UNIT-III : Data Manipulation Language: Insert, Update, Delete and Select statement with all its clauses – **Subqueries** : Nested and Correlated subqueries - **JOINS** : Self Join, Equi Join, Non-Equi Join, Outer Join **- VIEWS** : View Definition – Uses of Views -Simple and Complex Views- View Expansion - Updating tables through views.

DATA MANIPULATION LANGUAGE (DML)

- DML is used for **accessing and manipulating the data** organized by the DBMS.
- **The types of access are:**
  1. **Retrieval** of Information stored in database
  2. **Insertion** of new information into the Database
  3. **Deletion** of information in the database.
  4. **Modification** of information in the database.
- **Query is a statement requesting retrieval of information.**
- ***Query Language* is the portion of DML that deals with information retrieval**
- *Two types of DMLs* :
  1. **Procedural DML– user specifies what data is required and how to get those data**
  2. **Declarative (non-procedural) DML– user specifies what data is required without specifying how to get those data**

Note : SQL COMMANDS ARE NOT CASE SENSITIVE. BUT, THE DATA USED TO COMPARE IS CASE SENSITIVE.

Insertion of new information into the Database

SQL INSERT Statement

- The INSERT Statement is used **to add new rows of data to a table.**
- **When adding a new row, we should ensure the datatype of the value and the column matches and follow the integrity constraints, if any, defined for the table.**
- **When adding a row, only the characters or date values should be enclosed with single quotes.**

We can insert data to a table in two ways,

1) *Inserting the data directly to a table.*

Syntax:

```
INSERT INTO TABLE_NAME [ (col1, col2, col3,...,colN)]
                       VALUES (value1, value2, value3,...,valueN);
```

Here, col1, col2,...,colN -- the names of the columns in the table into which we want to insert

Example: To insert a row to the employee table,

```
INSERT INTO employee (id, name, dept, age, salary location)
                       VALUES (105, 'Srinath', 'Aeronautics', 27, 33000);
```

- **While inserting a row, if we are adding values for all the columns of the table we need not specify the column(s) name in the sql query. But we need to make sure the order of the values is in the same order as the columns in the table. The sql insert query will be as follows**

Syntax:

INSERT INTO TABLE_NAME VALUES (value1, value2, value3,...valueN);

Example: **To insert a row to the employee table,**

INSERT INTO employee VALUES (105, 'Srinath', 'Aeronautics', 27, 33000);

Inserting data to a table through a select statement (Subquery).

Syntax:

INSERT INTO table_name [(column1, column2, ... columnN)]
SELECT column1, column2, ...columnN FROM table_name [WHERE condition];

Here, the select from where statement is evaluated fully before any of its results are inserted into the relation.

Example: To insert a row into the employee table from a temporary table,

INSERT INTO employee (id, name, dept, age, salary location)

SELECT emp_id, emp_name, dept, age, salary, location FROM temp_employee;

If we are inserting data to all the columns, the above insert statement can also be written as,

INSERT INTO employee SELECT * FROM temp_employee;

Modification of the Database – Updation

SQL UPDATE Statement

- The UPDATE Statement is used **to modify the existing rows in a table.**
- **In the Update statement, WHERE clause identifies the rows that get affected. If we do not include the WHERE clause, column values for all the rows get affected.**

Syntax:

```
UPDATE table_name SET column_name1 = value1, column_name2 = value2, ...
                      [WHERE condition]
```

table_name - the table name which has to be updated.

column_name1, column_name2.. - the columns that gets changed.

value1, value2... - are the new values.

Example: To update the location of an employee:

```
UPDATE employee SET location ='Mysore' WHERE id = 101;
```

Example: To change the salaries of all the employees:

```
UPDATE employee SET salary = salary + (salary * 0.2);
```

Deletion of information in the database.

SQL Delete Statement

- The DELETE Statement is used **to delete rows from a table.**
- **The WHERE clause in the sql delete command is optional and it identifies the rows that satisfies the condition given gets deleted.**
- **If you do not include the WHERE clause all the rows in the table will be deleted.**

Syntax :

DELETE FROM table_name [WHERE condition];

Here, table_name is the table which has to be updated.

Example

To delete an employee with id 100 from the employee table:

DELETE FROM employee WHERE id = 100;

To delete all the rows from the employee table:

DELETE FROM employee;

To Retrieve data from Database - SQL SELECT Statement

- **SQL SELECT statement is used to query or retrieve data from a table in the database.**
- **A query may retrieve information from specified columns or from all of the columns in the table.**
- **To create a simple SQL SELECT Statement, we must specify the column(s) name and the table name.**

Syntax :

SELECT *column_list* FROM *table-name*
[WHERE Clause] [GROUP BY clause] [HAVING clause] [ORDER BY clause];

Here, table-name is the name of the table from which the information is retrieved.
column_list includes one or more columns from which data is retrieved. **The code within the brackets is optional.**

NOTE – 1 : In a SQL SELECT statement, **only SELECT and FROM clauses are mandatory**.
Other clauses like WHERE, ORDER BY, GROUP BY, HAVING are optional.

NOTE -2 : *Consider the database with following tables in following examples:*

student_details(id first_name, last_name, age, subject, games)
branch(branch_name, branch_city, assets)
customer(customer_name, customer_street, customer_city)
loan(loan_number, branch_name, amount)
borrower(customer_name, loan_number)
account(account_number, branch_name, balance)
depositor(customer_name, account_number

THE SELECT CLAUSE

Selecting Columns from a table

- The **select** clause lists the columns / attributes whose data are desired in the result of a query.

  Example :, To select first name and last name of all the students.

  SELECT first_name, last_name FROM student_details;

- An **asterisk** in the select clause denotes "all attributes" and "all rows" in the table

  Example : To display all rows and all columns in the *student_details* relations

  SELECT * FROM *student_details;*

- SQL allows duplicates in relations as well as in query results.
- To force the elimination of duplicates, insert the keyword **DISTINCT** after SELECT.

  Example *: To* Find the names of first_name in the *student_details* relations, and remove duplicates

  SELECT DISTINCT *first_name* FROM *student_details;*

- The keyword **ALL** specifies that duplicates not be removed.

  Example : Find the names of first_name in the *student_details* relation and do not remove duplicates

  SELECT ALL *first_name* FROM *student_details;*

- The **SELECT** clause can contain arithmetic expressions involving the operation, +, –, *, and /, and operating on constants or attributes of tuples.

  Example: **SELECT loan_number, branch_name, amount * 100 FROM loan;**

THE FROM CLAUSE

- The **from** clause lists the relations involved in the query.
- It corresponds to the Cartesian product operation of the relational algebra.

Example: To find the Cartesian product *borrower X loan*

SELECT * FROM *borrower, loan* ;

Example: Find the name, loan number and loan amount of all customers having a loan at the Avadi branch.

SELECT *customer_name, borrower.loan_number, amount* FROM *borrower, loan*
WHERE *borrower.loan_number = loan.loan_number* AND *branch_name* = 'Avadi' ;

THE WHERE CLAUSE

- WHERE Clause is used to retrieve specific information from a table excluding other irrelevant data.
- ***Selecting Rows from a table***

Selecting Rows from a table based on a specific condition can be done using ***Where Clause***

Example: To List the branch details if assets greater than 500000.

SELECT * FROM branch WHERE assets > 500000;

- WHERE clause can be used to restrict the data / row that is retrieved. The condition provided in the WHERE clause filters the rows retrieved from the table and gives you only those rows which you expected to see.
- WHERE clause can be used along with SELECT, DELETE, UPDATE statements.

Example: To Delete the branch details if assets greater than 500000.

DELETE FROM branch WHERE assets > 500000;

THE GROUP BY CLAUSE

- The GROUP BY clause groups the selected rows based on identical values in a column or expression. This clause is typically used with aggregate functions to generate a single result row for each set of unique values in a set of columns or expressions.
- A simple GROUP BY clause consists of a list of one or more columns or expressions that define the sets of rows that aggregations (like SUM, COUNT, MIN, MAX, and AVG) are to be performed on. A change in the value of any of the GROUP BY columns or expressions triggers a new set of rows to be aggregated.
- The following query, which uses a simple GROUP BY clause, obtains the number of orders for each part number in the orders table:

```
SELECT partno, count(*) FROM orders
GROUP BY partno;
```

  The preceding query returns one row for each part number in the orders table, even though there can be many orders for the same part number.
- Nulls are used to represent unknown data, and two nulls are typically not considered to be equal in SQL comparisons. However, the GROUP BY clause treats nulls as equal and returns a single row for nulls in a grouped column or expression.
- Grouping can be performed on multiple columns or expressions. For example, to display the number of orders for each part placed each day:

```
SELECT odate, partno, count(*) FROM orders
GROUP BY odate, partno;
```

- If you specify the GROUP BY clause, columns referenced must be all the columns in the SELECT clause that do not contain an aggregate function.

  For example:

```
SELECT cust_no, odate, COUNT(*) AS number_of_orders   FROM orders
                                  GROUP BY cust_no, odate ;
```

THE HAVING CLAUSE

- The HAVING clause filters the results of the GROUP BY clause by using an aggregate function. The HAVING clause uses the same restriction operators as the WHERE clause.
- For example, to return parts that have sold more than ten times :

```
SELECT odate, partno, count(*) FROM orders
GROUP BY odate, partno   HAVING count(*) > 10;
```

THE ORDER BY CLAUSE

- The ORDER BY clause allows you to specify the columns on which the results table is to be sorted.
- Default ordering will be in ascending order /alphabetical order
- Ordering will be based on the first column following the ORDER BY clause. If a tie occurs, ordering will be based on the next column

  Example :

```
SELECT manager, emp_name, dept_no FROM employee_dim
ORDER BY manager, dept_no;
```

- To display result columns sorted in descending order (reverse numeric or alphabetic order), specify ORDER BY *column_name* DESC. For example, to display the employees in each department from oldest to youngest:

  Example :

```
SELECT dept_no, emp_name, emp_age FROM employee_dim
ORDER BY dept_no, emp_age DESC;
```

How to use expressions in SQL SELECT Statement?

- **SIMPLE ARITHMETIC EXPRESSIONS CAN BE USED IN SELECT, WHERE AND ORDER BY CLAUSES OF THE SQL SELECT STATEMENT.**
- ***When more than one arithmetic operator is used in an expression, then order of evaluation is*:**
  - **Parentheses,**
  - **Division,**
  - **Multiplication,**
  - **Addition and**
  - **Subtraction.**

  THE EVALUATION IS PERFORMED FROM THE LEFT TO THE RIGHT OF THE EXPRESSION.

How to use expressions in the WHERE Clause?

- **Expressions can be used in the WHERE clause of the SELECT statement.**

Example:

Lets consider the employee table. If you want to display employee name, current salary, and a 20% increase in the salary for only those products where the percentage increase in salary is greater than 30000.

```
SELECT name, salary, salary*1.2 AS new_salary FROM employee
                                WHERE salary*1.2 > 30000;
```

Note :

ALIASES DEFINED FOR THE COLUMNS IN THE SELECT STATEMENT CANNOT BE USED IN THE WHERE CLAUSE TO SET CONDITIONS.

The Rename Operation (SQL Alias)

- SQL Aliases are defined for columns and tables.
- Basically aliases are created to make the column selected more readable.
- *ALIASES DEFINED FOR THE COLUMNS IN THE SELECT STATEMENT CANNOT BE USED IN THE WHERE CLAUSE TO SET CONDITIONS. ONLY ALIASES CREATED FOR TABLES CAN BE USED TO REFERENCE THE COLUMNS IN THE TABLE.*
- *ALIASES IS MORE USEFUL WHEN*
  - There are more than one tables involved in a query,
  - Functions are used in the query,
  - The column names are big or not readable,
  - More than one columns are combined together
- SQL allows renaming relations and attributes using the as clause:

  old-name **AS** *new-name*
- Keyword as is optional and may be omitted. Some database such as Oracle require AS to be omitted.

  Example : Find the name, loan number and loan amount of all customers; rename the column name *loan_number* as *loan_id.*

  SELECT *customer_name, borrower.loan_number* **AS** *loan_id, amount* FROM *borrower, loan*
  WHERE *borrower.loan_number = loan.loan_number*

Tuple Variables (*Aliases for tables*) :

- **Tuple variables are defined in the FROM clause using AS clause.**
- **They are used to rename the table names involved in the query to make the query easily readable and understandable .**
- **Keyword AS is optional and may be omitted.**

borrower AS T ≡ borrower T

Example: Find the customer names and their loan numbers and amount for all customers having a loan at some branch.

```
SELECT customer_name, T.loan_number, S.amount  FROM borrower AS T, loan AS S
        WHERE  T.loan_number = S.loan_number ;
```

Example: Find the names of all branches that have greater assets than some branch located in Brooklyn.

```
SELECT DISTINCT T.branch_name  FROM branch AS T, branch AS S
        WHERE T.assets > S.assets AND S.branch_city = 'Brooklyn' ;
```

SQL Operators

- SQL operators are **used mainly in the WHERE clause and HAVING clause to filter the data to be selected.**

Comparison Operators:

- **Comparison operators are used to compare the column data with specific values in a condition.**
- **Comparison Operators are also used along with the SELECT statement to filter data based on specific conditions.**

COMPARISON OPERATORS	DESCRIPTION
=	equal to
<>, !=	is not equal to
<	less than
>	greater than
>=	greater than or equal to
<=	less than or equal to

Logical Operators:

- Logical Operators compare two conditions at a time to determine whether a row can be selected for the output. There are three Logical Operators namely, AND, OR, and NOT.
- When retrieving data using a SELECT statement, we can use logical operators in the WHERE clause, which allows to combine more than one condition.

LOGICAL OPERATORS	DESCRIPTION
OR	For the row to be selected at least one of the conditions must be true.
AND	For a row to be selected all the specified conditions must be true.
NOT	For a row to be selected the specified condition must be false.

"OR" Logical Operator:

If you want to **select rows that satisfy at least one of the given conditions**, you can use the logical operator, **OR**.

Example Find the names of students who are studying either Maths or Science

```
SELECT first_name, last_name, subject FROM student_details
        WHERE subject = 'Maths' OR subject = 'Science' ;
```

back

Logical Operators:

"AND" Logical Operator:

If you want to select rows that must satisfy all the given conditions, you can use the logical operator, AND.

Example Find the names of the students between the age 10 to 15 years

```
SELECT first_name, last_name, age FROM student_details WHERE age >= 10 AND age <= 15;
```

"NOT" Logical Operator:

If we want to find rows that do not satisfy a condition, we can use the logical operator NOT.
NOT results in the reverse of a condition.
That is, if a condition is satisfied, then the row is not returned.

Example: To find the names of the students **who do not play football**,:

```
SELECT first_name, last_name, games FROM student_details
    WHERE NOT games = 'Football' ;
```

Note : ***SQL COMMANDS ARE NOT CASE SENSITIVE. BUT,***
THE DATA USED TO COMPARE IS CASE SENSITIVE

Nested Logical Operators:

We can use multiple logical operators in an SQL statement.

The order in which question frame the condition is important, if the order changes we are likely to get a different result.

When we combine the logical operators in a SELECT statement, the order in which the statement is processed is

1) NOT

2) AND

3) OR

Example: To select the names of the students who age is between 10 and 15 years, or those who do not play football,

```
SELECT first_name, last_name, age, games FROM student_details
        WHERE age >= 10 AND age <= 15 OR NOT games = 'Football'
```

Here,

Condition 1: All the students you do not play football are selected.

Condition 2: All the students whose are aged between 10 and 15 are selected.

Condition 3: Finally the result is, the rows which satisfy atleast one of the above conditions is returned.

SQL Comparison Operators

Comparison Operators	Description
LIKE	column value is similar to specified character(s).
IN	column value is equal to any one of a specified set of values.
BETWEEN...AND	column value is between two values, including the end values specified in the range.
IS NULL	column value does not exist.

SQL LIKE Operator

The LIKE operator is used **to list all rows in a table whose column values match a specified pattern.** It is **USEFUL WHEN YOU WANT TO SEARCH ROWS TO MATCH A SPECIFIC PATTERN, OR WHEN YOU DO NOT KNOW THE ENTIRE VALUE.**

SQL includes a **string-matching operator for comparisons on character strings.**

The operator "like" uses patterns that are described using two special characters:

percent (%) The **% character matches any substring.**

underscore (_) The **_** character matches any single character.

Example: To find the names of all customers whose street includes the substring "Main".

```
SELECT customer_name FROM customer WHERE customer_street LIKE '% Main%' ;
```

- For patterns to include the *special pattern characters (that is, % and-)*, **SQL allows with the specification of an ESCAPE CHARACTER.**
  - ***THE ESCAPE CHARACTER IS USED IMMEDIATELY BEFORE A SPECIAL PATTERN CHARACTER TO INDICATE THAT THE SPECIAL PATTERN CHARACTER IS TO BE TREATED LIKE A NORMAL CHARACTER. WE DEFINE THE ESCAPE CHARACTER FOR A LIKE COMPARISON USING THE ESCAPE KEYWORD.***

- ***SQL allows us to search for mismatches instead of matches by using the NOT LIKE comparison operator.***

Example:

Consider the following patterns, which use a backslash (\) as the escape character:

like 'ab\%cd%' escape '\' **matches all strings beginning with "ab%cd"**

like 'ab \\cd%' escape '\' **matches all strings beginning with "ab\cd"**

SQL BETWEEN ... AND Operator

The operator BETWEEN and AND, are **used to compare data for a range of values.**

Here the bounds are inclusive.

Example:

To find the names of the students between age 10 to 15 years,

```
SELECT first_name, last_name, age FROM student_details  WHERE age BETWEEN 10 AND 15;
```

SQL IN Operator:

The IN operator is used **when you want to compare a column value with more than one value.** It is **similar to an OR condition**. The **comparison process ends when the first match is found.**

Example: To find the names of students who are studying either Maths or Science, the query would be like,

```
SELECT first_name, last_name, subject FROM student_details
                    WHERE subject IN ('Maths', 'Science');
```

Note : ***SQL COMMANDS ARE NOT CASE SENSITIVE. BUT, THE DATA USED TO COMPARE IS CASE SENSITIVE***

SQL IS NULL Operator

A column value is NULL if it does not exist.

The **IS NULL operator can be used to display all the rows for columns that do not have a value.**

Example: To find the names of students who do not participate in any games

```
SELECT first_name, last_name FROM student_details WHERE games IS NULL ;
```

There would be no output if every student participates in a game in the table student_details, else the names of the students who do not participate in any games would be displayed.

SQL NOT IN

The NOT IN operator is used when you want to retrieve a column that has no entries in the table or referencing table.

Example: To find the names of customers who have not done any transactions

A customer table will be containing records of all the customers and the transaction table keeps the records of any transaction between the store and the customer.

Customers table:

Cust_id	first_name	last_name	subject

Transaction table:

Transaction_ID	Cust_id	Product_ID	Amount

```
SELECT first_name, last_name, cust_id FROM customer
                    WHERE cust_id NOT IN ( SELECT cust_id FROM transactions);
```

AGGREGATE FUNCTIONS - SQL GROUP Functions

Group functions are built-in SQL functions that operate on groups of rows and return one value for the entire group.
These functions are: **COUNT, MAX, MIN, AVG, SUM, DISTINCT**

SQL COUNT (): This function **returns the number of rows in the table that satisfies the condition specified in the WHERE condition. If the WHERE condition is not specified, then the query returns the total number of rows in the table.**

Example: To find the number of employees in a particular department:

```
SELECT COUNT (*) FROM employee
        WHERE dept = 'Electronics';
```

SQL DISTINCT(): This function is used **to select the distinct rows.**

Example: To find all distinct department names from employee table:

```
SELECT DISTINCT dept FROM employee;
```

Example: To get the count of employees with unique name:

```
SELECT COUNT (DISTINCT name) FROM employee;
```

SQL MAX(): This function is used **to get the maximum value from a column.**

Example: To get the maximum salary drawn by an employee:

```
SELECT MAX (salary) FROM employee;
```

SQL MIN(): This function is used **to get the minimum value from a column.**

Example: To get the minimum salary drawn by an employee:

```
SELECT MIN (salary) FROM employee;
```

SQL AVG(): This function is used **to get the average value of a numeric column.**

Example: To get the average salary

```
SELECT AVG (salary) FROM employee;
```

SQL SUM(): This function is used **to get the sum of a numeric column**

Example: To get the total salary given out to the employees,

```
SELECT SUM (salary) FROM employee;
```

SUB-QUERIES

- Subquery / Inner Query / Nested Query is a QUERY IN A QUERY.
- A sub-query is a select-from-where expression that is nested within another query.
- SQL subquery is usually added in the WHERE Clause of the SQL statement.
- Generally, A subquery is used when we know how to search for a value using a SELECT statement, but do not know the exact value in the database.
- Subqueries are an alternate way of returning data from multiple tables.
- Subqueries can be used with the following SQL statements along with the comparison operators like =, <, >, >=, <= etc. SELECT, INSERT, UPDATE, DELETE
- Usually, a sub query should return only one record, but sometimes it can also return multiple records when used with operators LIKE, IN, NOT IN in the WHERE clause.
- A COMMON USE OF SUB-QUERIES is to PERFORM TESTS FOR SET MEMBERSHIP, SET COMPARISONS, AND SET CARDINALITY.
- WE CAN NEST AS MANY QUERIES WE WANT but it is recommended *NOT TO NEST MORE THAN 16 SUBQUERIES IN ORACLE*
- *IN SQL STATEMENT WITH SUB QUERY, FIRST THE INNER QUERY IS PROCESSED AND THEN THE OUTER QUERY IS PROCESSED.*

NESTED SUB-QUERIES - Set Membership

- SQL allows testing tuples for membership in a relation.
- The *IN* connective tests for set membership, where the set is a collection of values produced by a SELECT clause.
- The *NOT IN* connective tests for the absence of set membership.

"IN" Construct ("IN" operator / "IN" connective)

Examples:

Find all customers who have both an account and a loan at the bank.

```
SELECT DISTINCT customer_name  FROM borrower
        WHERE customer_name IN (SELECT customer_name  FROM depositor );
```

Find all customers who have a loan at the bank but do not have an account at the bank

```
SELECT DISTINCT customer_name FROM borrower
        WHERE customer_name NOT IN (SELECT customer_name  FROM depositor );
```

Find all customers who have both an account and a loan at the Perryridge branch

```
SELECT DISTINCT customer_name FROM borrower, loan
    WHERE borrower.loan_number = loan.loan_number AND
                branch_name = 'Perryridge'  AND
              (branch_name, customer_name )  IN
          (SELECT branch_name, customer_name  FROM depositor, account
              WHERE depositor.account_number =    account.account_number )
```

student_details(id first_name, last_name, age, subject, games)
branch(branch_name, branch_city, assets)
customer(customer_name, customer_street, customer_city)
loan(loan_number, branch_name, amount)
borrower(customer_name, loan_number)
account(account_number, branch_name, balance)
depositor(customer_name, account_number

NESTED SUB-QUERIES - Set Membership

"SOME" Construct

Find all branches that have greater assets than some branch located in Brooklyn.

```
SELECT branch_name FROM branch
    WHERE assets > SOME (SELECT assets FROM branch WHERE branch_city = 'Brooklyn')
```

Note : SQL also allows < SOME, <= SOME, = SOME, and <> SOME comparisons.

"ALL" Construct

Find the names of all branches that have greater assets than all branches located in Brooklyn.

```
SELECT branch_name FROM branch
    WHERE assets > ALL (SELECT assets FROM branch WHERE branch_city = 'Brooklyn') ;
```

NESTED SUB-QUERIES - Set Comparisons

Test for Empty Relations

"EXISTS" Construct

SQL includes a feature **for testing whether a subquery has any tuples in its result.**

The **EXISTS construct returns the value true if the argument subquery is nonempty.**

Example:

"Find all customers who have-both an account and a loan at the bank".

```
SELECT customer_name FROM borrower
WHERE EXISTS (SELECT *  FROM depositor
                    WHERE depositor.customer_name= borrower.customer_name )
```

NESTED SUB-QUERIES

Absence of Duplicate Tuples

"UNIQUE" Construct

The unique construct **tests whether a subquery has any duplicate tuples in its result.**

Examples:

Find all customers who have at most one account at the Perryridge branch.

```
SELECT T.customer_name   FROM depositor as T
        WHERE UNIQUE (SELECT R.customer_name   FROM account, depositor AS R
                WHERE T.customer_name = R.customer_name
                        AND
                        R.account_number = account.account_number
                        AND
                        account.branch_name = 'Perryridge')
```

NESTED SUB-QUERIES

Find all customers who have at least two accounts at the Perryridge branch.

```
SELECT DISTINCT T.customer_name FROM depositor AS T
    WHERE NOT UNIQUE (  SELECT R.customer_name FROM account, depositor AS R
                            WHERE  T.customer_name = R.customer_name
                                AND
                            R.account_number = account.account_number
                                AND
                            account.branch_name = 'Perryridge')
```

NOTE : *VARIABLE FROM OUTER LEVEL IS KNOWN AS A CORRELATION VARIABLE*

NESTED SUB-QUERIES - DERIVED RELATIONS

SQL allows a subquery expression to be used in the from clause

Find the average account balance of those branches where the average account balance is greater than $1200.

```
SELECT branch_name, avg_balance FROM
        (SELECT branch_name, AVG (balance)  FROM account GROUP BY  branch_name ) AS
        branch_avg ( branch_name, avg_balance )   WHERE avg_balance > 1200;
```

NOTE :

We do not need to use the having clause, since we compute the temporary relation branch_avg in the from clause, and the attributes of branch_avg can be used directly in the where clause (avg_balance).

NON-CORELATED SUBQUERY :

If a subquery that is not dependent on the outer query, then it is called as NON-CORRELATED SUBQUERY

CORRELATED SUBQUERY (*Synchronized Subquery*)

- **A correlated / synchronized subquery is a subquery that uses values from the outer query.**
  - **Since the subquery may be evaluated once for each row processed by the outer query, it can be inefficient.**
- **Correlated subqueries are used for row-by-row processing. Each subquery is executed once for every row of the outer query.**
- A correlated subquery is evaluated once for each row processed by the parent statement. The parent statement can be a **SELECT**, **UPDATE**, or **DELETE** statement.
- A correlated subquery is one way of reading every row in a table and comparing values in each row against related data. It is used whenever a subquery must return a different result or set of results for each candidate row considered by the main query. In other words, you can use a correlated subquery to answer a multipart question whose answer depends on the value in each row processed by the parent statement.

CORRELATED SUBQUERY (*Synchronized Subquery*)

Syntax :

```
SELECT column1, column2, ....  FROM table1 outer
        WHERE column1 operator  (SELECT column1, column2 FROM table2
                                 WHERE expr1 =   outer.expr2);
```

Example - 1:

"Find all customers who have-both an account and a loan at the bank".

```
SELECT customer_name FROM borrower
    WHERE customer_name  =   (SELECT customer_name  FROM depositor
                              WHERE depositor.customer_name= borrower.customer_name );
```

The outer query is:

```
SELECT customer_name FROM borrower   WHERE customer_name  = ...
```

The inner query (the Correlated Subquery) is:

```
SELECT customer_name  FROM depositor  WHERE depositor.customer_name= borrower.customer_name
```

Here, THE INNER QUERY HAS TO BE RE-EXECUTED FOR EACH BORROWER (EACH ROW PROCESSED BY THE OUTER QUERY).

CORRELATED SUBQUERY (*Synchronized Subquery*)

Example - 2:

To find all employees whose salary is above average for their department

```
SELECT employee_number, name FROM employees AS emp
        WHERE salary > ( SELECT AVG(salary) FROM employees
                         WHERE department = emp.department);
```

The outer query is:

```
SELECT employee_number, name FROM employees AS emp WHERE salary > ...
```

The inner query (the Correlated Subquery) is:

```
SELECT AVG(salary) FROM employees WHERE department = emp.department
```

Here, THE INNER QUERY HAS TO BE RE-EXECUTED FOR EACH EMPLOYEE (EACH ROW PROCESSED BY THE OUTER QUERY).

Nested Subqueries Versus Correlated Subqueries :

With a normal **Nested Subquery**, **the inner SELECT query runs first and executes once, returning values to be used by the main query**.

BUT

A Correlated Subquery, however, executes once for each candidate row considered by the outer query.

In other words, the inner query is driven by the outer query.

NOTE :

You can also use the ANY and ALL operator in a correlated subquery.

JOINS

WHY WE NEED JOINS

In a database where the tables are normalized, **one table may not give all the information about a particular entity. Hence we need to join two or more tables for comprehensive data analysis.**

QUALITIES OF A GOOD JOIN

- The **query must be based on a primary key or a foreign key**. They are more logical and useful.
- To produce a meaningful result, **the columns being compared should have compatible datatypes and they need not have the same name.**

JOINS IN SQL

- SQL Joins are **used to relate information in different tables.**
- The SQL Syntax for joining two tables is:

```
SELECT col1, col2, col3.. FROM table_name1, table_name2
                    WHERE table_name1.col2 = table_name2.col1;
```

- **If a SQL join condition is omitted or if it is invalid the join operation will result in a Cartesian product.**
  - ✓ The Cartesian product returns a number of rows equal to the product of all rows in all the tables being joined.
- A **Join Condition** is a part of the SQL query that **retrieves attributes values in related rows from two or more tables.**
  - ✓ Join condition is used in the WHERE Clause of **Select, Update, Delete** statements.

TYPES OF JOINS

- **THETA JOIN –** Theta Join is a join between 2 tables R and S, that defines a table which contains rows satisfying a predicate and is denoted by R.ai θ S.bi where θ may be one of the comparison operators.
  - ✓ The Comparison Operator in a join condition need not be equality alone.
    1. If the comparison operator is EQUALITY, then the join is called an EQUIJOIN.
       - ▶ An Equijoin is further classified into two categories:

         a) SQL Inner Join b) SQL Outer Join
    2. If the comparison operator is NOT EQUALITY, then it is a NON-EQUIJOIN.
- **NATURAL JOIN -** The Equijoin will produce a result containing the primary key column of the first table and the foreign key column of the second table, which are identical. If one of the columns is eliminated, then the join is called the NATURAL JOIN.
- **SELF-JOIN –** When a table is joined to itself, it is called as Self-Join
  - ✓ It is generally used to compare values within a column of a single table.

SELF JOIN: As the name signifies, in SELF JOIN a table is joined to itself. That is, each row of the table is joined with itself and all other rows depending on some conditions. In other words we can say that it is a join between two copies of the same table. Here we create a copy of the same table and have different alias names for each copy.

Example –

```
SELECT a.ROLL_NO , b.NAME FROM Student a, Student b
WHERE a.ROLL_NO < b.ROLL_NO;
```

EQUI JOIN :

EQUI JOIN creates a JOIN for equality or matching column(s) values of the relative tables. EQUI JOIN also create JOIN by using JOIN with ON and then providing the names of the columns with their relative tables to check equality using equal sign (=).

Example –

```
SELECT student.name, student.id, record.class, record.city
FROM student, record  WHERE student.city = record.city;
```

(or)

```
SELECT student.name, student.id, record.class, record.city
FROM student JOIN record ON student.city = record.city;
```

NON EQUI JOIN :

NON EQUI JOIN performs a JOIN using comparison operator other than equal(=) sign like >, <, >=, <= with conditions.

Example –

```
SELECT student.name, record.id, record.city FROM student, record
WHERE Student.id < Record.id ;
```

Natural Join :

Natural Join joins two tables based on same attribute name and datatypes. The resulting table will contain all the attributes of both the table but keep only one copy of each common column.

Example –

```
SELECT s.STUDENT _ID, std_mark FROM STUDENT s, MARKS m
WHERE s.STUDENT_ID = m.STUDENT_ID;
```

Inner Join

This is the most common type of join. It combines the results of one or more tables and displays the results when all the filter conditions are met.

Example –

```
SELECT STUDENT _ID, std_mark FROM STUDENT s, MARKS m
WHERE s.STUDENT_ID = m.STUDENT_ID;
```

(or)

```
SELECT STUDENT _ID, std_mark  FROM STUDENT s
 INNER JOIN MARKS   m ON s.STUDENT_ID = m.STUDENT_ID;
```

Here it combines the two tables STUDENT and MARKS. It displays the result only if there is matching students in both the tables. In the above query we can even specify simply 'JOIN' instead of INNER JOIN to represent the same.

Difference between Natural Join and Inner Join

In **Natural Join**, The resulting table will contain all the attributes of both the tables but keep only one copy of each common column	In **Inner Join**, The resulting table will contain all the attribute of both the tables including duplicate columns also

Left Outer Join

This join retrieves all the records from the table which is on the LHS of 'LEFT OUTER JOIN' clause and only the matching records from RHS.

Below example of left outer join on DEPT and EMPLOYEE table combines the matching combination of DEPT_ID = 10 with values. But DEPT_ID = 30 does not have any employees yet. Hence it displays NULL for those employees. Thus this outer join makes more meaningful to combining two relations than a cartesian product.

Example –

```
SELECT d.DEPT_ID, d.DEPT_NAME,    e.EMP_ID,  e.ENAME,  e.DEPT_ID
FROM DEPT d, EMP e   WHERE d.DEPT_ID = e.DEPT_ID (+);
```

(or)

```
SELECT d.DEPT_ID, d.DEPT_NAME,    e.EMP_ID,  e.ENAME, e.DEPT_ID
FROM DEPT d LEFT OUTER JOIN EMP e ON d.DEPT_ID = e.DEPT_ID;
```

DEPT

DEPT_ID	DEPT_NAME
10	Account
20	Design
30	Testing

Left Outer Join

EMPLOYEE

EMP_ID	ENAME	DEPT_ID
100	James	10
101	Kathy	20
103	Rose	10
104	Marry	20

DEPT_ID	DEPT_NAME	EMP_ID	ENAME	DEPT_ID
10	Account	100	James	10
10	Account	103	Rose	10
20	Design	101	Kathy	20
20	Design	104	Marry	20
30	Testing			

Right Outer Join

This join is opposite of left outer join. It retrieves all the records from the table which is on the RHS of 'RIGHT OUTER JOIN' clause and only the matching records from LHS.

Example –

```
SELECT d.DEPT_ID, d.DEPT_NAME,    e.EMP_ID, e.ENAME, e.DEPT_ID
FROM EMP e, DEPT d  WHERE e.DEPT_ID (+) = d.DEPT_ID;
```

(or)

```
SELECT d.DEPT_ID, d.DEPT_NAME,    e.EMP_ID, e.ENAME, e.DEPT_ID
FROM EMP e RIGHT OUTER JOIN DEPT d ON e.DEPT_ID = d.DEPT_ID;
```

EMPLOYEE		
EMP_ID	ENAME	DEPT_ID
100	James	10
101	Kathy	20
103	Rose	10
104	Marry	20

Right Outer Join

DEPT	
DEPT_ID	DEPT_NAME
10	Account
20	Design
30	Testing

EMP_ID	ENAME	DEPT_ID	DEPT_ID	DEPT_NAME
100	James	10	10	Account
101	Kathy	20	20	Design
103	Rose	10	10	Account
104	Marry	20	20	Design
			30	Testing

Full Outer Join

This is combination of both left and outer join. It displays all the matching columns from both the tables, and if it does not find any matching row, it displays NULL.

Example –

```
SELECT d.DEPT_ID, d.DEPT_NAME, e.EMP_ID, e.ENAME, e.DEPT_ID
FROM EMP e FULL OUTER JOIN DEPT d ON e.DEPT_ID = d.DEPT_ID;
```

EMPLOYEE		
EMP_ID	ENAME	DEPT_ID
100	James	10
101	Kathy	20
103	Rose	10
104	Marry	20
105	Alex	40

Full Outer Join

DEPT	
DEPT_ID	DEPT_NAME
10	Account
20	Design
30	Testing

EMP_ID	ENAME	DEPT_ID	DEPT_ID	DEPT_NAME
100	James	10	10	Account
101	Kathy	20	20	Design
103	Rose	10	10	Account
104	Marry	20	20	Design
105	Alex	40		
			30	Testing

VIEWS

- A view is stored as a Database Object in the database.
- View is the result set of a stored query. This pre-established query command is kept in the Database Dictionary.
- Views do not contain data of their own, but we can query a view as we do with a table in the database.
- View is a Virtual Table computed or collated dynamically from data in the database when access to that view is requested.
- Changes applied to the data in a relevant underlying table are reflected in the data shown in subsequent invocations of the view.
- Views are used to restrict access to the database or to hide data complexity.

USES OF VIEWS (Advantages of Views over tables)

- Views can join and simplify multiple tables into a single virtual table.
- Views can act as aggregated tables, where the database engine aggregates data (sum, average, etc.) and presents the calculated results as part of the data.
- Views can hide the complexity of data.
- Views take very little space to store; the database contains only the definition of a view, not a copy of all the data that it presents.
- Views can represent a subset of the data contained in a table.
  - Therefore, a view can limit the degree of exposure of the underlying tables to the outer world.
  - A given user may have permission to query the view, while denied access to the rest of the base table.

Example :

Consider a user who needs to know a customer's name, loan number and branch name, but has no need to see the loan amount. Define a view and grant the user permission to read the view, but not the underlying tables.

```
CREATE VIEW cust_loan_data AS
    SELECT customer_name, borrower.loan_number, branch_name FROM borrower, loan
                                    WHERE borrower.loan_number = loan.loan_number ;
```

SIMPLE AND COMPLEX VIEWS

- **Simple views can only contain a single base table.**
- **Complex views can be constructed on more than one base table.**
  - ✓ **Complex views can contain: join conditions, a group by clause, a order by clause.**

SIMPLE VIEW	COMPLEX VIEW
Contains only one single base table or is created from only one table. Differences between Simple and Complex View	Contains more than one base tables or is created from more than one tables.
We cannot use group functions like MAX(), COUNT(), etc.	We can use group functions.
Does not contain groups of data.	It can contain groups of data.
DML operations could be performed through a simple view.	DML operations could not always be performed through a complex view.
INSERT, DELETE and UPDATE are directly possible on a simple view.	We cannot apply INSERT, DELETE and UPDATE on complex view directly.
Simple view does not contain group by, distinct, pseudocolumn like rownum, columns defiend by expressions.	It can contain group by, distinct, pseudocolumn like rownum, columns defiend by expressions.
Does not include NOT NULL columns from base tables.	NOT NULL columns that are not selected by simple view can be included in complex view.

CREATING A VIEW

- ***A view is defined using the create view statement which has the form :***

CREATE VIEW V AS < query expression >

where <query expression> is any legal SQL expression and V is the view name.

Example for creation of Simple View (view based on one table)

```
CREATE VIEW sal5 AS
SELECT employee_id ID_NUMBER, last_name NAME ,salary*12 ANN_SALARY  FROM emp
                                                    WHERE dept_id=50;
```

Example for creation of Complex View (view based on more than one table)

```
CREATE VIEW empview AS  SELECT *  FROM emp , dept
                                    WHERE emp.deptno = dept.deptno ;
```

VIEW MANIPULATION

View Execution

- When a view is created the query expression alone is stored in the database along with the view name .
- The query expression in view definition is executed whenever the View name is referred.

Querying through a View

- Once a view is defined, the view name can be used to refer to the virtual relation that the view generates and can be queried.
- To query the underlying tables through a view, create the view and use the view name as an ordinary table name in Select query.

Example :

To find details of all employees of department with deptno 50

```
SELECT * FROM empview WHERE deptno = 50 ;
```

VIEW MANIPULATION - **UPDATING THROUGH VIEW**

Updating through View

- **To update the underlying tables through a view, create the view and use the view name as an ordinary table name in Insert or Update queries.**
- **Most SQL implementations allow updates only on simple views (without aggregates) .**

Example: **Create a view of all loan data in the loan relation, hiding the amount attribute**

```
CREATE VIEW loan_branch AS SELECT loan_number, branch_name FROM loan;
```

Add a new tuple to loan_branch

```
INSERT INTO loan_branch VALUES ('L-37', 'Avadi') ;
```

This results in the insertion of the tuple ('L-37', 'Avadi', null) into the loan relation

VIEW MANIPULATION

Problems in Updating Through Views

- **Some updates through views are impossible to translate into updates on the database relations. OTHERS CANNOT BE TRANSLATED UNIQUELY**

```
CREATE VIEW V AS  SELECT loan_number, branch_name, amount  FROM loan
                  WHERE branch_name = 'Avadi' ;
```

```
INSERT INTO V VALUES  ( 'L-99','Downtown', '23') ;
```

Others cannot be translated uniquely

```
INSERT INTO all_customer VALUES ('Avadi', 'John');
```

have to choose loan or account, and create a new loan/account number!

VIEW MANIPULATION

Restrictions on updating data through Views

You can insert, update, and delete rows in a view, subject to the following limitations:

- **If the view contains joins between multiple tables, you can only insert and update one table in the view, and you can't delete rows.**
- **You can't directly modify data in views based on union queries. You can't modify data in views that use GROUP BY or DISTINCT statements.**
- **All columns being modified are subject to the same restrictions as if the statements were being executed directly against the base table.**
- **Text and image columns can't be modified through views.**