The SQL ANY and ALL Operators

The ANY and ALL operators are used with a WHERE or HAVING clause.
The ANY operator returns true if any of the subquery values meet the condition.
The ALL operator returns true if all of the subquery values meet the condition.
ANY Syntax
SELECT *column_name(s)*
FROM *table_name*
WHERE *column_name operator* ANY

(SELECT *column_name* FROM *table_name* WHERE *condition*);
ALL Syntax
SELECT *column_name(s)*
FROM *table_name*
WHERE *column_name operator* ALL

(SELECT *column_name* FROM *table_name* WHERE *condition*);

SQL ANY Examples

The ANY operator returns TRUE if any of the subquery values meet the condition.
The following SQL statement returns TRUE and lists the productnames if it finds ANY records in the OrderDetails table that quantity = 10:

Example

```
SELECT ProductName
FROM Products
WHERE ProductID      = ANY (SELECT ProductID FROM OrderDetails WHERE Quantity = 10);
```

Example

```
SELECT ProductName
FROM Products
WHERE ProductID      = ANY (SELECT ProductID FROM OrderDetails WHERE Quantity > 99);
```

SQL ALL Example

The ALL operator returns TRUE if all of the subquery values meet the condition.
The following SQL statement returns TRUE and lists the productnames if ALL the records in the OrderDetails table has quantity = 10:

Example

```
SELECT ProductName
FROM Products
WHERE ProductID= ALL (SELECT ProductID FROM OrderDetails WHERE Quantity= 10)
;
```

Oracle Joins

Join is a query that is used to combine rows from two or more tables, views, or materialized views. It retrieves data from multiple tables and creates a new table.
Join Conditions
There may be at least one join condition either in the FROM clause or in the WHERE clause for joining two tables. It compares two columns from different tables and combines pair of rows, each containing one row from each table, for which join condition is true.

Types of Joins

- Inner Joins (Simple Join)
- Outer Joins
  - Left Outer Join (Left Join)
  - Right Outer Join (Right Join)
  - Full Outer Join (Full Join)
- Equijoins
- Self Joins
- Cross Joins (Cartesian Products)
- Antijoins
- Semijoins

SQL JOIN

A JOIN clause is used to combine rows from two or more tables, based on a related column between them.

Let's look at a selection from the "Orders" table:

OrderID	CustomerID	OrderDate
10308	2	1996-09-18
10309	1	1996-09-19
10310	3	1996-09-20

Then, look at a selection from the "Customers" table:

CustomerID	CustomerName	ContactName	Country
1	Prakash Patel	Mr. Sohil	Germany
2	Sunidhi Chauhan	Mr. Jay	Mexico
3	Hemant Vyas	Ms. Prabha	Mexico

Then, we can create the following SQL statement (that contains an INNER JOIN), that selects records that have matching values in both tables:

Example

```
SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderDate
FROM Orders
INNER JOIN Customers ON Orders.CustomerID=Customers.CustomerID;
```

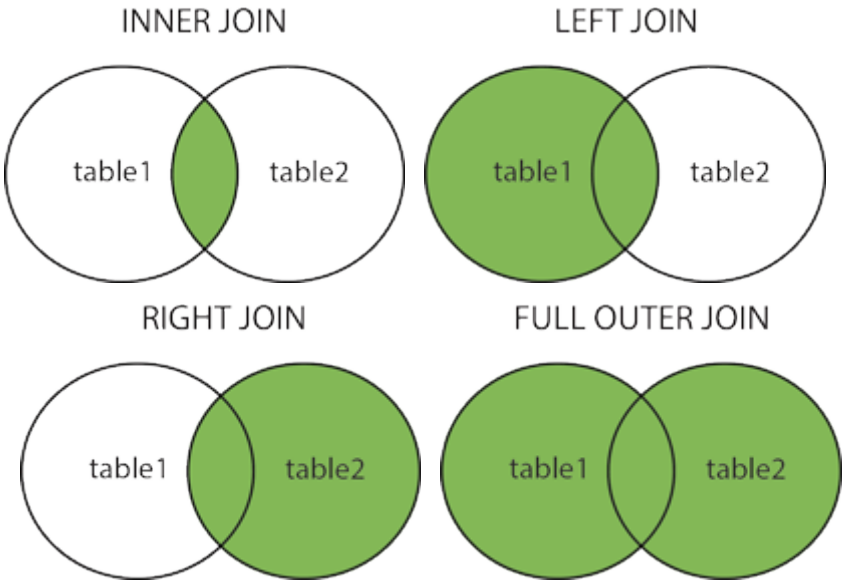
and it will produce something like this:

OrderID	CustomerName	OrderDate
10308	Sunidhi Chauhan	9/18/1996
10310	Hemant Vyas	11/27/1996
10309	Prakash Patel	12/16/1996
10308	Sunidhi Chauhan	11/15/1996
10310	Hemant Vyas	8/12/1996

Different Types of SQL JOINs

Here are the different types of the JOINs in SQL:

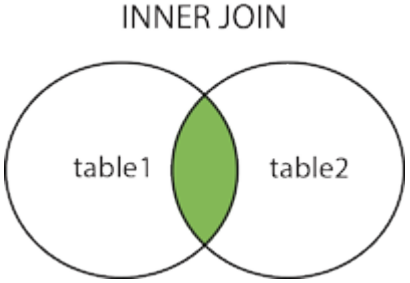
- **(INNER) JOIN**: Returns records that have matching values in both tables
- **LEFT (OUTER) JOIN**: Returns all records from the left table, and the matched records from the right table
- **RIGHT (OUTER) JOIN**: Returns all records from the right table, and the matched records from the left table
- **FULL (OUTER) JOIN**: Returns all records when there is a match in either left or right table



SQL INNER JOIN Keyword

The INNER JOIN keyword selects records that have matching values in both tables.

INNER JOIN Syntax
SELECT *column_name(s)*
FROM *table1*
INNER JOIN *table2*
ON *table1.column_name = table2.column_name*;



Below is a selection from the "Orders" table:

OrderID	CustomerID	EmployeeID	OrderDate	ShipperID
10308	2	7	1996-09-18	3
10309	37	3	1996-09-19	1

10310	77	8	1996-09-20	2

And a selection from the "Customers" table:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico

SQL INNER JOIN Example

The following SQL statement selects all orders with customer information:

Example

```
SELECT Orders.OrderID, Customers.CustomerName
FROM Orders
INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID;
```

JOIN Three Tables

The following SQL statement selects all orders with customer and shipper information:

Example

```
SELECT Orders.OrderID, Customers.CustomerName, Shippers.ShipperName
FROM ((Orders
INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID)
INNER JOIN Shippers ON Orders.ShipperID = Shippers.ShipperID);
```

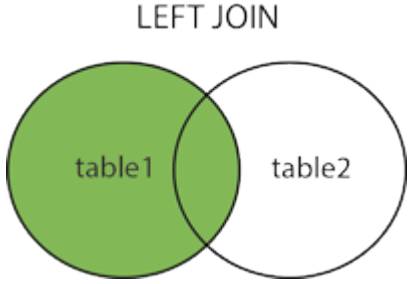
SQL LEFT JOIN Keyword

The LEFT JOIN keyword returns all records from the left table (table1), and the matched records from the right table (table2). The result is NULL from the right side, if there is no match.

LEFT JOIN Syntax

```
SELECT column_name(s)
FROM table1
LEFT JOIN table2
ON table1.column_name = table2.column_name;
```

Note: In some databases LEFT JOIN is called LEFT OUTER JOIN.



Below is a selection from the "Customers" table:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico

And a selection from the "Orders" table:

OrderID	CustomerID	EmployeeID	OrderDate	ShipperID
10308	2	7	1996-09-18	3
10309	37	3	1996-09-19	1
10310	77	8	1996-09-20	2

SQL LEFT JOIN Example

The following SQL statement will select all customers, and any orders they might have:

Example

```
SELECT Customers.CustomerName, Orders.OrderID
FROM Customers
LEFT JOIN Orders ON Customers.CustomerID = Orders.CustomerID
ORDER BY Customers.CustomerName;
```

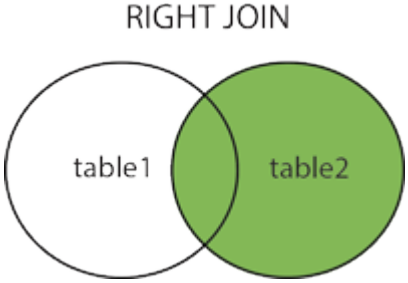
SQL RIGHT JOIN Keyword

The RIGHT JOIN keyword returns all records from the right table (table2), and the matched records from the left table (table1). The result is NULL from the left side, when there is no match.

RIGHT JOIN Syntax

```
SELECT column_name(s)
FROM table1
RIGHT JOIN table2
ON table1.column_name = table2.column_name;
```

Note: In some databases RIGHT JOIN is called RIGHT OUTER JOIN.



Below is a selection from the "Orders" table:

OrderID	CustomerID	EmployeeID	OrderDate	ShipperID
10308	2	7	1996-09-18	3
10309	37	3	1996-09-19	1
10310	77	8	1996-09-20	2

And a selection from the "Employees" table:

EmployeeID	LastName	FirstName	BirthDate	Photo
1	Davolio	Nancy	12/8/1968	EmpID1.pic
2	Fuller	Andrew	2/19/1952	EmpID2.pic
3	Leverling	Janet	8/30/1963	EmpID3.pic

SQL RIGHT JOIN Example

The following SQL statement will return all employees, and any orders they might have placed:

Example

```
SELECT Orders.OrderID, Employees.LastName, Employees.FirstName
FROM Orders
RIGHT JOIN Employees ON Orders.EmployeeID = Employees.EmployeeID
ORDER BY Orders.OrderID;
```

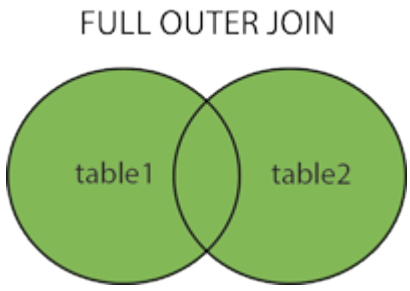
SQL FULL OUTER JOIN Keyword

The FULL OUTER JOIN keyword return all records when there is a match in left (table1) or right (table2) table records.

Remarks: FULL OUTER JOIN and FULL JOIN are the same.

FULL OUTER JOIN Syntax

```
SELECT column_name(s)
FROM table1
FULL OUTER JOIN table2
ON table1.column_name = table2.column_name
WHERE condition;
```



Below is a selection from the "Customers" table:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico

And a selection from the "Orders" table:

OrderID	CustomerID	EmployeeID	OrderDate	ShipperID
10308	2	7	1996-09-18	3
10309	37	3	1996-09-19	1
10310	77	8	1996-09-20	2

SQL FULL OUTER JOIN Example
The following SQL statement selects all customers, and all orders:

```
SELECT Customers.CustomerName, Orders.OrderID
FROM Customers
FULL OUTER JOIN Orders ON Customers.CustomerID=Orders.CustomerID
ORDER BY Customers.CustomerName;
```

SQL Self JOIN
A self JOIN is a regular join, but the table is joined with itself.
Self JOIN Syntax

```
SELECT column_name(s)
FROM table1 T1, table1 T2
WHERE condition;
```

T1 and *T2* are different table aliases for the same table.

Demo Database
In this tutorial we will use the well-known Northwind sample database.
Below is a selection from the "Customers" table:

CustomerID	CustomerName	ContactName	Address	City	Postal Code	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany

2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico

SQL Self JOIN Example

The following SQL statement matches customers that are from the same city:

Example

```
SELECT A.CustomerName AS CustomerName1,
B.CustomerName AS CustomerName2, A.City
FROM Customers A, Customers B
WHERE A.CustomerID <> B.CustomerID
AND A.City = B.City
ORDER BY A.City;
```

What is Natural Join in Oracle?

- The join is based on all the columns in the two tables that have the same name and data types.
- The join creates, by using the NATURAL JOIN keywords.
- It selects rows from the two tables that have equal values in all matched columns.
- When specifying columns that are involved in the natural join, do not qualify the column name with a table name or table alias.

Syntax

```
SELECT table1.column, table2.column
FROM table1
NATURAL JOIN table2;
```

Where table1, table2 are the name of the tables participating in joining.

Example: Oracle Natural Joins

In this example, the LOCATIONS table is joined to the COUNTRY table by the country_id column, which is the only column of the same name in both tables. If other common columns were present, the join would have used them all.

SQL Code:

```
SQL> SELECT postal_code, city,
  2  region_id, country_name
  3  FROM locations
  4  NATURAL JOIN countries;
```

Sample Output:

```
POSTAL_CODE  CITY                          REGION_ID COUNTRY_NAME
-----------  ---------------------------- ---------- ------------------------
00989        Roma                                  1 Italy
10934        Venice                                1 Italy
1689         Tokyo                                 3 Japan
6823         Hiroshima                             3 Japan
26192        Southlake                             2 United States of America
```

Natural Joins with a WHERE Clause

You can implement additional restrictions on a natural join using a WHERE clause. In the previous example the LOCATIONS table was joined to the DEPARTMENT table by the COUNTRY_ID column, now you can limit the rows of output to those with a location_id greater than 2000.

SQL Code:

```
SQL> SELECT postal_code, city,
  2  region_id, country_name
  3  FROM locations
  4  NATURAL JOIN countries
  5  WHERE location_id>2000;
```

Sample Output:

```
POSTAL_CODE  CITY                          REGION_ID COUNTRY_NAME
-----------  ----------------------------- ---------- ------------------
490231       Bombay                                 3 India
2901         Sydney                                 3 Australia
540198       Singapore                              3 Singapore
```

Oracle Cross Join (Cartesian Products)

The CROSS JOIN specifies that all rows from first table join with all of the rows of second table. If there are "x" rows in table1 and "y" rows in table2 then the cross join result set have x*y rows. It normally happens when no matching join columns are specified.

In simple words you can say that if two tables in a join query have no join condition, then the Oracle returns their Cartesian product.

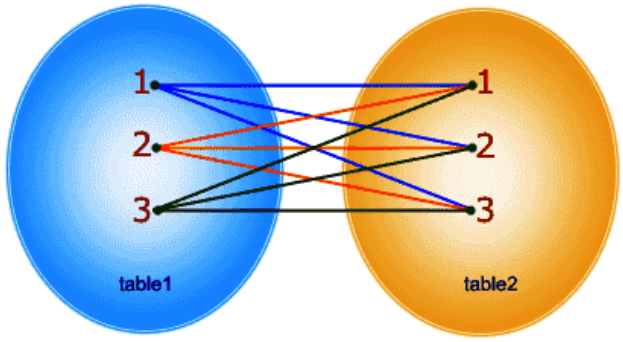
1. **SELECT** *
2. **FROM** table1
3. CROSS JOIN table2;

Or

1. **SELECT** * **FROM** table1, table2

Both the above syntax are same and used for Cartesian product. They provide similar result after execution.

Image representation of cross join



Oracle Cross Join Example

Let's take two tables "customer" and "supplier".

Customer table detail

```
1. CREATE TABLE  "CUSTOMER"
2.    (    "CUSTOMER_ID" NUMBER,
3.     "FIRST_NAME" VARCHAR2(4000),
4.     "LAST_NAME" VARCHAR2(4000)
5.    )
6. /
```

Supplier table detail

```
1. CREATE TABLE  "SUPPLIER"
2.    (    "SUPPLIER_ID" NUMBER,
3.     "FIRST_NAME" VARCHAR2(4000),
4.     "LAST_NAME" VARCHAR2(4000)
5.    )
6. /
```

Execute this query

```
1. SELECT * FROM customer,supplier
```

What is view ?

View is the simply subset of table which are stored logically in a database means a view is a virtual table in the database whose contents are defined by a query.

To the database user, the view appears just like a real table, with a set of named columns and rows of data. SQL creates the illusion of the view by giving the view a name like a table name and storing the definition of the view in the database.

Views are used for security purpose in databases,views restricts the user from viewing certain column and rows means by using view we can apply the restriction on accessing the particular rows and columns for specific user. Views display only those data which are mentioned in the query, so it shows only data which is returned by the query that is defined at the time of creation of the View.

Advantages of views

Security

Each user can be given permission to access the database only through a small set of views that contain the specific data the user is authorized to see, thus restricting the user's access to stored data

Query Simplicity

A view can draw data from several different tables and present it as a single table, turning multi-table queries into single-table queries against the view.

Structural simplicity
Views can give a user a "personalized" view of the database structure, presenting the database as a set of virtual tables that make sense for that user.

Consistency
A view can present a consistent, unchanged image of the structure of the database, even if the underlying source tables are split, restructured, or renamed.

Data Integrity

If data is accessed and entered through a view, the DBMS can automatically check the data to ensure that it meets the specified integrity constraints.

Logical data independence.

View can make the application and database tables to a certain extent independent. If there is no view, the application must be based on a table. With the view, the program can be established in view of above, to view the program with a database table to be separated.

Disadvantages of views

Performance

Views create the appearance of a table, but the DBMS must still translate queries against the view into queries against the underlying source tables. If the view is defined by a complex, multi-table query then simple queries on the views may take considerable time.

Update restrictions

When a user tries to update rows of a view, the DBMS must translate the request into an update on rows of the underlying source tables. This is possible for simple views, but more complex views are often restricted to read-only.

About Views

A **view** is a logical representation of another table or combination of tables. A view derives its data from the tables on which it is based. These tables are called **base tables**. Base tables might in turn be actual tables or might be views themselves. All operations performed on a view actually affect the base table of the view. You can use views in almost the same way as tables. You can query, update, insert into, and delete from views, just as you can standard tables.
Views can provide a different representation (such as subsets or supersets) of the data that resides within other tables and views. Views are very powerful because they allow you to tailor the presentation of data to different types of users.
The following statement creates a view on a subset of data in the emp table:

```
CREATE VIEW sales_staff AS
    SELECT empno, ename, deptno
    FROM emp
    WHERE deptno = 10
  WITH CHECK OPTION CONSTRAINT sales_staff_cnst;
```

The query that defines the sales_staff view references only rows in department 10. Furthermore, the CHECK OPTION creates the view with the constraint (named sales_staff_cnst) that INSERT and UPDATE statements issued against the view cannot result in rows that the view cannot select. For example, the following INSERT statement successfully inserts a row into the emp table by means of the sales_staff view, which contains all rows with department number 10:

```
INSERT INTO sales_staff VALUES (7584, 'OSTER', 10);
```

However, the following INSERT statement returns an error because it attempts to insert a row for department number 30, which cannot be selected using the sales_staff view:

```
INSERT INTO sales_staff VALUES (7591, 'WILLIAMS', 30);
```

The view could have been constructed specifying the WITH READ ONLY clause, which prevents any updates, inserts, or deletes from being done to the base table through the view. If no WITH clause is specified, the view, with some restrictions, is inherently updatable.

Join Views

You can also create views that specify more than one base table or view in the FROM clause. These are called **join views**. The following statement creates the division1_staffview that joins data from the emp and dept tables:

```
CREATE VIEW division1_staff AS
    SELECT ename, empno, job, dname
    FROM emp, dept
    WHERE emp.deptno IN (10, 30)
      AND emp.deptno = dept.deptno;
```

An **updatable join view** is a join view where UPDATE, INSERT, and DELETE operations are allowed. See "Updating a Join View" for further discussion.

Expansion of Defining Queries at View Creation Time

When a view is created, Oracle Database expands any wildcard (*) in a top-level view query into a column list. The resulting query is stored in the data dictionary; any subqueries are left intact. The column names in an expanded column list are enclosed in quote marks to account for the possibility that the columns of the base object were originally entered with quotes and require them for the query to be syntactically correct.

As an example, assume that the dept view is created as follows:

```
CREATE VIEW dept AS SELECT * FROM scott.dept;
```

The database stores the defining query of the dept view as:

```
SELECT "DEPTNO", "DNAME", "LOC" FROM scott.dept;
```

Views created with errors do not have wildcards expanded. However, if the view is eventually compiled without errors, wildcards in the defining query are expanded.

Creating Views with Errors

If there are no syntax errors in a CREATE VIEW statement, the database can create the view even if the defining query of the view cannot be executed. In this case, the view is considered "created with errors." For example, when a view is created that refers to a nonexistent table or an invalid column of an existing table, or when the view owner does not have the required privileges, the view can be created anyway and entered into the data dictionary. However, the view is not yet usable.

To create a view with errors, you must include the FORCE clause of the CREATE VIEW statement.

Using Views in Queries

To issue a query or an INSERT, UPDATE, or DELETE statement against a view, you must have the SELECT, INSERT, UPDATE, or DELETE object privilege for the view, respectively, either explicitly or through a role.

Views can be queried in the same manner as tables. For example, to query the Division1_staff view, enter a valid SELECT statement that references the view:

```
SELECT * FROM Division1_staff;

ENAME      EMPNO      JOB           DNAME
--------------------------------------------------
CLARK       7782      MANAGER       ACCOUNTING
KING        7839     PRESIDENT      ACCOUNTING
MILLER      7934      CLERK         ACCOUNTING
ALLEN       7499     SALESMAN       SALES
WARD        7521     SALESMAN       SALES
JAMES       7900     CLERK         SALES
TURNER      7844      SALESMAN      SALES
MARTIN      7654      SALESMAN      SALES
BLAKE       7698     MANAGER       SALES
```

With some restrictions, rows can be inserted into, updated in, or deleted from a base table using a view. The following statement inserts a new row into the emp table using the sales_staff view:

```
INSERT INTO sales_staff
   VALUES (7954, 'OSTER', 30);
```

Restrictions on DML operations for views use the following criteria in the order listed:

1. If a view is defined by a query that contains SET or DISTINCT operators, a GROUP BY clause, or a group function, then rows cannot be inserted into, updated in, or deleted from the base tables using the view.
2. If a view is defined with WITH CHECK OPTION, a row cannot be inserted into, or updated in, the base table (using the view), if the view cannot select the row from the base table.
3. If a NOT NULL column that does not have a DEFAULT clause is omitted from the view, then a row cannot be inserted into the base table using the view.
4. If the view was created by using an expression, such as DECODE(deptno, 10, "SALES", ...), then rows cannot be inserted into or updated in the base table using the view.

The constraint created by WITH CHECK OPTION of the sales_staff view only allows rows that have a department number of 30 to be inserted into, or updated in, the emp table. Alternatively, assume that the sales_staff view is defined by the following statement (that is, excluding the deptno column):

```
CREATE VIEW sales_staff AS
   SELECT empno, ename
   FROM emp
   WHERE deptno = 10
   WITH CHECK OPTION CONSTRAINT sales_staff_cnst;
```

Considering this view definition, you can update the empno or ename fields of existing records, but you cannot insert rows into the emp table through the sales_staff view because

the view does not let you alter the deptno field. If you had defined a DEFAULT value of 10 on the deptno field, then you could perform inserts.

When a user attempts to reference an invalid view, the database returns an error message to the user:

```
ORA-04063: view 'view_name' has errors
```

This error message is returned when a view exists but is unusable due to errors in its query (whether it had errors when originally created or it was created successfully but became unusable later because underlying objects were altered or dropped).

Updating a Join View

An updatable join view (also referred to as a **modifiable join view**) is a view that contains more than one table in the top-level FROM clause of the SELECT statement, and is not restricted by the WITH READ ONLY clause.

Introduction to PL/SQL

The PL/SQL programming language was developed by Oracle Corporation in the late 1980s as procedural extension language for SQL and the Oracle relational database. Following are certain notable facts about PL/SQL −

- PL/SQL is a completely portable, high-performance transaction-processing language.
- PL/SQL provides a built-in, interpreted and OS independent programming environment.
- PL/SQL can also directly be called from the command-line **SQL*Plus interface**.
- Direct call can also be made from external programming language calls to database.
- PL/SQL's general syntax is based on that of ADA and Pascal programming language.
- Apart from Oracle, PL/SQL is available in **Times Ten in-memory database** and **IBM DB2**.

Features of PL/SQL

PL/SQL has the following features −

- PL/SQL is tightly integrated with SQL.
- It offers extensive error checking.
- It offers numerous data types.
- It offers a variety of programming structures.
- It supports structured programming through functions and procedures.
- It supports object-oriented programming.
- It supports the development of web applications and server pages.

Advantages of PL/SQL

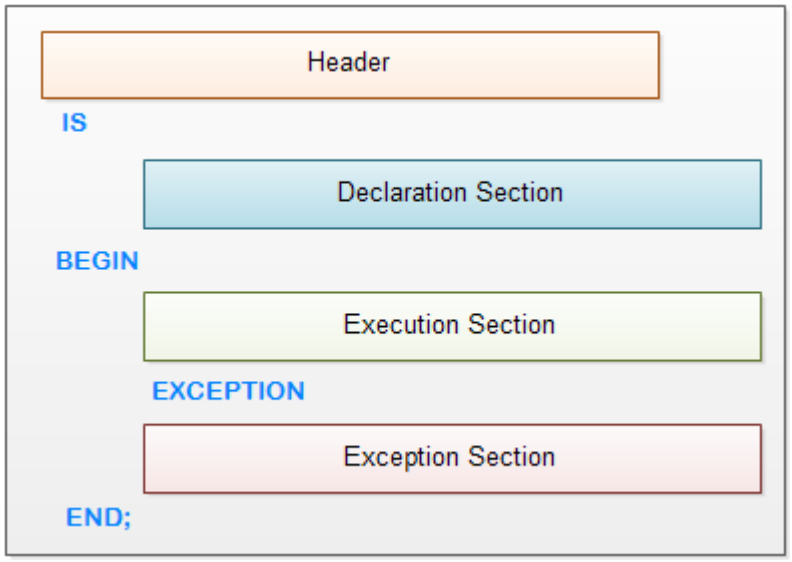
PL/SQL has the following advantages −

- SQL is the standard database language and PL/SQL is strongly integrated with SQL. PL/SQL supports both static and dynamic SQL. Static SQL supports DML operations and transaction control from PL/SQL block. PL/SQL allows sending an entire block of statements to the database at one time. This reduces network traffic and provides high performance for the applications.
- PL/SQL gives high productivity to programmers as it can query, transform, and update data in a database.
- PL/SQL saves time on design and debugging by strong features, such as exception handling, encapsulation, data hiding, and object-oriented data types.
- Applications written in PL/SQL are fully portable.
- PL/SQL provides high security level.
- PL/SQL provides access to predefined SQL packages.
- PL/SQL provides support for Object-Oriented Programming.
- PL/SQL provides support for developing Web Applications and Server Pages.

Introducing PL/SQL block structure

PL/SQL program units organize the code into blocks. A block without a name is known as an anonymous block. The anonymous block is the simplest unit in PL/SQL. It is called anonymous block because it is not saved in the Oracle database.

Let's examine the PL/SQL block structure in greater detail.



PL/SQL Block Structure

The anonymous block has three basic sections that are the declaration, execution, and exception handling. Only the execution section is mandatory and the others are optional.

- The declaration section allows you to define data types, structures, and variables. You often declare variables in the declaration section by giving those names, data types, and initial values.
- The execution section is required in a block structure and it must have at least one statement. The execution section is the place where you put the execution code or business logic code. You can use both procedural and SQL statements inside the execution section.
- The exception handling section is starting with the EXCEPTION keyword. The exception section is the place that you put the code to handle exceptions. You can either catch or handle exceptions in the exception section.

Notice that the single forward slash (/) is a signal to instruct SQL*Plus to execute the PL/SQL block.

PL/SQL Transaction Commit, Rollback, Savepoint, Autocommit

Oracle PL/SQL transaction oriented language. Oracle transactions provide a data integrity. PL/SQL transaction is a series of SQL data manipulation statements that are work logical unit. Transaction is an atomic unit all changes either committed or rollback.

At the end of the transaction that makes database changes, Oracle makes all the changes permanent save or may be undone. If your program fails in the middle of a transaction, Oracle detect the error and rollback the transaction and restoring the database.

You can use the COMMIT, ROLLBACK, SAVEPOINT, and SET TRANSACTION command to control the transaction.

1. COMMIT: COMMIT command to make changes permanent save to a database during the current transaction.
2. ROLLBACK: ROLLBACK command execute at the end of current transaction and undo/undone any changes made since the begin transaction.
3. SAVEPOINT: SAVEPOINT command save the current point with the unique name in the processing of a transaction.
4. AUTOCOMMIT: Set AUTOCOMMIT ON to execute COMMIT Statement automatically.
5. SET TRANSACTION: PL/SQL SET TRANSACTION command set the transaction properties such as read-write/read only access.

Commit

The COMMIT statement to make changes permanent save to a database during the current transaction and visible to other users,

Commit Syntax

```
SQL>COMMIT [COMMENT "comment text"];
```

Commit comments are only supported for backward compatibility. In a future release commit comment will come to a deprecated.

Commit Example

```
SQL>BEGIN
   UPDATE emp_information SET emp_dept='Web Developer'
      WHERE emp_name='Saulin';
   COMMIT;
END;
/
```

Rollback

The ROLLBACK statement ends the current transaction and undoes any changes made during that transaction. If you make a mistake, such as deleting the wrong row from a table, a rollback restores the original data. If you cannot finish a transaction because an exception is raised or a SQL statement fails, a rollback lets you take corrective action and perhaps start over.

ROLLBACK Syntax

```
SQL>ROLLBACK [To SAVEPOINT_NAME];
```

ROLLBACK Example

```
SQL>DECLARE
  emp_id  emp.empno%TYPE;
BEGIN
  SAVEPOINT dup_found;
  UPDATE emp SET eno=1
```

```
      WHERE empname = 'Forbs ross'
EXCEPTION
  WHEN DUP_VAL_ON_INDEX THEN
    ROLLBACK TO dup_found;
END;
/
```

Savepoint

SAVEPOINT savepoint_names marks the current point in the processing of a transaction. Savepoints let you rollback part of a transaction instead of the whole transaction.

SAVEPOINT Syntax

```
SQL>SAVEPOINT SAVEPOINT_NAME;
```

SAVEPOINT Example

```
SQL>DECLARE
  emp_id  emp.empno%TYPE;
BEGIN
  SAVEPOINT dup_found;
  UPDATE emp SET eno=1
      WHERE empname = 'Forbs ross'
EXCEPTION
  WHEN DUP_VAL_ON_INDEX THEN
    ROLLBACK TO dup_found;
END;
/
```

Autocommit

No need to execute COMMIT statement every time. You just set AUTOCOMMIT ON to execute COMMIT Statement automatically. It's automatic execute for each DML statement. set auto commit on using following statement,

AUTOCOMMIT Example

```
SQL>SET AUTOCOMMIT ON;
```

You can also set auto commit off,

```
SQL>SET AUTOCOMMIT OFF;
```

Set Transaction

SET TRANSACTION statement is use to set transaction are read-only or both read write. you can also assign transaction name.

SET TRANSACTION Syntax

```
SQL>SET TRANSACTION [ READ ONLY | READ WRITE ]
      [ NAME 'transaction_name' ];
```

Set transaction name using the SET TRANSACTION [...] NAME statement before you start the transaction.

SET TRANSACTION Example

```
SQL>SET TRANSACTION READ WRITE NAME 'tran_exp';
```