifications. The database must perform other actions related to the modification of the data such as generating redo and undo data

How oracle database processes DDL

- Oracle Database processes DDL differently from DML. For example, when you create a table, the database does not optimize the `CREATE TABLE` statement. Instead, Oracle Database parses the DDL statement and carries out the command
- The database processes DDL differently because it is a means of defining an object in the data dictionary
- Typically, Oracle Database must parse and execute many recursive SQL statements to execute a DDL statement. Suppose you create a table as follows: `CREATE TABLE mytable (mycolumn INTEGER);`
- Typically, the database would run dozens of recursive statements to execute the preceding statement. The recursive SQL would perform actions such as the following:
  - Issue a `COMMIT` before executing the `CREATE TABLE` statement
  - Verify that user privileges are sufficient to create the table
  - Determine which tablespace the table should reside in
  - Ensure that the tablespace quota has not been exceeded
  - Ensure that no object in the schema has the same name
  - Insert rows that define the table into the data dictionary
  - Issue a `COMMIT` if the DDL statement succeeded or a `ROLLBACK` if it did not

DBMS_SQL execution flow

1. OPEN_CURSOR

- To process a SQL statement, you must have an open cursor
- When you call the `OPEN_CURSOR` Functions, you receive a cursor ID number for the data structure representing a valid cursor maintained by Oracle
- These cursors are distinct from cursors defined at the precompiler, OCI, or PL/SQL level, and are used only by the `DBMS_SQL` package

2. PARSE

- Every SQL statement must be parsed by calling the `PARSE` procedures

- Parsing the statement checks the statement's syntax and associates it with the cursor in your program
- You can parse any DML or DDL statement. DDL statements are run on the parse, which performs the implied commit

3. BIND_VARIABLE, BIND_VARIABLE_PKG or BIND_ARRAY

- Many DML statements require that data in your program be input to Oracle
- When you define a SQL statement that contains input data to be supplied at runtime, you must use placeholders in the SQL statement to mark where data must be supplied
- For each placeholder in the SQL statement, you must call one of the `BIND_ARRAY` Procedures, or `BIND_VARIABLE` Procedures, or the `BIND_VARIABLE_PKG` Procedure to supply the value of a variable in your program (or the values of an array) to the placeholder
- When the SQL statement is subsequently run, Oracle uses the data that your program has placed in the output and input, or bind variables
- `DBMS_SQL` can run a DML statement multiple times — each time with a different bind variable
- The `BIND_ARRAY` procedure lets you bind a collection of scalars, each value of which is used as an input variable once for each `EXECUTE`. This is similar to the array interface supported by the OCI
- Note that the datatype of the values bound to placeholders cannot be PL/SQL-only datatypes

4. DEFINE_COLUMN, DEFINE_COLUMN_LONG, or DEFINE_ARRAY

- The `DEFINE_COLUMN, DEFINE_COLUMN_LONG, and DEFINE_ARRAY` procedures specify the variables that receive `SELECT` values on a query
- The columns of the row being selected in a `SELECT` statement are identified by their relative positions as they appear in the select list, from left to right
- For a query, you must call one of the define procedures to specify the variables that are to receive the `SELECT` values, much the way an INTO clause does for a static query

5. EXECUTE

- Call the `EXECUTE` Function to run your SQL statement

6. FETCH_ROWS or EXECUTE_AND_FETCH

- The `FETCH_ROWS` Function retrieves the rows that satisfy the query. Each successive fetch retrieves another set of rows, until the fetch is unable to retrieve any more rows
- Instead of calling `EXECUTE` Function and then `FETCH_ROWS`, you may find it more efficient to call `EXECUTE_AND_FETCH` Function if you are calling `EXECUTE` for a single execution

7. VARIABLE_VALUE, VARIABLE_VALUE_PKG, COLUMN_VALUE, or COLUMN_VALUE_LONG

- The type of call determines which procedure or function to use
- For queries, call the `COLUMN_VALUE` Procedure to determine the value of a column retrieved by the `FETCH_ROWS` Function
- For anonymous blocks containing calls to PL/SQL procedures or DML statements with returning clause, call the `VARIABLE_VALUE` Procedures or the `VARIABLE_VALUE_PKG` Procedure to retrieve the values assigned to the output variables when statements were run
- To fetch only part of a `LONG` database column (which can be up to two gigabytes in size), use the `DEFINE_COLUMN_LONG` Procedure. You can specify the offset (in bytes) into the column value, and the number of bytes to fetch

8. CLOSE_CURSOR

- When you no longer need a cursor for a session, close the cursor by calling the `CLOSE_CURSOR` Procedure
- If you are using an Oracle Open Gateway, then you may need to close cursors at other times as well. Consult your Oracle Open Gateway documentation for additional information

SQL Processing Concepts Oracle 7

- Under construction, READ for now

✅Use Native Dynamic SQL (NDS)

Dynamic sql overview: 7 - 7.1

NDS: 7.2

DBMS_SQL package: 7.3

DBMS_SQL package reference: READ ONLY

SQL injection: 7.4 READ ONLY

Overview

- Dynamic SQL is a programming methodology for generating and running SQL statements at run time
- It is useful when writing general-purpose and flexible programs like ad hoc query systems, when writing programs that must run database definition language (DDL) statements, or when you do not know at compile time the full text of a SQL statement or the number or data types of its input and output variables
- PL/SQL provides two ways to write dynamic SQL:
  - Native dynamic SQL, a PL/SQL language (that is, native) feature for building and running dynamic SQL statements
  - `DBMS_SQL` package, an API for building, running, and describing dynamic SQL statements
- Native dynamic SQL code is easier to read and write than equivalent code that uses the DBMS_SQL package, and runs noticeably faster (especially when it can be optimized by the compiler)
- However, to write native dynamic SQL code, you must know at compile time the number and data types of the input and output variables of the dynamic SQL statement
- If you do not know this information at compile time, you must use the DBMS_SQL package
- You must also use the DBMS_SQL package if you want a stored subprogram to return a query result implicitly (not through an OUT REF CURSOR parameter)
- When you need both the DBMS_SQL package and native dynamic SQL, you can switch between them, using the `DBMS_SQL.TO_REFCURSOR` Function and `DBMS_SQL.TO_CURSOR_NUMBER` Function

When you need dynamic SQL

- In PL/SQL, you need dynamic SQL to run:
  - SQL whose text is unknown at compile time. For example, a SELECT statement that includes an identifier that is unknown at compile time (such as a table name) or a WHERE clause in which the number of subclauses is unknown at compile time

  - SQL that is not supported as static SQL
- If you do not need dynamic SQL, use static SQL, which has these advantages:
  - Successful compilation verifies that static SQL statements reference valid database objects and that the necessary privileges are in place to access those objects
  - Successful compilation creates schema object dependencies

Native dynamic SQL

- Native dynamic SQL processes most dynamic SQL statements with the `EXECUTE IMMEDIATE` statement
- If the dynamic SQL statement is a `SELECT` statement that returns multiple rows, native dynamic SQL gives you these choices:
  - Use the `EXECUTE IMMEDIATE` statement with the `BULK COLLECT INTO` clause
  - Use the `OPEN FOR`, `FETCH`, and `CLOSE` statements
- The SQL cursor attributes work the same way after native dynamic SQL `INSERT, UPDATE, DELETE, MERGE`, and single-row `SELECT` statements as they do for their static SQL counterparts

EXECUTE IMMEDIATE statement

- The `EXECUTE IMMEDIATE` statement is the means by which native dynamic SQL processes most dynamic SQL statements

- If the dynamic SQL statement is **self-contained** (that is, if it has no placeholders for bind variables and the only result that it can possibly return is an error), then the `EXECUTE IMMEDIATE` statement needs no clauses

- If the dynamic SQL statement includes placeholders for bind variables, each placeholder must have a corresponding bind variable in the appropriate clause of the `EXECUTE IMMEDIATE` statement, as follows:

  - If the dynamic SQL statement is a `SELECT` statement that can return at most one row, put out-bind variables (defines) in the `INTO` clause and in-bind variables in the `USING` clause
  - If the dynamic SQL statement is a `SELECT` statement that can return multiple rows, put out-bind variables (defines) in the `BULK COLLECT INTO` clause and in-bind variables in the `USING` clause
  - If the dynamic SQL statement is a DML statement without a `RETURNING INTO` clause, other than `SELECT`, put all bind variables in the `USING` clause
  - If the dynamic SQL statement is a DML statement with a `RETURNING INTO` clause, put in-bind variables in the `USING` clause and out-bind variables in the `RETURNING INTO` clause
  - If the dynamic SQL statement is an anonymous PL/SQL block or a `CALL` statement, put all bind variables in the `USING` clause If the dynamic SQL statement invokes a subprogram, ensure that:
    - The subprogram is either created at schema level or declared and defined in a package specification
    - Every bind variable that corresponds to a placeholder for a subprogram parameter has the same parameter mode as that subprogram parameter and a data type that is compatible with that of the subprogram parameter
    - No bind variable is the reserved word NULL. To work around this restriction, use an uninitialized variable where you want to use NULL
    - No bind variable has a data type that SQL does not support (such as associative array indexed by string. When used in dynamic SQL the associative array must be indexed by

PLS_INTEGER). If the data type is a collection or record type, then it must be declared in a package specification

- > Note: Bind variables can be evaluated in any order. If a program determines order of evaluation, then at the point where the program does so, its behavior is undefined

- syntax and semantics:

```
EXECUTE IMMEDIATE dynamic_sql_stmt
[ {into_clause | bulk_collect_into_clause} [using_clause]
| using_clause [dynamic_returning_clause]
| dynamic_returning_clause
]
```

  - using_clause:

    ```
    USING [IN | OUT | IN OUT] bind_argument [,...]
    ```

    - Specifies bind variables, using positional notation
    - > Note: If you repeat placeholder names in dynamic_sql_statement, be aware that the way placeholders are associated with bind variables depends on the kind of dynamic SQL statement
    - Use if and only if dynamic_sql_stmt includes placeholders for bind variables
    - If dynamic_sql_stmt has a `RETURNING INTO` clause (static_returning_clause), then using_clause can contain only `IN` bind variables. The bind variables in the `RETURNING INTO` clause are `OUT` bind variables by definition
    - `IN, OUT, IN OUT`:
      - Parameter modes of bind variables. Default: `IN`
      - An `IN` bind variable passes its value to dynamic_sql_stmt
      - An `OUT` bind variable stores a value that dynamic_sql_stmt returns
      - An `IN OUT` bind variable passes its initial value to dynamic_sql_stmt and stores a value that dynamic_sql_stmt returns
      - For DML a statement with a `RETURNING` clause, you can place `OUT` bind variables in the `RETURNING INTO` clause without specifying the parameter mode, which is always `OUT`
    - bind_argument:
      - An expression whose value replaces its corresponding placeholder in dynamic_sql_stmt at run time
      - Every placeholder in dynamic_sql_stmt must be associated with a bind_argument in the `USING` clause or `RETURNING INTO` clause (or both) or with a define variable in the `INTO` clause
      - You can run dynamic_sql_stmt repeatedly using different values for the bind variables. You incur some overhead, because `EXECUTE IMMEDIATE` prepares the

dynamic string before every execution

- > Note: Bind variables can be evaluated in any order. If a program determines order of evaluation, then at the point where the program does so, its behavior is undefined
- bind_argument cannot be an associative array indexed by string
- bind_argument cannot be the reserved word `NULL`. To pass the value `NULL` to the dynamic SQL statement, use an uninitialized variable where you want to use `NULL`

- dynamic_sql_statement:
  - String literal, string variable, or string expression that represents a SQL statement. Its type must be either `CHAR, VARCHAR2, or CLOB`
  - > Note: If dynamic_sql_statement is a `SELECT` statement, and you omit both into_clause and bulk_collect_into_clause, then execute_immediate_statement never executes
- into_clause:
  - Specifies the variables or record in which to store the column values that the statement returns
  - Use if and only if dynamic_sql_stmt returns a single row
- bulk_collect_into_clause:
  - Specifies one or more collections in which to store the rows that the statement returns
  - Use if and only if dynamic_sql_stmt can return multiple rows
  - dynamic_returning_clause: Returns the column values of the rows affected by the dynamic SQL statement, in either individual variables or records. Use if and only if dynamic_sql_stmt has a `RETURNING INTO` clause

OPEN FOR, FETCH, and CLOSE Statements

- If the dynamic SQL statement represents a `SELECT` statement that returns multiple rows, you can process it with native dynamic SQL as follows:
  1. Use an `OPEN FOR` statement to associate a cursor variable with the dynamic SQL statement. In the `USING` clause of the `OPEN FOR` statement, specify a bind variable for each placeholder in the dynamic SQL statement. The `USING` clause cannot contain the literal `NULL`
  2. Use the `FETCH` statement to retrieve result set rows one at a time, several at a time, or all at once
  3. Use the `CLOSE` statement to close the cursor variable

Repeated placeholder names in dynamic SQL statements

- If you repeat placeholder names in dynamic SQL statements, be aware that the way placeholders are associated with bind variables depends on the kind of dynamic SQL statement

- Dynamic SQL Statement is Not Anonymous Block or `CALL` Statement

  - If the dynamic SQL statement does not represent an anonymous PL/SQL block or a `CALL` statement, repetition of placeholder names is insignificant

  - Placeholders are associated with bind variables in the `USING` clause by position, not by name
  - For example, in this dynamic SQL statement, the repetition of the name :x is insignificant:

    ```
    sql_stmt := 'INSERT INTO payroll VALUES (:x, :x, :y, :x)';
    ```

  - In the corresponding `USING` clause, you must supply four bind variables. They can be different; for example:

    ```
    EXECUTE IMMEDIATE sql_stmt USING a, b, c, d;
    ```

  - The preceding `EXECUTE IMMEDIATE` statement runs this SQL statement:

    ```
    INSERT INTO payroll VALUES (a, b, c, d)
    ```

  - To associate the same bind variable with each occurrence of :x, you must repeat that bind variable; for example:

    ```
    EXECUTE IMMEDIATE sql_stmt USING a, a, b, a;
    ```

  - The preceding `EXECUTE IMMEDIATE` statement runs this SQL statement:

    ```
    INSERT INTO payroll VALUES (a, a, b, a)
    ```

- Dynamic SQL Statement is Anonymous Block or CALL Statement
  - If the dynamic SQL statement represents an anonymous PL/SQL block or a `CALL` statement, repetition of placeholder names is significant
  - Each unique placeholder name must have a corresponding bind variable in the `USING` clause
  - If you repeat a placeholder name, you need not repeat its corresponding bind variable. All references to that placeholder name correspond to one bind variable in the `USING` clause
  - Example

    ```
    CREATE PROCEDURE calc_stats (
      w NUMBER,
      x NUMBER,
      y NUMBER,
      z NUMBER )
    ```

```
IS
BEGIN
  DBMS_OUTPUT.PUT_LINE(w + x + y + z);
END;
/
DECLARE
  a NUMBER := 4;
  b NUMBER := 7;
  plsql_block VARCHAR2(100);
BEGIN
  plsql_block := 'BEGIN calc_stats(:x, :x, :y, :x); END;';
  EXECUTE IMMEDIATE plsql_block USING a, b;  -- calc_stats(a, a, b, a)
END;
/
```

DBMS_SQL Package

- The `DBMS_SQL` package defines an entity called a SQL cursor number. Because the SQL cursor number is a PL/SQL integer, you can pass it across call boundaries and store it
- You must use the `DBMS_SQL` package to run a dynamic SQL statement if any of the following are true:
  - You do not know the `SELECT` list until run time
  - You do not know until run time what placeholders in a `SELECT` or DML statement must be bound
  - You want a stored subprogram to return a query result implicitly (not through an `OUT REF CURSOR` parameter), which requires the `DBMS_SQL.RETURN_RESULT` procedure
- In these situations, you must use native dynamic SQL instead of the `DBMS_SQL` package:
  - The dynamic SQL statement retrieves rows into records
  - You want to use the SQL cursor attribute `%FOUND`, `%ISOPEN`, `%NOTFOUND`, or `%ROWCOUNT` after issuing a dynamic SQL statement that is an `INSERT`, `UPDATE`, `DELETE`, `MERGE`, or single-row `SELECT` statement
- When you need both the `DBMS_SQL` package and native dynamic SQL, you can switch between them, using the functions `DBMS_SQL.TO_REFCURSOR` and `DBMS_SQL.TO_CURSOR_NUMBER`

- > Note: You can invoke DBMS_SQL subprograms remotely

DBMS_SQL.RETURN_RESULT Procedure

- The `DBMS_SQL.RETURN_RESULT` procedure lets a stored subprogram return a query result implicitly to either the client program (which invokes the subprogram indirectly) or the immediate caller of the subprogram
- After `DBMS_SQL.RETURN_RESULT` returns the result, only the recipient can access it
- The `DBMS_SQL.RETURN_RESULT` has two overloads: The rc parameter is either an open cursor variable (`SYS_REFCURSOR`) or the cursor number (`INTEGER`) of an open cursor
- To open a cursor and get its cursor number, invoke the `DBMS_SQL.OPEN_CURSOR` function
- When the to_client parameter is `TRUE` (the default), the `DBMS_SQL.RETURN_RESULT` procedure returns the query result to the client program (which invokes the subprogram indirectly); when this parameter is `FALSE`, the procedure returns the query result to the subprogram's immediate caller

DBMS_SQL.GET_NEXT_RESULT Procedure

- The `DBMS_SQL.GET_NEXT_RESULT` procedure gets the next result that the `DBMS_SQL.RETURN_RESULT` procedure returned to the recipient. The two procedures return results in the same order
- The `DBMS_SQL.GET_NEXT_RESULT` has two overloads: The rc parameter is either a cursor variable (`SYS_REFCURSOR`) or the cursor number (`INTEGER`) of an open cursor
- The c parameter is the cursor number of an open cursor that directly or indirectly invokes a subprogram that uses the `DBMS_SQL.RETURN_RESULT` procedure to return a query result implicitly
- To open a cursor and get its cursor number, invoke the `DBMS_SQL.OPEN_CURSOR` function
- `DBMS_SQL.OPEN_CURSOR` has an optional parameter, treat_as_client_for_results. When this parameter is `FALSE` (the default), the caller that opens this cursor (to invoke a subprogram) is not treated as the client that receives query results for the client from the subprogram that uses `DBMS_SQL.RETURN_RESULT`—those query results are returned to the client in a upper tier instead. When this parameter is `TRUE`, the caller is treated as the client

DBMS_SQL.TO_REFCURSOR Function

- The `DBMS_SQL.TO_REFCURSOR` function converts a SQL cursor number to a weak cursor variable, which you can use in native dynamic SQL statements
- Before passing a SQL cursor number to the `DBMS_SQL.TO_REFCURSOR` function, you must `OPEN, PARSE, and EXECUTE` it (otherwise an error occurs)
- After you convert a SQL cursor number to a `REF CURSOR` variable, `DBMS_SQL` operations can access it only as the `REF CURSOR` variable, not as the SQL cursor number. For example, using the `DBMS_SQL.IS_OPEN` function to see if a converted SQL cursor number is still open causes an error

DBMS_SQL.TO_CURSOR_NUMBER Function

- The `DBMS_SQL.TO_CURSOR_NUMBER` function converts a `REF CURSOR` variable (either strong or weak) to a SQL cursor number, which you can pass to DBMS_SQL subprograms
- Before passing a `REF CURSOR` variable to the `DBMS_SQL.TO_CURSOR_NUMBER` function, you must `OPEN` it
- After you convert a `REF CURSOR` variable to a SQL cursor number, native dynamic SQL operations cannot access it

✅Bind PL/SQL types in SQL statements

see NDS in section above