Course Title: Database Systems

(M.C.A 1st Semester)

(Evening Batch)

UNIT 2

Set Operators

Set operators are used to join the results of two (or more) SELECT statements.The SET operators available in Oracle 11g are UNION,UNION ALL,INTERSECT,and MINUS.

The UNION set operator returns the combined results of the two SELECT statements.Essentially,it removes duplicates from the results i.e. only one row will be listed for each duplicated result.To counter this behavior,use the UNION ALL set operator which retains the duplicates in the final result.INTERSECT lists only records that are common to both the SELECT queries; the MINUS set operator removes the second query's results from the output if they are also found in the first query's results. INTERSECT and MINUS set operations produce unduplicated results.

All the SET operators share the same degree of precedence among them.Instead,during query execution, Oracle starts evaluation from left to right or from top to bottom.If explicitly parentheses are used, then the order may differ as parentheses would be given priority over dangling operators.

Points to remember -

- Same number of columns must be selected by all participating SELECT statements.Column names used in the display are taken from the first query.
- Data types of the column list must be compatible/implicitly convertible by oracle. Oracle will not perform implicit type conversion if corresponding columns in the component queries belong to different data type groups.For example, if a column in the first component query is of data type DATE, and the corresponding column in the second component query is of data type CHAR,Oracle will not perform implicit conversion, but raise ORA-01790 error.
- Positional ordering must be used to sort the result set. Individual result set ordering is not allowed with Set operators. ORDER BY can appear once at the end of the query. For example,
- UNION and INTERSECT operators are commutative, i.e. the order of queries is not important; it doesn't change the final result.
- Performance wise, UNION ALL shows better performance as compared to UNION because resources are not wasted in filtering duplicates and sorting the result set.
- Set operators can be the part of sub queries.
- Set operators can't be used in SELECT statements containing TABLE collection expressions.
- The LONG, BLOB, CLOB, BFILE, VARRAY,or nested table are not permitted for use in Set operators.For update clause is not allowed with the set operators.

UNION

When multiple SELECT queries are joined using UNION operator, Oracle displays the combined result from all the compounded SELECT queries,after removing all duplicates and in sorted order (ascending by default), without ignoring the NULL values.

UNION ALL

UNION and UNION ALL are similar in their functioning with a slight difference. But UNION ALL gives the result set without removing duplication and sorting the data.

INTERSECT

Using INTERSECT operator, Oracle displays the common rows from both the SELECT statements, with no duplicates and data arranged in sorted order (ascending by default).

MINUS

Minus operator displays the rows which are present in the first query but absent in the second query, with no duplicates and data arranged in ascending order by default.

Using ORDER BY clause in SET operations

The ORDER BY clause can appear only once at the end of the query containing compound SELECT statements.It implies that individual SELECT statements cannot have ORDER BY clause. Additionally, the sorting can be based on the columns which appear in the first SELECT query only.

Locks

In a multiprogramming environment where multiple transactions can be executed simultaneously, it is highly important to control the concurrency of transactions. We have concurrency control protocols to ensure atomicity, isolation, and serializability of concurrent transactions. Concurrency control protocols can be broadly divided into two categories −

- Lock based protocols
- Time stamp based protocols

Lock-based Protocols

Database systems equipped with lock-based protocols use a mechanism by which any transaction cannot read or write data until it acquires an appropriate lock on it. Locks are of two kinds −

- **Binary Locks** − A lock on a data item can be in two states; it is either locked or unlocked.
- **Shared/exclusive** − This type of locking mechanism differentiates the locks based on their uses. If a lock is acquired on a data item to perform a write operation, it is an exclusive lock. Allowing more than one transaction to write on the same data item would lead the database into an inconsistent state. Read locks are shared because no data value is being changed.

There are four types of lock protocols available −

Simplistic Lock Protocol

Simplistic lock-based protocols allow transactions to obtain a lock on every object before a 'write' operation is performed. Transactions may unlock the data item after completing the ‘write’ operation.

Pre-claiming Lock Protocol

Pre-claiming protocols evaluate their operations and create a list of data items on which they need locks. Before initiating an execution, the transaction requests the system for all the locks it needs beforehand. If all the locks are granted, the transaction executes and releases all the locks when all its operations are over. If all the locks are not granted, the transaction rolls back and waits until all the locks are granted.

Two-Phase Locking 2PL

This locking protocol divides the execution phase of a transaction into three parts. In the first part, when the transaction starts executing, it seeks permission for the locks it requires. The second part is where the transaction acquires all the locks. As soon as the transaction releases its first lock, the third phase starts. In this phase, the transaction cannot demand any new locks; it only releases the acquired locks.

Two-phase locking has two phases, one is **growing**, where all the locks are being acquired by the transaction; and the second phase is shrinking, where the locks held by the transaction are being released.

To claim an exclusive (write) lock, a transaction must first acquire a shared (read) lock and then upgrade it to an exclusive lock.

Strict Two-Phase Locking

The first phase of Strict-2PL is same as 2PL. After acquiring all the locks in the first phase, the transaction continues to execute normally. But in contrast to 2PL, Strict-2PL does not release a lock after using it. Strict-2PL holds all the locks until the commit point and releases all the locks at a time. Strict-2PL does not have cascading abort as 2PL does.

Timestamp-based Protocols

The most commonly used concurrency protocol is the timestamp based protocol. This protocol uses either system time or logical counter as a timestamp.

Lock-based protocols manage the order between the conflicting pairs among transactions at the time of execution, whereas timestamp-based protocols start working as soon as a transaction is created.

Every transaction has a timestamp associated with it, and the ordering is determined by the age of the transaction. A transaction created at 0002 clock time would be older than all other transactions that come after it. For example, any transaction 'y' entering the system at 0004 is two seconds younger and the priority would be given to the older one.

In addition, every data item is given the latest read and write-timestamp. This lets the system know when the last ‘read and write’ operation was performed on the data item.

Timestamp Ordering Protocol

The timestamp-ordering protocol ensures serializability among transactions in their conflicting read and write operations. This is the responsibility of the protocol system that the conflicting pair of tasks should be executed according to the timestamp values of the transactions.

- The timestamp of transaction T_i is denoted as $TS(T_i)$.
- Read time-stamp of data-item X is denoted by R-timestamp(X).
- Write time-stamp of data-item X is denoted by W-timestamp(X).

Timestamp ordering protocol works as follows −

- **If a transaction Ti issues a read(X) operation −**
    - If TS(Ti) < W-timestamp(X)
        - Operation rejected.
    - If TS(Ti) >= W-timestamp(X)
        - Operation executed.
    - All data-item timestamps updated.
- **If a transaction Ti issues a write(X) operation −**
    - If TS(Ti) < R-timestamp(X)
        - Operation rejected.
    - If TS(Ti) < W-timestamp(X)

- Operation rejected and Ti rolled back.
- Otherwise, operation executed.

Thomas' Write Rule

This rule states if TS(Ti) < W-timestamp(X), then the operation is rejected and T_i is rolled back.

Time-stamp ordering rules can be modified to make the schedule view serializable.

Instead of making T_i rolled back, the 'write' operation itself is ignored.

UNIT 3

Concept of Error Handling

An error condition during a program execution is called an exception in PL/SQL. PL/SQL supports programmers to catch such conditions using EXCEPTION block in the program and an appropriate action is taken against the error condition.

1) What is Exception Handling?

PL/SQL provides a feature to handle the Exceptions which occur in a PL/SQL Block known as exception Handling. Using Exception Handling we can test the code and avoid it from exiting abruptly.

When an exception occurs a messages which explains its cause is recieved.
PL/SQL Exception message consists of three parts.
1) Type of Exception
2) An Error Code
3) A message

By Handling the exceptions we can ensure a PL/SQL block does not exit abruptly.

Syntax for Exception Handling

The General Syntax for exception handling is as follows. Here you can list down as many as exceptions you want to handle. The default exception will be handled using *WHEN others THEN*:

```
DECLARE
   <declarations section>
BEGIN
   <executable command(s)>
EXCEPTION
   <exception handling goes here >
   WHEN exception1 THEN
       exception1-handling-statements
   WHEN exception2  THEN
      exception2-handling-statements
   WHEN exception3 THEN
      exception3-handling-statements
   ........
   WHEN others THEN
      exception3-handling-statements
END;
```

Example

Let us write some simple code to illustrate the concept. We will be using the CUSTOMERS table we had created and used in the previous chapters:

```
DECLARE
   c_id customers.id%type := 8;
```

```
   c_name  customers.name%type;
   c_addr customers.address%type;
BEGIN
   SELECT  name, address INTO  c_name, c_addr
   FROM customers
   WHERE id = c_id;

   DBMS_OUTPUT.PUT_LINE ('Name: '||  c_name);
   DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);
EXCEPTION
   WHEN no_data_found THEN
      dbms_output.put_line('No such customer!');
   WHEN others THEN
      dbms_output.put_line('Error!');
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
No such customer!

PL/SQL procedure successfully completed.
```

The above program displays the name and address of a customer whose ID is given. Since there is no customer with ID value 8 in our database, the program raises the run-time exception **NO_DATA_FOUND**, which is captured in **EXCEPTION** block.

Raising Exceptions

Exceptions are raised by the database server automatically whenever there is any internal database error, but exceptions can be raised explicitly by the programmer by using the command **RAISE**. Following is the simple syntax of raising an exception:

```
DECLARE
   exception_name EXCEPTION;
BEGIN
   IF condition THEN
      RAISE exception_name;
   END IF;
EXCEPTION
   WHEN exception_name THEN
   statement;
END;
```

You can use above syntax in raising Oracle standard exception or any user-defined exception. Next section will give you an example on raising user-defined exception, similar way you can raise Oracle standard exceptions as well.

User-defined Exceptions

PL/SQL allows you to define your own exceptions according to the need of your program. A user-defined exception must be declared and then raised explicitly, using either a RAISE statement or the procedure DBMS_STANDARD.RAISE_APPLICATION_ERROR.

The syntax for declaring an exception is:

```
DECLARE
   my-exception EXCEPTION;
```

Example:

The following example illustrates the concept. This program asks for a customer ID, when the user enters an invalid ID, the exception invalid_id is raised.

```
DECLARE
```

```
   c_id customers.id%type := &cc_id;
   c_name  customers.name%type;
   c_addr customers.address%type;

   -- user defined exception
   ex_invalid_id  EXCEPTION;
BEGIN
   IF c_id <= 0 THEN
      RAISE ex_invalid_id;
   ELSE
      SELECT  name, address INTO  c_name, c_addr
      FROM customers
      WHERE id = c_id;

      DBMS_OUTPUT.PUT_LINE ('Name: '||  c_name);
      DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);
   END IF;
EXCEPTION
   WHEN ex_invalid_id THEN
      dbms_output.put_line('ID must be greater than zero!');
   WHEN no_data_found THEN
      dbms_output.put_line('No such customer!');
   WHEN others THEN
      dbms_output.put_line('Error!');
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

```
Enter value for cc_id: -6 (let's enter a value -6)
old  2: c_id customers.id%type := &cc_id;
new  2: c_id customers.id%type := -6;
ID must be greater than zero!

PL/SQL procedure successfully completed.
```

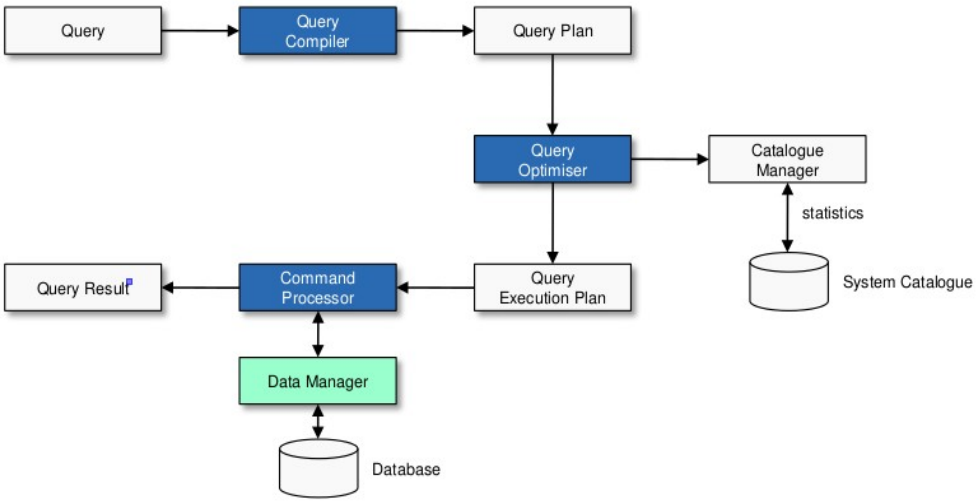
UNIT 4

Query Processing and Optimization

Introduction to Query Processing

With higher level database query languages such as SQL, a special component of the DBMS called the Query Processor takes care of arranging the underlying access routines to satisfy a given query. Thus queries can be specified in terms of the required results rather than in terms of how to achieve those results.

Basic Query Processing Steps



A query is processed in four general steps:

1. Scanning and Parsing
2. Query Optimization or planning the execution strategy
3. Query Code Generator (interpreted or compiled)
4. Execution in the runtime database processor

1. Scanning and Parsing

1. When a query is first submitted (via an applications program), it must be scanned and parsed to determine if the query consists of appropriate syntax.
2. **Scanning** is the process of converting the query text into a tokenized representation.
3. The tokenized representation is more compact and is suitable for processing by the parser.
4. This representation may be in a tree form.
5. The **Parser** checks the tokenized representation for correct syntax.
6. In this stage, checks are made to determine if columns and tables identified in the query exist in the database and if the query has been formed correctly with the appropriate keywords and structure.
7. If the query passes the parsing checks, then it is passed on to the Query Optimizer.

2. Query Optimization or Planning the Execution Strategy

1. For any given query, there may be a number of different ways to execute it.
2. Each operation in the query (SELECT, JOIN, etc.) can be implemented using one or more different *Access Routines*.
3. For example, an access routine that employs an index to retrieve some rows would be more efficient that an access routine that performs a full table scan.
4. The goal of the **query optimizer** is to find a *reasonably efficient* strategy for executing the query (not quite what the name implies) using the access routines.
5. Optimization typically takes one of two forms: *Heuristic Optimization* or *Cost Based Optimization*
6. In **Heuristic Optimization** (also called Rule Based), the query execution is refined based on *heuristic rules* for reordering the individual operations.

7. With **Cost Based Optimization** (also called Systematic), the overall cost of executing the query is systematically reduced by estimating the costs of executing several different execution plans.

3. Query Code Generator (interpreted or compiled)

1. Once the query optimizer has determined the execution plan (the specific ordering of access routines), the code generator writes out the actual access routines to be executed.
2. With an interactive session, the query code is interpreted and passed directly to the runtime database processor for execution.
3. It is also possible to *compile* the access routines and store them for later execution.

4. Execution in the runtime database processor

1. At this point, the query has been scanned, parsed, planned and (possibly) compiled.
2. The runtime database processor then executes the access routines against the database.
3. The results are returned to the application that made the query in the first place.
4. Any runtime errors are also returned.