Contents

[Key points: 3](#_Toc212589539)

[Char Datatype: 6](#_Toc212589540)

[Numeric Datatype: 7](#_Toc212589541)

[Long Datatype: 7](#_Toc212589542)

[Date Datatype: 7](#_Toc212589543)

[RAW Datatype 8](#_Toc212589544)

[ROW ID DATATYPE 8](#_Toc212589545)

[Classification of SQL Statement. 9](#_Toc212589546)

[Constraints 10](#_Toc212589547)

[Grants and Revoke (DCL) 13](#_Toc212589548)

[Comments 13](#_Toc212589549)

[Creating a table in SQL (DDL) 13](#_Toc212589550)

[Alter Statement in SQL(DDL) 14](#_Toc212589551)

[Dropping a Table (DDL) 16](#_Toc212589552)

[TABLE SPACE AND DATAFILES 16](#_Toc212589553)

[Select Statement (DQL/DML) 20](#_Toc212589554)

[WHERE Clause 20](#_Toc212589555)

[Insert Statement (DML) 22](#_Toc212589556)

[Update Statement (DML) 22](#_Toc212589557)

[Delete Statement (DML) 23](#_Toc212589558)

[Truncate in SQL (DDL) 23](#_Toc212589559)

[TCL (Transaction Control Language) 25](#_Toc212589560)

[Merge Statement 26](#_Toc212589561)

[Order by clause 27](#_Toc212589562)

[SQL JOINS 29](#_Toc212589563)

[Natural Joins 31](#_Toc212589564)

[SQL SELF JOIN 35](#_Toc212589565)

[SQL EQUI JOIN 36](#_Toc212589566)

[LEFT OUTER JOIN IN SQL 37](#_Toc212589567)

[RIGHT OUTER JOIN IN SQL 38](#_Toc212589568)

[FULL OUTER JOIN IN SQL 40](#_Toc212589569)

[CROSS-JOIN IN SQL 41](#_Toc212589570)

[INNER JOIN IN SQL 42](#_Toc212589571)

[SET Operations 45](#_Toc212589572)

[UNION [ALL] 45](#_Toc212589573)

[MINUS 46](#_Toc212589574)

[INTERSECT 47](#_Toc212589575)

[Comparison Operators 48](#_Toc212589576)

[Equal = 49](#_Toc212589577)

[Not Equal != or <> 49](#_Toc212589578)

[Greater than > 49](#_Toc212589579)

[Less than < 49](#_Toc212589580)

[Greater than or equal to >= 50](#_Toc212589581)

[Less than or equal to <= 50](#_Toc212589582)

[Logical Operators 50](#_Toc212589583)

[AND 51](#_Toc212589584)

[OR 51](#_Toc212589585)

[NOT 51](#_Toc212589586)

[More SQL operators 52](#_Toc212589587)

[NOT 52](#_Toc212589588)

[[NOT] BETWEEN 53](#_Toc212589589)

[[NOT] IN 53](#_Toc212589590)

[[NOT] EXISTS 54](#_Toc212589591)

[IS [NOT] NULL 59](#_Toc212589592)

[LIKE 60](#_Toc212589593)

[\*Wildcards 62](#_Toc212589594)

[ERRORS 65](#_Toc212589595)

Introduction to Oracle SQL

# Key points:

**Data** refers to raw facts, figures, or symbols that can be processed and analyzed to extract useful information. Information is obtained by measuring, analyzing, and observing phenomenon. In digital world data is stored in the form of digital bits. Data can be divided into two forms based on its structure; Structured and Unstructured data.

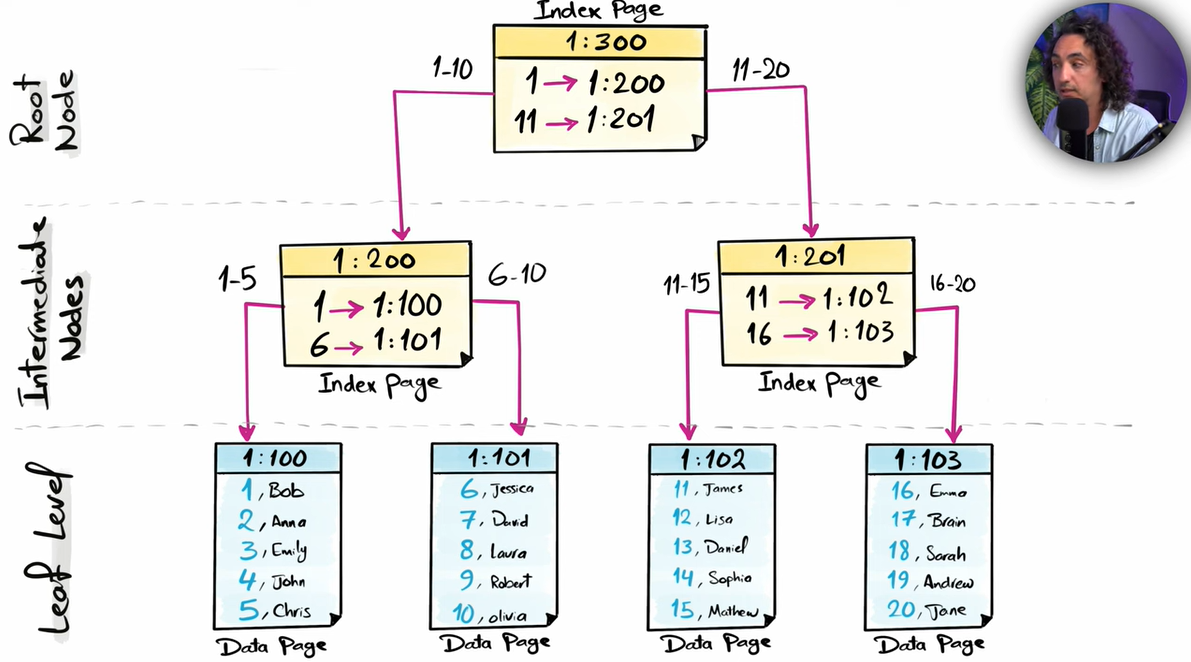
Structured data refers to data that can be organized into rows and columns. Structured data is also referred to as searchable data.

Unstructured data on the other hand is difficult to deal with because it doesn’t follow a specific pattern or doesn’t fit in any predefined model.

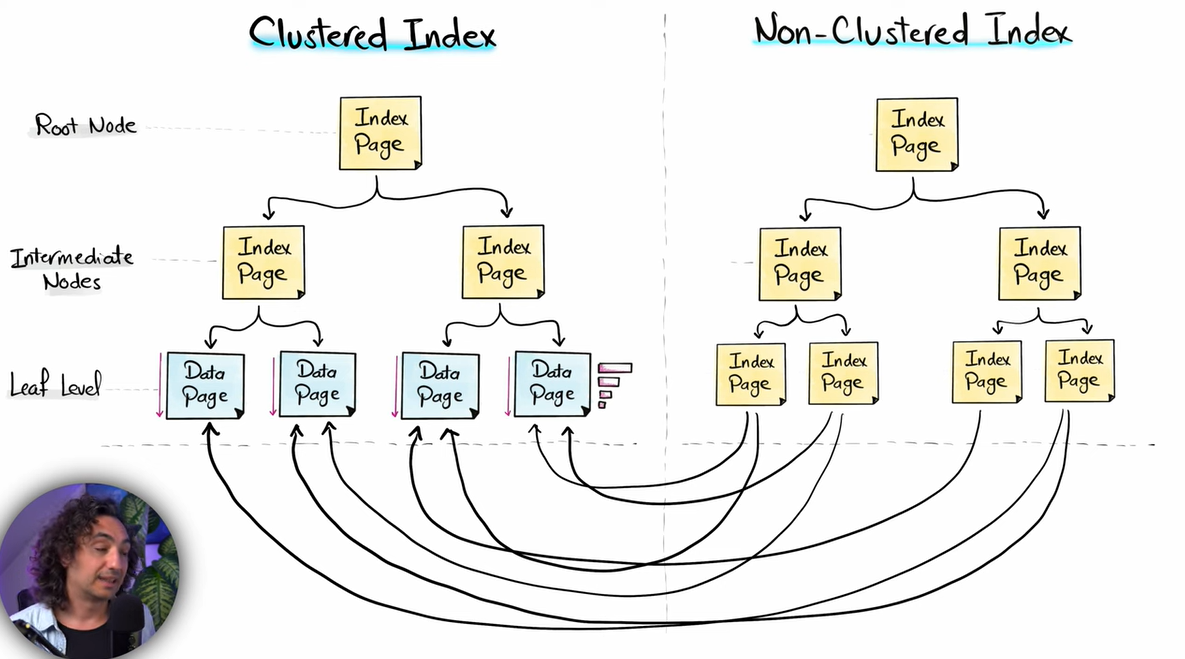
**Database** is something where we store, organize and manage information. Databases are built to store data efficiently and retrieve it quickly. In database data is stored in pages in datafile(datafile.mdf). A page is the smallest unit of data storage in a database(8kb), SQL can store anything inside it. A datafile contains datapages where rows are stored. Oracle Database was co-founded in 1977 by Larry Ellison.

//**Heap:** A table without any clustered index. Because Heaps save data in random order they are fast write, slow read data structures. SQL needs to search entire table to find an entry. //

**Clustered index**: when a clustered index is created in a column, SQL automatically sorts the entries in the column in the data page. Incase of numbers the rows will be arranged from the lowest to the highest. But that’s not all, SQL also creates a Balance tree(B-tree) for the entries; Entries in the B-tree are stored in the leaf notes. Intermediate nodes are usually index pages. Index pages do not contain data rather the contain pointers to data pages where data is located. Root note contains pointers to intermediate index notes. A user can create only one clustered index in a table, because physically the data can only be sorted in one way. Clustered indexes are faster at reading and slower at writing data.



**Non-Clustered indexes:** Unlike clustered indexes, leaf nodes in non clustered indexes contain index pages that point to data pages. Non-Clustered indexes are not concerned wheather the data being accessed isn sorted or not. Leaf nodes in non-clustered indexes usually point to data pages of clustrered indexes. A single table can have multiple non-clustered indexes. Non-clustered indexes are faster at writing and slower at reading data.



**ACID** stands for Atomicity, Consistency, Isolation, Durability. ACID defines a set of properties that ensure consistency of a database.

**Atomicity:** All transactions are performed or none is performed.

**Consistency:** The transaction takes the database from one consistent state to another.

**Isolation:** The effect of a transaction is not visible to other transactions until the transaction is committed.

**Durability:** Changes made by committed transactions are permanent. Database makes sure that changes from the transaction are not lost.

A database contains all the users you’ve created and their data.

**Oracle user and Schema**: In Oracle, Schema is a collection of logical data structures or schema objects. Schema objects are user created structures that directly refer to database objects. A database user owns a database schema with the same name as the user. A Schema is essentially a user’s workspace.

**Relational database** stores information in the form of tables; rows, and columns.

Applications retrieve information from the database, backend acts as a bridge between these two. **Frontend 🡪 Backend 🡪 Database**

There are mainly two types of databases.

* **Relational Database**
* **NoSQL Database**

**Oracle** works on relational databases, and SQL (Structured Query Language) is used to manage these databases.

**NoSQL** databases use a more flexible approach to storing data as compared to relational databases. Information here is stored in documents, and files instead of tables.

**Data Types in SQL:**

* Numeric
* Long
* Char
* Date
* Raw
* Row ID

A datatype imposes some rule/restriction on a particular column. It tells what type of data to store. Datatypes are applied while creating a table.

# Char Datatype:

**Char(n):** Stores fixed length strings, Oracle reserves space for exactly **n** characters, even if the actual value is shorter. Remaining spaces are padded with blank spaces. 2000 bytes max. Each character takes 1 byte.

**VARCHAR2(n):** Stores only the number of characters actually inserted. 4000 bytes max.

**NCHAR(n):** Same as Char, but for Unicode characters. Each character takes 2 bytes.

**NVARCHAR2(n):** Same as NVARCHAR2, but for Unicode characters.

Numeric Datatype:

**Number(p,s):** Commonly used to store numbers in Oracle. p refers to precision (the number of digits input is limited to), and s refers to scale (number of digits after decimal point). If **Number** is used without scale or precision, it can take any input (any number of digits and no limit to numbers after decimal).

e.g Number (4,3) means that it would only store less than 5 digit number either without decimal or decimal included.

**Integer:** A sub-block of **Number**, used to store Integers only. i-e: Number (4)

**Float:** Sub-block of **Number**, used to store decimal numbers regardless of scale.

*Note: if an input exceeds the required limit of a datatype, SQL generates a limit error.*

# Long Datatype:

**LONG:** Like Char it long is used to store text data. Long is mostly used when text is too long and has a large number of characters. LONG could handle up to 2 GB of data.

Limitations: Only one LONG can be used in a column.

Cannot be used with Where, Join and most other functions.

# Date Datatype:

Date: Date datatype is used to store both date and time in Oracle. Oracle stores date in the following format: DD/MM/YYYY /HH/MM/SS

**TIMESTAMP**: Extended version of Date, adds fractional seconds for more precision.

**TIMESTAMP with TIME ZONE:** Precision upto 6-digit fraction of a seconds with any time zone.

**TIMESTAMP with LOCAL TIME ZONE:** Precision upto 6-digit fraction of a seconds with local server time zone.

# RAW Datatype

**RAW:** Raw datatype is used to store binary data (data in bytes not readable text).

Encrypted values, digital signatures, etc. It can store only 2000 byte of data.

Why use RAW instead of CHAR?

Oracle may modify text in order to make it compatible across different platforms, storing binary data in CHAR can be a risk of losing the integrity of stored information. In contrast if data is stored as RAW, Oracle keeps the structure as it is.

**BLOB (Binary Large Object):** Stores very large binary data (Images, Pdfs, videos). Used when a file size exceeds 2000 bytes. Can store upto 128 GBs depending on the system.

**CLOB (Character Large Object):** Stores large amount of text data, it can typically store up to 8 GBs of character data. Unlike LONG multiple CLOB can be used in a column.

**NCLOB (National Large Object):** Same as CLOB, but for multilingual text.

# ROW ID DATATYPE

**ROWID:**  Every row in an Oracle table is assigned a rowid. ROWID is typically used to access a row faster. Oracle automatically assigns rowid to every row in a table which can’t be modified manually.

Each rowid is unique and point to the exact row in the table.

Updating a row does not change the rowid, although operations like deleting and inserting new row, importing or exporting data, or moving the row to another table and rebuild it.

**ROWIDs** are stored inside the data block of the table.

# Classification of SQL Statement.

**SQL (Structured Query Language):** SQL is the standard language used to interact with relational databases. SQL is used to control, and manipulate a database.

SQL is classified into following categories based on their operation:

* DDL (Data Definition Language)
* DML (Data Manipulation Language)
* DCL (Data Control Language)
* TCL (Transaction Control Language)
* DQL (Data Query Language)

**DDL (Data Definition Language):** DDL is used to change the definition/structure of a table. Commands such as **CREATE**, **ALTER** and **DROP** are included in DDL. Database commits the transactions before and after DDL statements. In Oracle database DDL commands come with an implicit commit, which means that its execution will commit all preceding DML commands also. We don’t deal with data at any instance to this point, DDL commands are used to define the structure for data.

**DML (Data Manipulation Language):** DML is usually used to manipulate data within a table. This includes **INSERT**, **UPDATE**, and **DELETE** operations. User needs to execute commit explicitly, to make changes permanent.

**DCL (Data Control Language):** DCL is used to control access and permissions in a table. **GRANT** and **REVOKE** are included in DCL.

**TCL (Transaction Control Language):** TCL is used to when saving or revoking transaction in a database. TCL uses operations such as **COMMIT**, **ROLLBACK**, and **SAVEPOINT**. Rollback and commit act as end points of any transaction. Commit/Rollback by a user changes data for all users using the database.

**DRL/DQL (Data Retrieval Language/ Data Query Language):** DQL is usually used to view data, or retrieve something from the data rather than alter a part of it. **SELECT** command is included in DRL.

# Constraints

Rules defined by user to allow only valid values in a column. Constraints control how and what data should be entered. Constrains are usually applied during column creation. Some constraints are defined below;

**Not Null:** No Null value allowed in the column.

**Unique Constraint:** Do not allow duplicate values in a column, or combination of column. Null values are allowed here, and **one null value is not equal to other.** A table may have multiple unique columns.

**Primary key:** Primary key is one of the most important factors of any table. It is usually defined as a column or group of columns within a table. Primary keys need to be unique between columns and cannot contain null values. A table could only have one primary key.

**Difference between primary and unique Constraint:**

On surface level the mainly differ on the basis of usage within a table, Unique constraints can be applied on multiple columns, on the other hand there can only be one primary key. Alternately, the differ in terms of how these are stored in indexes;

Index gets automatically generated when primary key or unique constraints are defined in a table. The play a vital role in SQL, indexes help improve the overall performance and the data retrieval timing. Keys are cached in a structure that allow SQL servers to access rows attached with these keys efficiently. Indexes are of two types mainly, Clustered and non-clustered indexes. Defining a primary key automatically creates a clustered index, while adding a unique constraint creates a non-clustered index.

**Foreign key:** Foreign key constraint is applied when a column in one table is a primary key of another table. Foreign key acts as a bridge between the tables. The inclusion of foreign key in a table establishes a database constraint rule that the foreign key column in the child table should always be updated with values already present in the parent table of the key. Foreign key help keep the database consistent.

**Candidate Key:** A candidate key is an attribute or a set of attributes in a database table that can uniquely identify each row. There can be multiple candidate keys in a table, and they are called "candidates" because any one of them could be chosen as the primary key.

**Check Constraints:** check constraints are applied to control input values. It specifies a boolean expression that must be true for every row. Check constraints are usually applied to keep values in ranges, checks are applied in number columns to keep values in ranges, and date column to define dates. Check constraint passes unknown results.

Conditions of check constraints cannot contain the following constructs:

* Subqueries and scalar subquery expressions
* Calls to the functions that are **not deterministic** (CURRENT\_DATE,  
  CURRENT\_TIMESTAMP, DBTIMEZONE,  
  LOCALTIMESTAMP, SESSIONTIMEZONE,  
  **SYSDATE**, SYSTIMESTAMP, UID, USER, and USERENV)
* Calls to user-defined functions
* Dereferencing of REF columns (for example, using the DEREF function)
* Nested table columns or attributes
* The pseudocolumns CURRVAL, NEXTVAL, LEVEL, or ROWNUM
* Date constants that are not fully specified

**Default Constraint:** Default constraint applies a default value to a column when no value is applied explicitly, during insertion.

**User defined Constraints:** User defined constraints refer to specific rules defined by user which cannot be implemented using pre-defined constraints. User defined constraints usually involve a series of check statements combined to keep inputs in control.

**Informational Constraints:** Informational constraints are rules which developers follow, these are not specifically implemented by the database itself.

**Defining Constraint in a table**

Constraints can be applied at column level; **Inline constraints**, or they can also be applied at table level; **out-of-line constraints**. Constraints applied on multiple columns should be applied on table level.

Not null constraint must be applied inline.

# Grants and Revoke (DCL)

**Grants** are provided by one user to another, these usually include schema permissions. GRANT command is used to give specific permissions to users, or roles. Grant permissions include SELECT, UPDATE, INSERT, and DELETE. These privileges can be assigned to different database objects, like views, tables and schema’s.

GRANT <List of grants separated by commas> on <TABLE\_NAME> to <USER>

**Revoke:** Revoke as the name signifies, is used to revoke grants/permissions provided to users/roles using GRANT command.

REVOKE <List of grants separated by comma> on <TABLE\_NAME> from <USER>

# Comments

**Comments** acts as notes for future developers. Comments make a code more efficient to understand.

COMMENT ON TABLE <Table\_Name> is “<comment>”;

COMMENT ON COLUMN <Table\_Name.Column.Name> is “<comment>”;

# Creating a table in SQL (DDL)

A table may include following entities;

Table name, Column name, Column data type, Constraints, Grants, Comment on table and column, and datafiles. Table creation is perhaps the most important part of database creation process.

Oracle restrictions on naming table and column.

* Table names cannot be the same in a single schema.
* Table name should not exceed 128 characters.
* A table should have at least one column at the time of creation.
* Table name cannot start with a digit.
* Table should not include a space in its name, if its absolutely necessary then it should be wrapped in double quotes.
* Table name is automatically capitalized by oracle, it can be avoided by double quote wrapping the table name while creating, ideally double quote wrapping in not recommended.

Oracle capitalizes table and column names before executing a query.

CREATE command is used to create a table in SQL.

CREATE <Table Name> (

Column <Datatype>,

Column <Datatype>,

Column <Datatype>

)

CREATE TABLE schema\_name.Table\_name (

Column\_1 <Data\_Type><Default\_value><Column\_level\_constraints>

Column\_2<Data\_Type><Default\_Value><Column\_level\_constraints>

Table\_Constraint

);

*Note: Use all uppercase letters for writing commands.*

# Alter Statement in SQL(DDL)

Alter statement is used for modification in SQL. This includes adding column, removing column, and renaming column. Alternatively, it can be used to modify a columns datatype and removing or modifying constraint in a column.

**Datatypes**

ALTER TABLE <TABLE NAME> ADD <COLUMN NAME> <DATATYPE>;

ALTER TABLE <TABLE NAME> MODIFY <COLUMN NAME> <DATATYPE>;

ALTER TABLE <TABLE NAME> DROP COLUMN <COLUMN NAME>;

ALTER TABLE <TABLE NAME> RENAME <COLUMN NAME > TO <COLUMN NAME>;

**Constraints**

ALTER TABLE <TABLE NAME> ADD CONSTRAINT <CONSTRAINT\_NAME> <change\_expression>;

ALTER TABLE <TABLE NAME> MODIFY <CONSTRAINT\_NAME> <change\_expression>;

ALTER TABLE <TABLE NAME> DROP COLUMN <COLUMN CONSTRAINT>;

ALTER TABLE <TABLE\_NAME> DROP CONSTRAINT <CONSTRAINT NAME>;

ALTER TABLE RENAME COLUMN <COLUMN\_NAME> TO <COLUMN\_NAME>;

ALTER TABLE <TABLE\_NAME> RENAME TO <TABLE\_NAME>;

ALTER TABLE <TABLE\_NAME> MODIFY <COLUMN\_NAME> NOT NULL;

**Renaming a Table/Object**

Renaming can be performed by a database or table owner, however there are certain instances where renaming a table is not permitted, a few of such being:

* When there is some trigger attached to the table.
* When a View is linked to the table.
* Or when a foreign key refers to the table.

Any above case and the renaming operation will generate an error.

RENAME <ExistingTableName> to <NewTableName>

# Dropping a Table (DDL)

DROP command is used for dropping a table, with consequences being the deletion of everything in the table; any view or synonym attached becomes invalid. Only users with permission and DBA owners can perform Drop operation. It should be kept in mind that, unlike previous alter operations DROP operation is irreversible!

DROP TABLE <Table Name>;

Table with special relationships cannot be dropped using just drop command, rather some keywords are used along with drop command for affects to take place;

* **CASCADE CONSTRAINTS**
* **PURGE**

CASCADE CONSTRAINTS Clause is used when dropping a table that has foreign keys referencing its primary or unique key from other tables. Drop command without cascade constraints will fail with an error, if referential integrity constraints exist.

Purge: When purge is specified, the table and its associated objects (like indexes) are immediately and permanently removed from the database. They are not moved to recycle bin and cannot be recovered later.

# TABLE SPACE AND DATAFILES

Oracle **tablespaces** are essential for organizing data in a database. They act like different containers to store data, helping keep everything neat and running smoothly.

A **tablespace** is a logical storage container for organizing data. Think of a tablespace like a big box where different kinds of data are stored. Instead of throwing all your data into one place, Oracle uses tablespaces to help separate and manage different types of information.

**Why Tablespaces?**

* **Organization**: Tablespaces help keep data organized.
* **Efficiency**: By spreading data across different tablespaces, you improve database performance.
* **Management**: It’s easier to allocate and manage disk space.

When you create an Oracle database, two important tablespaces are automatically created: **SYSTEM** and **SYSAUX**.

**SYSTEM Tablespace**

* The **SYSTEM tablespace** is like the brain of the database. It contains the **data dictionary**, which stores vital information about how the database operates — things like user permissions, table definitions, and indexing structures.
* This tablespace is crucial for the database to function, and without it, the database wouldn’t know how to manage its own data.

**SYSAUX Tablespace**

* The **SYSAUX tablespace** acts as a helper for the SYSTEM tablespace. It offloads some of the extra work so the SYSTEM tablespace isn’t overloaded.
* It stores components like:
* **Performance monitoring** data (e.g., Oracle Enterprise Manager repository).
* **Automatic Workload Repository (AWR)** data, which helps collect performance statistics.
* Other auxiliary information that helps the database run efficiently.

Together, SYSTEM and SYSAUX keep the database organized and functioning properly by handling all the important behind-the-scenes data.

**User Tablespaces**

While SYSTEM and SYSAUX handle the database’s internal operations, **user tablespaces** are where you store the data that matters to your applications, things like customer records, sales data, and employee details.

**Why Use User Tablespaces?**

* **Data organization**: You can create separate tablespaces for different types of data or for different departments (e.g., HR, Sales).
* **Security**: You can apply specific security settings to protect sensitive data.
* **Performance**: You can spread data across tablespaces to improve performance and balance the load on the database.

For example, you might create a **user tablespace** called EMPLOYEES to store all employee-related data.

**4. UNDO and TEMP Tablespaces**

Oracle databases also use two special types of tablespaces to manage temporary data and transactions:

**UNDO Tablespace**

* The **UNDO tablespace** is like an eraser. It stores information that allows the database to undo or roll back changes if needed. This is especially useful if a transaction fails, or if you want to reverse a mistake.
* It also helps ensure read consistency so that users querying data see a stable view of the data, even when others are making changes.

**TEMP Tablespace**

* The **TEMP tablespace** is like scratch paper. When Oracle needs to do big calculations (like sorting or joining data), it uses TEMP tablespace to store temporary results. Once the operation is finished, the TEMP space is cleared out, ready for the next task.
* This helps improve performance by giving Oracle space to handle large queries or sorts.

**5. Extents and Segments**

Now, let’s zoom in further to see how data is actually stored inside these tablespaces.

**Segments**

* A **segment** is a collection of data related to a specific object, like a table or an index. Think of a segment as a group of shelves inside a box (tablespace), where each shelf holds a specific type of data (like table rows or index entries).
* **Types of segments**:
* **Table segments**: Store data for tables.
* **Index segments**: Store data for indexes that speed up queries.
* **Undo segments**: Store undo information for transactions.
* **Temporary segments**: Store temporary data during query processing.

**Extents**

* A **segment** is made up of smaller pieces called **extents**. An **extent** is a continuous chunk of storage space inside a tablespace. When a segment (like a table) needs more space, Oracle allocates a new extent for it.
* Think of an extent like a shelf in a cabinet (tablespace) that holds a specific group of items (data). As more data is added, more shelves (extents) can be added to the segment.

So, the flow is like this:

1. **Tablespace**: The overall storage area (box).
2. **Segments**: Groups of related data inside the tablespace (shelves).
3. **Extents**: Smaller pieces of storage within each segment (individual sections of a shelf).

Oracle allocates and manages extents as needed, ensuring that your data objects (tables, indexes) have enough room to grow within the tablespace.

**6. Adding Data Files to a Tablespace**

Just like you might need to add more shelves when you run out of space, you can expand a tablespace by adding **data files**. These are physical files on your disk that store the data from a tablespace. When a tablespace gets full, you can add more data files to give it additional capacity.

Databases 🡪 Table Spaces 🡪 Segments 🡪 Extents 🡪 Data

SELECT \* FROM user\_users;

# Select Statement (DQL/DML)

Select statement is used to retrieve data from one or more tables in the database. User can retrieve desired data by applying different clauses along with SELECT statement.

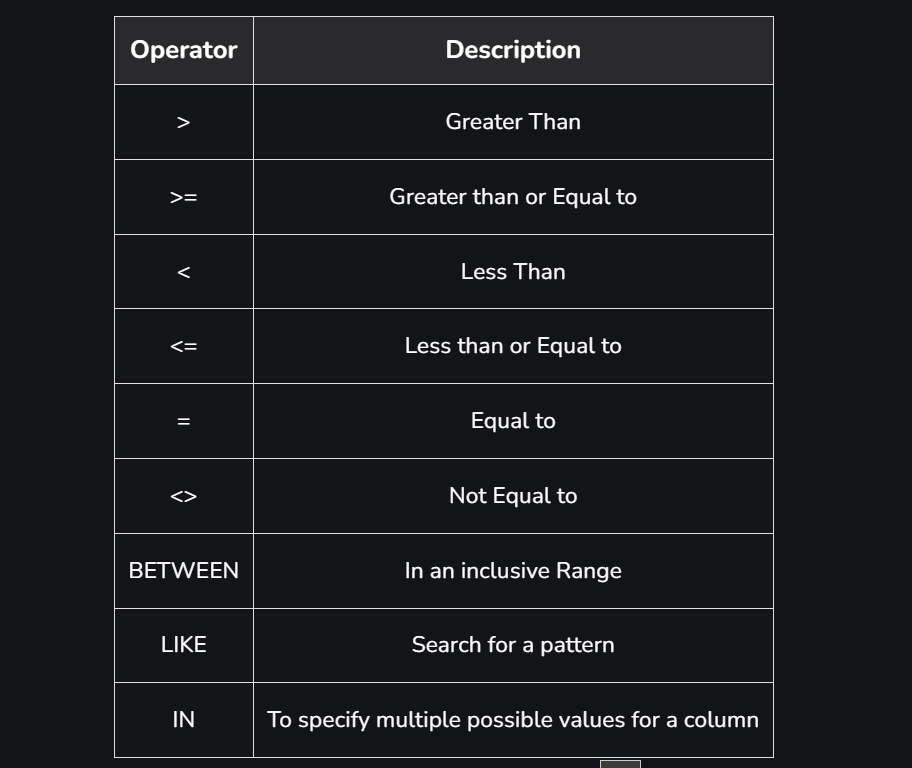
SELECT <Column names separated by comma’s> FROM <Table\_Name>

WHERE <Condition’s separated by comma’s>

# WHERE Clause

The WHERE clause is used to filter rows based on specific conditions. Whether you are retrieving, updating, or deleting data, WHERE ensures that only relevant records are affected. Without it, your query applies to every row in the table! The WHERE clause helps you:

* Filter rows that meet certain conditions
* Target specific data using logical, comparison and pattern-based operators
* Control SELECT, UPDATE, DELETE or even INSERT statements



SELECT \* FROM <TABLE\_NAME>   
WHERE < COLUMN\_NAME > BETWEEN <n> AND <n>;

SELECT \* FROM <TABLE\_NAME> WHERE Name LIKE 'S%';

SELECT Name FROM <TABLE\_NAME> WHERE Age IN (21,23);

# Insert Statement (DML)

Insert statement is used to insert data into a table. It can specify column names and values, or just values if all columns are added in the same order.

INSERT INTO <Table\_Name> ( <Column names separated by comma’s> )

VALUES ( <Values list separated by commas> ).

Above statement is used to add a single record in a table.

Some points to keep in mind while inserting data in table:

* A NOT NULL column should always be passed a value, otherwise the query will fail.
* Referential integrity constraints should also be kept in mind.
* Primary key values should not be repeated!

# Update Statement (DML)

Update statement is Data manipulation command used to modify/update rows/data within a table. It allows you to change the values of one or more columns for one or more rows. WHERE clause is mostly used along UPDATE command to modify certain value in a row.

UPDATE <Table\_Name>

SET <Column\_Name> = <expression/value>,

<Column\_Name> = <expression/value>,

…

WHERE <WHERE Conditions>;

# Delete Statement (DML)

DELETE statement is used to delete/remove rows from a table, while retaining the table structure intact. It is different from drop, which deletes an entire table.

* Removes rows based on conditions.
* Retains table schema, constraints, and indexes.
* Can delete a single row or all rows.
* Useful for cleaning or managing datasets.

WHERE Clause is used along to specify rows that need to be deleted, we can delete a single are multiple rows according to the WHERE Clause. If no WHERE condition is specified, all the rows are deleted.

DELETE FROM <Table\_Name>

WHERE <Condition…>;

Since DELETE is a rollback operation, it can be rolled back when executed in a statement.

Quick Note: Multiple Columns are always separated using commas, whilst multiple conditions are applied using AND or OR statement.

# Truncate in SQL (DDL)

Truncate command is used to delete data in a table, without deleting the table.

Truncate deletes the whole data in one shot at once unlike DELETE that removes data row by row. TRUNCATE command is used for this operation. No WHERE clause is used. Truncate operation is irreversible and can only be performed by table owner or database administrator.

An easier way to put TRUNCATE statement is, that it drops and recreates the table its applied on, rather than deleting the data row by row, thus therefore it is termed as a DDL command.

TRUNCATE TABLE <Table\_Name>

HOW is **TRUNCATE** Different from **DELETE** statement, especially for condition below:  
DELETE FROM <Table\_Name>;

TRUNCATE TABLE <Table\_Name>;

* TRUNCATE is a DDL Command thus once applied it can’t be rolled back, while changes made using DELETE (DML) command can be rolled back.
* DELETE removes data row by row sequentially, thus it is slower in contrast TRUNCATE deletes the data in one go.
* DELETE is slower when dealing with large data because the database has to prepare a copy for a possible rollback, TRUNCATE does not support rollback therefore, it saves resources and time.

TRUNCATE Statement will not work if integrity constraints are present in a table but it **depending on data.** If data doesn’t violate constraint integrity after truncate operation, TRUNCATE is allowed. DROP on the other hand is not allowed at all if integrity constraints are present.

**Integrity constraints and their relation with DROP, TRUNCATE, and DELETE Statements.**

Suppose we have two tables named **sample\_table\_1** and **sample\_table\_2,** with sample\_table\_1 links to sample\_table\_2 using a foreign key.

**Case 1: No data in both tables:**

1. **DROP** is not allowed on table\_sample\_1(parent), since an integrity constraint exists.

Solution:

* Drop child table(sample\_table\_2) first, then drop parent table(sample\_table\_1).
* Remove foreign key constraint in sample\_table\_2(child), then drop.

1. **TRUNCATE**: Since the tables are both empty and no data to violate integrity constraints, **TRUNCATE** is allowed in this case.
2. **DELETE**: With no data present to delete, DELETE statement is allowed but not applicable in this case.

**Case 2: Data present in both tables:**

1. **DROP:** DROP is not allowed (same as case 1).
2. **TRUNCATE:** Now that there is data in the tables which is linked through integrity constraints, TRUNCATE is not allowed, as it can affect database integrity.
3. **DELETE:** DELETE is allowed in sample\_table\_2(parent table) for only values in primary key that are not present in the child table.

# TCL (Transaction Control Language)

TCL mainly includes three operations:

Commit, Rollback and Savepoint.

**Commit:** COMMIT Statement is used to permanently save transactions made in a session. Once, committed the changes are visible to all other users. Commit cannot be rolled back.

**Rollback:** Rollback is used to undo certain changes made in a database; rollback command does not reverse DDL operations.

**Savepoint:** Serves as a checkpoint/snapshot of a state of the database which the rollback command can rollback to.

# Merge Statement

Merge statement also called UPSERT (Update and Insert) statement is used to combine multiple DML statements (Insert, Delete, update) together, for a variable output. Merge statement typically uses two tables, a target and a source table, which are compared through a common column. Target table is updated using source table according to conditions specified. The target table must be local to the database to which the current session is connected, but you can specify a remote table as a source table, or in subqueries of the SET clause for UPDATE operations, and in subqueries of the VALUES clause for INSERT operations.

**MERGE INTO** <Table\_Name> <alias>

USING <Table\_Name> <alias>

ON (<merge\_condition>)

**WHEN MATCHED THEN**

UPDATE SET <operation1, operation2, …..>

WHERE <condition>

/DELETE WHERE <Delete\_Condition>

**WHEN NOT MATCHED THEN**

INSERT (col1, col2, …..)

VALUES (value1, value2, …..)

WHERE <conditions……….>

# Order by clause

Order by clause is used with select statement. SELECT statement by default returns unordered data, ORDER BY clause is used along to order data, different expressions like ASC/DESC or NULLS FIRST/NULLS LAST could also be used along the clause according to user requirements.

SELECT \* FROM <Table\_Name>

ORDER BY <Column\_Name>; -------No expression provided, data returned will be ordered in ascending order by default.

SELECT \* FROM <Table\_Name>

ORDER BY <Column\_Name> ASC|DESC; --------Ordering according to expression;

SELECT \* FROM <Table\_Name>

ORDER BY <Column\_1> ASC | DESC , <Column\_2> ASC|DESC;

---------With two columns provided, ordering will be made according to column\_1 first and incase of repetitive values, column\_2 will be considered for ordering.

Note: Instead of column name, column number can also be used.

SELECT \* FROM <Table\_Name>

ORDER BY <1>, <4>;

SELECT \* FROM <Table\_Name>

ORDER BY <1>, <4> NULLS FIRST | NULLS LAST ;

--------------Ordering according to null values.

**Pagination:**

Pagination can be used limit the number of rows returned in the result set. FETCH Clause is used for this purpose, it is usually used along with order by clause to return specified number of rows. FETCH clause is also known as row limiting clause, it can also be combined with offset clause.

SELECT \* FROM <Table\_Name>

ORDER BY <Column\_Name> DESC -------------Displays first row only

FETCH FIRST ROW **ONLY**;

SELECT \* FROM <Table\_Name>

ORDER BY <Column\_Name> DESC ---Displays all matching rows at the position.

FETCH FIRST ROW **WITH TIES**;

**OFFSET Clause**

The OFFSET clause is used to specify a number N of initial query results that should be skipped (not returned to the application). N is computed by an expression that may be a single integer literal, or a single external variable, or any expression which is built from literals and external variables and returns a single non-negative integer.

SELECT \* FROM <Table\_Name>

ORDER BY <Column\_Name> DESC

OFFSET <number of rows to skip> ROWS

FETCH NEXT ROW ONLY | WITH TIES;

NEXT and FIRST keywords return the same result when no OFFSET statement preceeds them. NEXT is usually used when FETCH is followed by OFFSET.

WITH TIES as discussed returns all matching rows that specify condition.

# SQL JOINS

Joins are used to logically combine rows in multiple tables using a common column between them. Join allow you to retrieve data that is spread across multiple tables and present it in a unified way.

When an SQL query features a join, the relational database management system (RDBMS) will search for data in all the tables you include within that join. It then combines the results and presents them in a new table.

**SQL Joins Are More Efficient**

At some point in your career, you’ll likely find yourself in a situation where you need to combine data from two or more tables. SQL joins to provide you with the most efficient way to do that.

Imagine you had a table that included the names of all the students in a grade and all of their classes. If you wanted to change one student’s name, you’d have to change it multiple times to ensure it’s consistent across all their different classes.

To accomplish this, you *could* separate all the data into a student table and a classes table.

This option would make it easier for you to update the student’s name. However, it wouldn’t make your job easier overall and would take a lot more time in the long run.

After you separated the tables to make the name change, you’d then have to put everything back together in a table that would help the database answer other important questions. With SQL joins, though, you have a faster option.

SQL joins will help you complete this task faster, producing the same results in less time.

**SQL Joins Maintain Normalized Databases**

Normalization is a process of organizing data in databases. It includes table creation and establishing relationships between those tables based on certain rules.

Normalization helps you to reduce data redundancy, allowing for fewer data anomalies when you delete or update records.

Reducing redundancy also prevents wasted disk space and saves you from many potential maintenance problems (which will also slow down your process and interfere with productivity).

SQL joins maintain normalized databases and save you from having to do extra work to fix redundancies, wasted space, etc.

**SQL Joins Reduce Database Workload**

SQL Joins also reduce the workload placed on the database because you can use one join query to get the same result as multiple queries. Why would you wear yourself (and the database) out with multiple steps when one step produces the same thing?

When you reduce the database workload, you can make better use of the database’s various functions (search, filter, sort, etc.).

You can also work more efficiently and reduce your chances of experiencing time-consuming (and potentially costly) maintenance issues, freeing yourself up to focus on other important aspects of your job.

*Note: Join is always* ***cross product + condition.***

**SQL Join Types**

The essential components of an SQL join include:

* At least two tables (except self-join)
* A specific join condition

The tables feature rows that you’ll combine, and the join condition provides instructions for which rows will be matched together.

There are several different types of SQL joins you can use to combine data from tables (the type you choose will depend on your goals). However, the following are some of the most commonly used joins you should be aware of:

## Natural Joins

A natural join is an SQL operation that combines two tables based on all columns that have the same name and data type in both tables. It is a type of equijoin, which means it works by matching equal values in common columns. We do not need to explicitly specify the column for joining, as SQL automatically takes care of finding the common column names and data types and joins the data.

The key features of a natural join are:

* It is an INNER JOIN by default that returns only matching rows between the tables.
* Tables are joined based on all columns with the same name and data types. The SQL engine automatically detects these common columns.
* Output contains unique columns; common join columns appear only once.
* ON or USING clauses cannot be used to specify join columns, as join uses all common column names implicitly.
* Can be combined with left, right, and full outer joins for more flexibility.
* If there are two or more common columns in tables being joined using a NATURAL JOIN, the join condition will implicitly include all of these common columns.
* For example, if you have two tables, **TableA** with columns (ID, Name, City) and **TableB** with columns (ID, City, Population), a NATURAL JOIN between **TableA** and **TableB** would implicitly create a join condition based on both ID and City.

**Syntax**

The syntax for a natural join in SQL is fairly simple and straightforward. It brings together two or more tables by just using the NATURAL JOIN keywords without needing to specify the actual join columns with the ON or USING clause. The basic syntax is:

**SELECT <column1, column2, column3,... column N>**

**FROM< Table\_Name1>**

**NATURAL JOIN <Table\_Name2>**

Natural Join can be opened up into:

**SELECT <Columns> FROM <Table\_>, <Table\_2>,…**

**<Table\_Name\_N> ---------- Cartesian product**

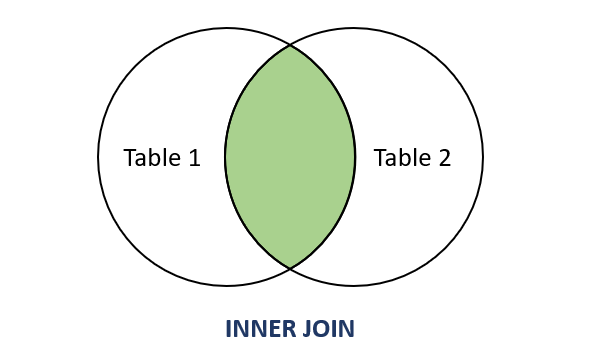
**WHERE <Table\_1.Common\_Column> = <Table\_n.Common\_Column>**

**SELECT \* FROM TableA  
NATURAL JOIN TableB ------------------------ Natural Join on more than 2 tables.  
NATURAL JOIN TableC;**

* The first NATURAL JOIN combines TableA and TableB based on their common columns. The result of this join is a temporary, intermediate table.
* The second NATURAL JOIN then takes this intermediate table and joins it with TableC, again matching on common column names and data types.

**What Is A Natural Join With No Common Attributes?**

A natural join on tables with no common columns results in a cartesian product where each row from one table is joined with every row of the other table.

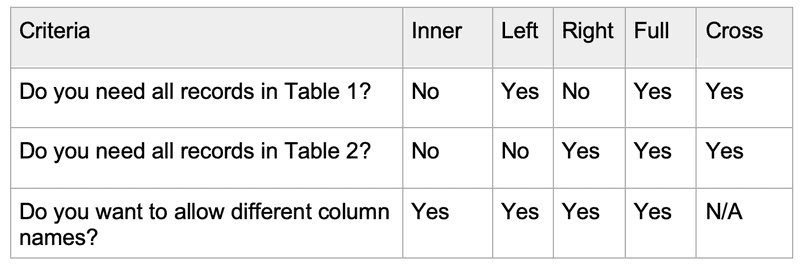


**How to Decide Which SQL Join to Use**

Sometimes, it’s obvious when you need to use an inner join, left join, etc. In other situations, though, you might be confused about the best way to proceed.

When that confusion arises, these guidelines can help:

* **Use an inner join when**: You don’t need to show all the records from the first table *or*all the records from the second table.
* **Use a left join when**: You need to show all the records from the first table, but you *don’t*need to show all the records from the second table.
* **Use a right join when**: You don’t need to show all the records from the first table, but you *do* need to show all the records from the second table.
* **Use a full join when**: You need to show all the records from the first and second tables, *and*you need to join the tables based on one or more columns.
* **Use a cross join when**: You need to show all the records from the first and second tables, but you *don’t* need to join the tables based on one or more columns.



## SQL SELF JOIN

A self-join joins a table to itself, creating a virtual second instance of that table that can be referenced as a different table. To perform a self-join, you need to alias the table references; otherwise, SQL will not know how to distinguish the two instances of the table.

The join predicates in a self-join work just like any other join predicate. You can join columns across the two instances of the table to compare values and return related rows.

**Syntax**

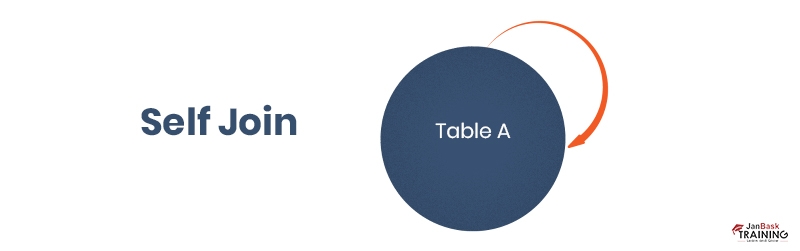
The self-join syntax involves aliasing the table references and joining them together, just like you would join two different tables.

**SELECT t1.column1, t2.column2**

**FROM table t1**

**JOIN table t2 ON t1.id = t2.id**

* SELECT t1.column1, t2.Column: You select the columns you want from each instance of the table. Use the aliases to distinguish columns from the two table instances.
* FROM table AS t1: defines the first reference to the table and aliases it as t1. This is the left side of the join.
* JOIN table AS t2: Joins the second reference to the table, aliasing it as t2. This right side will be joined to the left side.
* ON t1.id = t2.t1id: join predicate comparing the id column from the left table instance to the t1id column from the right instance. You can join any shared column.



So, in summary, you are creating two aliased references to the table that can then be treated as separate tables in the join syntax. The join predicate matches related rows from the two instances.

Find the **employee's name\*** who worked in **a department\*\*** having **location same as** **their address\*\*\***?

\* Find/Display/return usually refers to attributes/ columns which need to be stated at **SELECT** Statement level.

\*\* If the department and employees are on different tables, then it suggests we need a **JOIN** statement.

\*\*\* **location same as their address**: Suggests we need to apply a condition, mostly using **ON** statement.

## SQL EQUI JOIN

An Equi Join in SQL is a type of join that combines rows from two or more tables based on a common column or set of columns, using the equality operator = to compare column values.

It is one of the most frequently used types of joins because it matches columns from different tables that have equal values, allowing the combination of related data.

Remember that:

* The column names do not need to be the same.
* The resultant table contains repeated columns.
* It is possible to perform an equi join on more than two tables.
* **Equi join uses only equals sign.**

SYNATX

**SELECT \***

**FROM TableName1**

**JOIN TableName2**

**ON TableName1.ColumnName = TableName2.ColumnName;**

## LEFT OUTER JOIN IN SQL

A left outer join, also known as a left join, is an SQL command that combines two tables and returns all rows from the left table, along with matching rows from the right table. If there is no match in the right table, NULL values are returned for the columns from the right table.

In the syntax of a left outer join, the dominant table of the outer join appears to the left of the keyword that begins the outer join.

**Syntax**

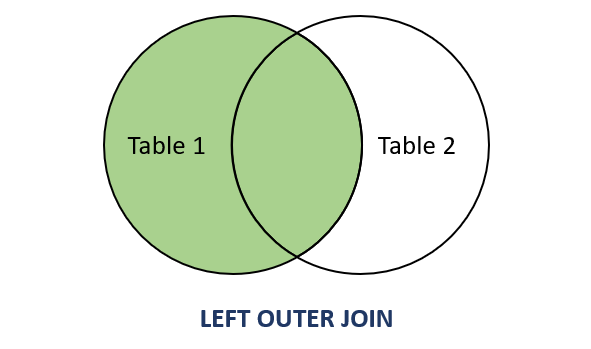
**SELECT \***

**FROM TableName1**

**LEFT OUTER JOIN TableName2 - ---**A **WHERE** Clause can also be used after **ON**.

**ON TableName1.ColumnName = TableName2.ColumnName;**

TABLE on left side of LEFT OUTER TABLE is dominant.



## RIGHT OUTER JOIN IN SQL

A Right Outer Join is a combined method in SQL that joins records from two separate tables together according to a given criteria, guaranteeing that every entry from an appropriate table can be found in the result set. The fundamental syntax involves defining the left database and specifying the requirement for joining using the ON clause, followed by the right table selection, the Right Join or Right Outer Join keywords, and so on. The prerequisite for the join specifies row matching between the two databases. A Right Outer Join in SQL returns every entry from the adjacent table as well as the identical entries from the table on the other side. For all columns in the left-hand table, the values of NULL are added wherever there is no result in the table on the left.

In the syntax of a right outer join, the dominant table of the outer join appears to the right of the keyword that begins the outer join.

**Syntax**

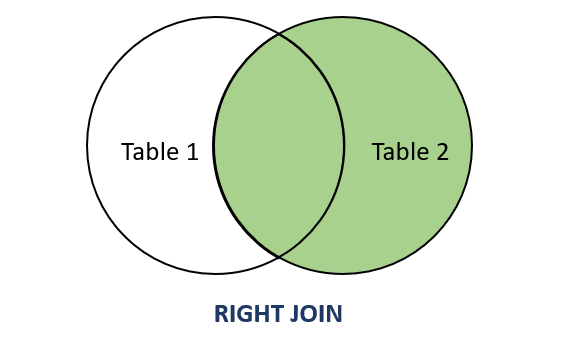
**SELECT \***

**FROM TableName1**

**RIGHT OUTER JOIN TableName2 - ---**A **WHERE** Clause can also be used after **ON**.

**ON TableName1.ColumnName = TableName2.ColumnName;**

TABLE on right side of RIGHT OUTER TABLE is dominant.



## FULL OUTER JOIN IN SQL

A Full Outer Join in SQL is like combining two lists and ensuring you see everything from both, even if something doesn't match exactly. You get all the items from both lists, and when something's missing in one, it shows up as a space.

In other words, it includes all records from both the left and right tables, and where there is no match, the columns from the side without a matching row will contain NULL values. This join type is useful when you want to retrieve all records from both tables, regardless of whether there is a match or not.

**Syntax**

**SELECT \***

**FROM table1**

**FULL OUTER JOIN table2 ON table1.column = table2.column;**

A Full Outer Join in SQL is employed when you want to retrieve all records from two tables, regardless of whether there is a match between the joined columns. This type of join is useful in various scenarios:

Inclusive Data Retrieval:

* When you want to include all rows from both tables in the result set.
* It ensures that no data is excluded, providing a comprehensive view of the data from both tables.

Handling Incomplete Data:

* In situations where the two tables may not have a perfect one-to-one match, a Full Outer Join allows you to capture all records and identify where matches occur while accommodating cases where no match is found.

Merging Datasets:

* When working with data from different sources or systems, a Full Outer Join can be useful for merging datasets with potentially incomplete or unmatched records.

## CROSS-JOIN IN SQL

A cross-join in SQL is a type of join operation that combines every row from one table with every row from another, resulting in a Cartesian product of the two tables. Unlike other join types, such as inner or outer, cross joins do not require any specific condition for matching rows between the tables. Instead, they generate all possible combinations of rows between the joined tables.

**Syntax**

**SELECT \***

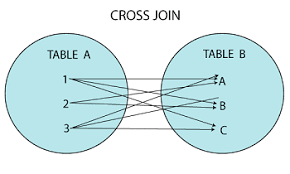
**FROM table1**

**CROSS JOIN table2;**

In this example, table1 and table2 are the names of the tables being joined. The result set of a cross join contains several rows equal to the product of the number of rows in both tables. If table1 has m rows and table2 has n rows, the result set will have m \* n rows.

Why Use Cross-Join In SQL?

Cross joins in SQL are used when you need to combine all possible combinations of rows from two tables, irrespective of any specific conditions or relationships between them. While cross-joins are less common than other join types, they serve specific purposes in certain scenarios:



## INNER JOIN IN SQL

An Inner Join in SQL is used to combine related rows of data from two or more tables based on a common column between them. The Inner Join creates a new merged result set by matching rows from the tables where the values are equal in the specified joining column. The query compares each row of one table with rows from another table to find all pairs that satisfy the Join condition defined in the ON clause. When the condition evaluates to true, column values from each input table are combined into the output result set table.

**Syntax**

**The basic syntax template for an SQL Inner Join is:**

**SELECT column\_list**

**FROM table1**

**INNER JOIN table 2**

**ON table1.join\_column = table2.join\_column;**

The ON condition specifies the common columns from each table that will be used to match rows and combine data. All rows from the tables for which the ON predicate evaluates to true will be included in the final result set. If no condition is specified in an Inner Join, the result will be a Cartesian product with a row for every possible combination of rows from the tables.

Why Use the Inner Join In SQL Queries?

There are several important reasons to utilize Inner Join in SQL operation queries:

1. Consolidate related data from multiple tables into a single result set. This avoids the need to run multiple separate queries and combine results in application code.
2. Filter unwanted rows that do not match the Join criteria. The Inner Join predicate removes non-matching rows through the inherent filtering action of matching data across tables.
3. Associate detailed child table records with parent table data, such as order details with customers. Joins form connections between primary and foreign keys.
4. Aggregate calculations can be performed across the joined results. When you combine tables, aggregated functions like SUM(), COUNT(), and AVG() can be applied.
5. Simplify complex queries by breaking logic across modular steps with incremental joins.

In summary, Inner Joins facilitates bringing related data together in SQL result sets, no matter how distributed it is across tables in the database schema.

**Difference Between Inner Join and Outer Join**

There are several advanced join techniques that are built upon Inner Joins with expanded capabilities:

* Outer Joins return all rows from one or both tables, matching rows when data exists but also preserving rows with NULL for non-matching data.
* Left Outer Joins return all rows from the "left" table, even if no match exists.
* Right Outer Joins return all rows from the "right" table if no match exists.
* Full Outer Joins return rows from both tables, appending NULLs if there is no match.
* Self Joins: A table joined to itself in order to analyze hierarchical data such as managers and employees, where the same table contains both types of data rows.
* Cross Joins: Joins every row from the first table with every row from the second table, multiplying rows for all combinations.

So, while Inner Joins only return strictly matching row pairs, discarding non-matching rows, Outer Joins, and Cross Joins have specialized logic around preserving non-matching results for inclusion as well.

## SET Operations

Set operations are used to combine the results of two queries. They act like mathematical set operations, letting us find the intersection, union, and difference between the rows returned by our queries. Queries with select statement are called **Compound Queries**.

Set operations include:

* **Union**: Merges all rows from two or more select statement eliminating **duplicates.**
* **Union All**: Merges all rows from two or more select statements, **keeping duplicates**. Union All is faster compared to Union for large data.
* **Intersect**: Returns only the rows that are common in both select statement.
* **Minus**: Returns rows from first statement that are not present in the second statement eliminating duplicates.

Set operation properties to keep in mind:

* Same number of columns and datatypes must be used while using SET Operators between two queries.
* Order of queries in not important when using Intersect, Union, Union All.
* Order by must be used for entire compound query, individual ordering of queries is not allowed.
* Set operators are not valid on column of type BLOB, CLOB, and Bfile.
* Set operators except Union All are not allowed on column type LONG.

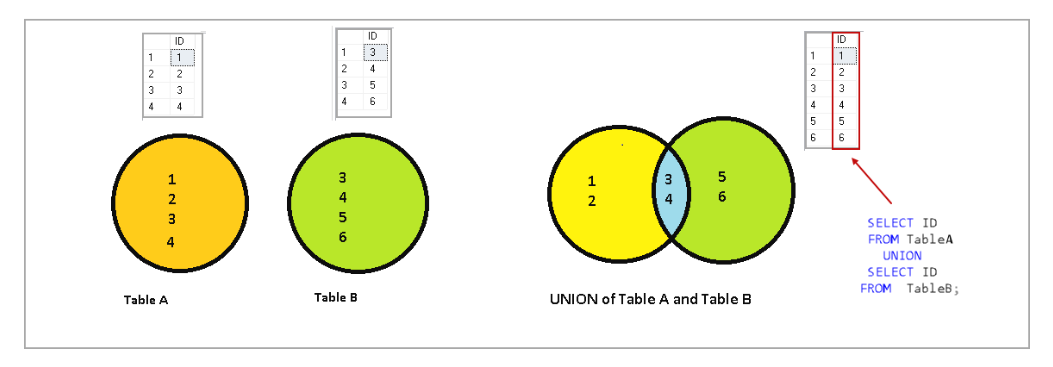
## UNION [ALL]

Union operator combines the result of two queries. It returns all the rows in table A and all the rows in table B. It automatically eliminates any repeating or duplicate rows. Union All works exactly like Union, with difference being it that it returns duplicates also, on the other hand in Union the rows are ordered on the basis of the first row while in Union All the order is random and predictable.

Syntax

SELECT <column\_list> FROM <Tables>

ORDER BY <order\_by\_column\_list>



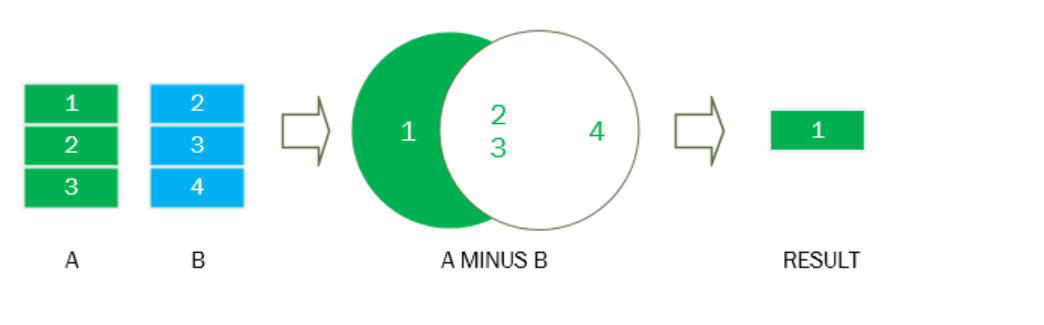
## MINUS

The Minus operator in SQL is used to return only those values in first query which are not present in subsequent query. It filters out all common values and returns unique values from first column.

Minus operator returns distinct results avoiding duplicates and orders by first column of first query by default.

Syntax

SELECT column1, column2, ..., columnN  
FROM table\_name1  
[WHERE condition]  
MINUS  
SELECT column1, column2, ..., columnN  
FROM table\_name2  
[WHERE condition];



## INTERSECT

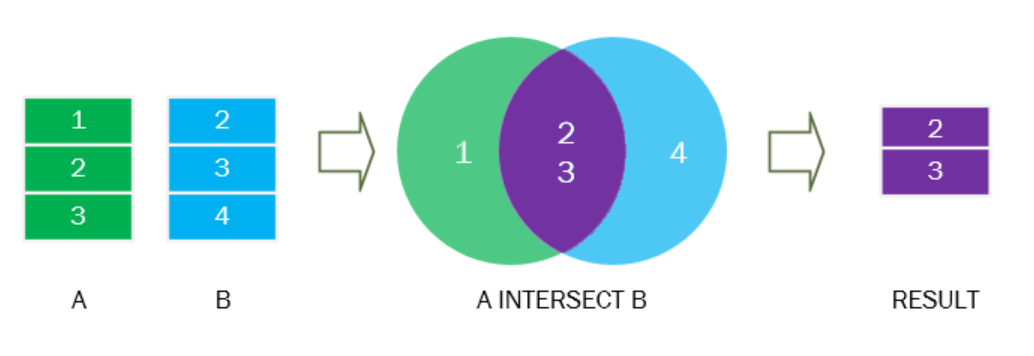
SQL Intersect operator combines two statements and returns only the dataset that is common in both the statements. Like other operators discussed above, Intersect operator also returns distinct results only ordered by first column.

Similar to the UNION, and MINUS operators in SQL, even the INTERSECT operator has to abide by the following two rules for successful execution:

* The number and order of columns in both queries has to be the same
* The data types of corresponding columns from both the select queries must be compatible with each other

Syntax

*SELECT column1 , column2 ....  
FROM table1  
WHERE condition****INTERSECT*** *SELECT column1 , column2 ....  
FROM table2  
WHERE condition*



# 

# Comparison Operators

Comparison operators sometimes referred to as relational or Boolean operators, compare values in a database and determine if they are equal (=), not equal (!= ,<>), greater than (>), less than (<), greater than or equal to (>=), less than or equal to (<=). They are used within SQL statements, such as in the WHERE clause.

## Equal =

In a SELECT statement, a WHERE clause could use the = operator to return rows where the value of a column is equal to a specified value, such as:

***SELECT column1, column2***

***FROM table***

***WHERE column2 = 'value';***

## Not Equal != or <>

In a SELECT statement, a WHERE clause could use the != or <> operator to return rows where the value of a column is not equal to a specified value, such as:

***SELECT column1, column2***

***FROM table***

***WHERE column2 != 'value';***

## Greater than >

In a SELECT statement, a WHERE clause could use the > operator to return rows where the value of a column is greater than a specified value, such as:

***SELECT column1, column2***

***FROM table***

***WHERE column2 > 'value';***

*Note: Greater than operators returns false if both operands are equal.*

## Less than <

In a SELECT statement, a WHERE clause could use the < operator to return rows where the value of a column is less than a specified value, such as:

***SELECT column1, column2***

***FROM table***

***WHERE column2 < 'value';***

*Note: Less than operators returns false if both operands are equal.*

## Greater than or equal to >=

In a SELECT statement, a WHERE clause could use the >= operator to return rows where the value of a column is greater than or equal to a specified value, such as:

***SELECT column1, column2***

***FROM table***

***WHERE column2 >= 'value';***

## Less than or equal to <=

In a SELECT statement, a WHERE clause could use the <= operator to return rows where the value of a column is less than or equal a specified value, such as:

***SELECT column1, column2***

***FROM table***

***WHERE column2 <= 'value';***

*Note: Comparison operator compares strings character by character from left to right.*

# 

# Logical Operators

SQL Logical Operators combine conditions in a WHERE clause to create more complex expressions.

These are the SQL Logical Operators:

* AND – Returns True if both conditions are True.
* OR – Returns True if either of the conditions is True.
* NOT – Returns true if the condition is false

## AND

The AND operator combines two conditions and returns TRUE if both conditions are TRUE.

**SELECT \***

**FROM products**

**WHERE prices > 50 AND category = 'dresses';**

In this example, the query retrieves all columns and rows from the products table where the prices are greater than 50, and the category is ‘dresses’. Only the products whose cost exceeds 50 and belong to the ‘dresses’ category will be in the result set.

## OR

The OR operator combines two conditions and returns TRUE if at least one condition is TRUE.

**SELECT \***

**FROM products**

**WHERE prices = 50 OR category = 'dresses';**

In this example, the query retrieves all columns and rows from the products table where prices are equal to 50 or the category is ‘dresses. As a result, you will see a mix of all the products that cost 50 and all dresses at any price.

## NOT

The NOT operator negates a condition and returns TRUE if the condition is FALSE.

**SELECT \***

**FROM products**

**WHERE NOT category = 'dresses';**

In this example, the NOT operator negates the condition, so only the products that do not belong in the ‘dresses’ category will be returned.

# More SQL operators

## NOT

The SQL NOT operator allows you to exclude specific rows from the results of a query, giving you more control over the data you retrieve. Excluding unwanted data can make your queries run more efficiently and faster.

The NOT operator negates a condition and returns TRUE if the condition is FALSE.

**SELECT \***

**FROM products**

**WHERE NOT category = 'dresses';**

In this example, the NOT operator negates the condition, so only the products that do not belong in the ‘dresses’ category will be returned.

**SELECT \***

**FROM products**

**WHERE NOT price > 10 AND quantity IS NULL;**

In this example, we combined the operators to return the products that cost less than 10 and does have an empty value for quantity.

## [NOT] BETWEEN

The SQL BETWEEN operator is used in the WHERE clause of a SQL query to specify a range of values to be returned. Between is an **inclusive** operator.

**NOT** Operator if used along BETWEEN however negates the condition (Select data except in range).

Examples;

**SELECT \***

**FROM table**

**WHERE price BETWEEN 50 AND 100;**

In this example, the query retrieves all columns and rows from the table where the price is between 50 and 100. The BETWEEN operator specifies the range of values that the price column must fall within in order to be included in the result set.

If the values are text, BETWEEN can filter within the alphabetical range.

**SELECT \***

**FROM table**

**WHERE name BETWEEN 'C' AND 'G';**

In this statement, the result would only include names that begin with the letter ‘C’ up to, but not including ones that begin with ‘G’. However, it would match a name set as ‘G’, but not ‘Gabriel’.

## [NOT] IN

The SQL IN operator is used to match a value against a list of values. The operator is often used in the WHERE clause of a SQL statement to test whether a column’s value is present in a set of specified values.

**NOT** Keywords used along with **IN**, negates it.

This SQL statement would retrieve all rows from the tv\_show table where the genre column has a value of either “Thriller” or “Comedy”:

**SELECT \***

**FROM SANDBOX.tv\_show**

**WHERE genre IN ('Thriller', 'Comedy');**

Here are some of its benefits:

* 1. Readability: Makes your SQL statements more readable and easier to understand, especially when you are comparing against a large list of values.
  2. Conciseness: Allows you to write shorter code than using multiple OR conditions.
  3. Flexibility: You can use it with different data types, including numeric, string, and date values, making it a versatile tool for filtering data in various situations.
  4. Improved performance: have better performance compared to using multiple OR conditions, as the database engine can process the values in the list more efficiently.

Overall, the IN operator provides a useful and flexible way to filter data in SQL, making your code more readable, concise, and performant.

## 

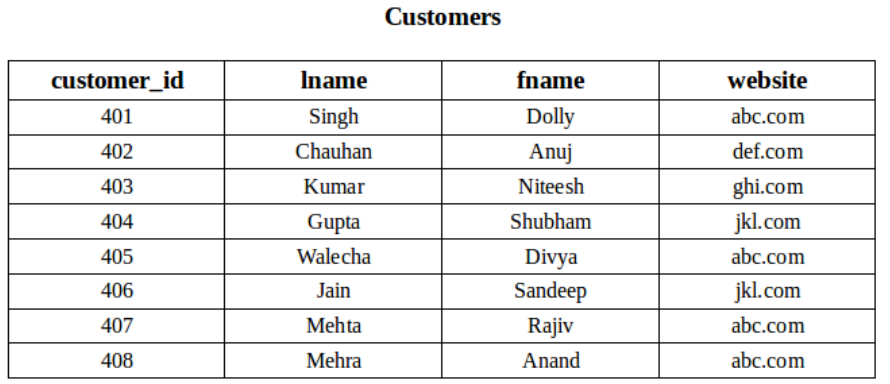
## [NOT] EXISTS

The **SQL EXISTS** condition is used to test whether a **correlated subquery** returns any results. If the subquery returns at least one row, the EXISTS condition evaluates to TRUE; otherwise, it evaluates to FALSE. The EXISTS operator can be used in various SQL statements like **SELECT**, **UPDATE**, **INSERT**, and **DELETE**to filter data based on whether certain conditions are met.

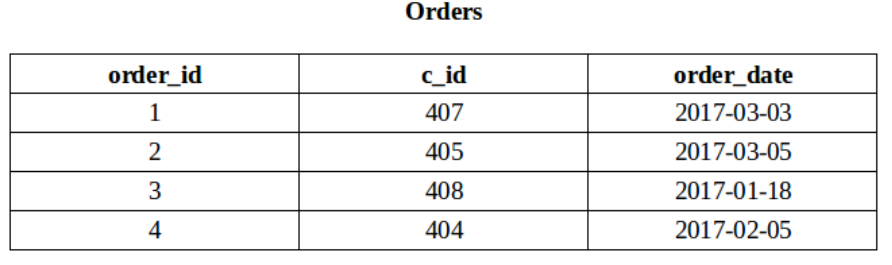
**Examples of SQL EXISTS**

Consider the following two relation "Customers" and "Orders".

**Customers Table**



**Orders Table:**



**Example 1 : Using EXISTS with SELECT**

To fetch the first and last name of the customers who placed atleast one order.

**Query:**

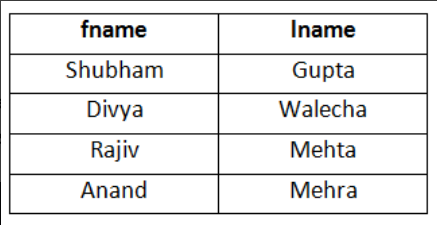
SELECT fname, lname

FROM Customers

WHERE EXISTS (SELECT \*

FROM Orders

WHERE Customers.customer\_id = Orders.c\_id);

**Output:**

**Example 2 : Using NOT with EXISTS**

Fetch last and first name of the customers who has not placed any order.

SELECT lname, fname

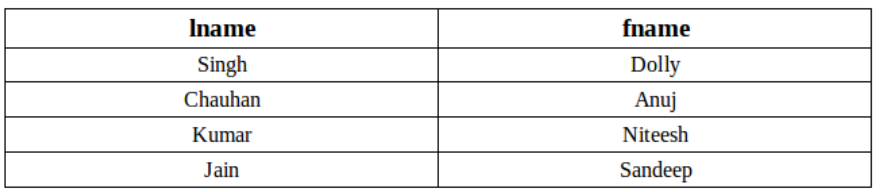
FROM Customers

WHERE NOT EXISTS (SELECT \*

FROM Orders

WHERE Customers.customer\_id = Orders.c\_id);

**Output:**



**Example 3 : Using EXISTS condition with DELETE statement**

Delete the record of all the customer from Order Table whose last name is 'Mehra'.

DELETE

FROM Orders

WHERE EXISTS (SELECT \*

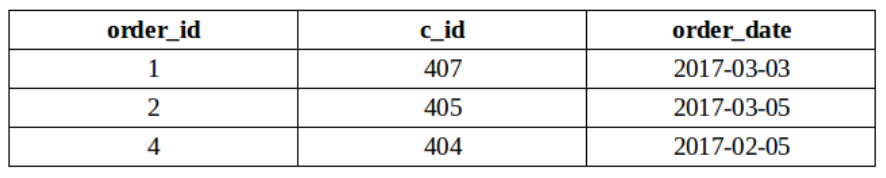
FROM customers

WHERE Customers.customer\_id = Orders.c\_id

AND Customers.lname = 'Mehra');

SELECT \* FROM Orders;

**Output:**



**Example 4 : Using EXISTS condition with UPDATE statement**

Update the lname as 'Kumari' of customer in Customer Table whose customer\_id is 401.

UPDATE Customers

SET lname = 'Kumari'

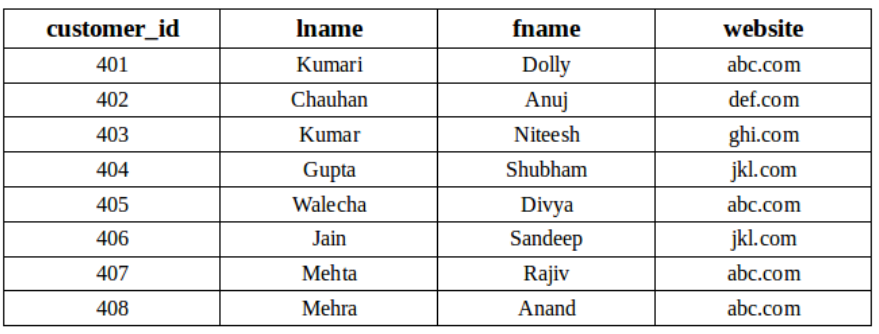
WHERE EXISTS (SELECT \*

FROM Customers

WHERE customer\_id = 401);

SELECT \* FROM Customers;

**Output:**



**When to Use SQL EXISTS**

The EXISTS condition is particularly useful in the following scenarios:

* **Checking Data Existence:** You can use **EXISTS** to check if related data exists in another table before performing an action (e.g., selecting, updating, deleting).
* **Performance:** EXISTS is often more efficient than using [IN](https://www.geeksforgeeks.org/sql/sql-between-in-operator/)when dealing with large **datasets**, as it stops searching once a match is found.
* **Correlated Subqueries:**EXISTS is ideal for correlated **subqueries**, where the subquery refers to the outer query’s values.

**Differences Between EXISTS and IN**

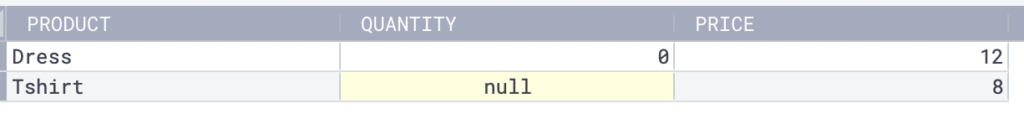
* EXISTS is used for checking the existence of rows, while **IN**checks if a value matches any value from a list or **subquery**result.
* EXISTS is more efficient when the subquery results in a large number of rows.
* IN works well for **small datasets** or static lists of values.

## IS [NOT] NULL

The SQL IS NULL is an operator that tests for NULL (or empty) values in a column. It is a common requirement to search for empty spaces in a database, as these values indicate missing or unknown data.

**NOT** is usually used with **IS NULL** Keyword to return non-null results.

A NULL value indicates that the data is missing or unknown, while a value of 0 indicates that the data is explicitly known and set to zero. If you are analyzing a stock table, the quantity of a product would be NULL when you don’t have the total units for that product. If it’s zero, it means it’s sold out.



Keep in mind that NULL values are treated differently than other values in SQL. When comparing two NULL values, they are considered unequal since NULL represents the absence of a value. This can sometimes lead to unexpected results, so it’s fundamental to know how your database management system handles them.

For example, the following SQL statement selects all rows where column1 is NULL:

**SELECT \***

**FROM table**

**WHERE column1 IS NULL;**

This information can be used to fill in missing data or to identify areas where data is lacking and needs to be collected.

In addition to the IS NULL operator, SQL also provides the IS NOT NULL operator, which is used to test for non-NULL values in a column.

**SELECT \***

**FROM table**

**WHERE column1 IS NOT NULL;**

As a result, you would see the rows that are not empty or do not have NULL values.

## LIKE

The SQL LIKE operator is used to search for a specific pattern in a string value. It is often used in the WHERE clause of a SELECT, UPDATE, or DELETE statement.

**NOT** used along LIKE returns all results except ones that match LIKE condition.

You can use two wildcard characters to help you define what you are looking for in a database. The percent sign (%) matches any number of characters, and the underscore (\_) corresponds to a single character.

The following SQL statement exemplifies how the LIKE operator returns the rows from the tv\_show table where the GENRE column starts with “Thriller”.

**SELECT \***

**FROM SANDBOX.tv\_show**

**WHERE GENRE LIKE 'Thriller%';**

To understand the difference, compare the result when we used “=” instead of LIKE.

SQL LIKE allows you to search for data that matches a specific pattern. It is handy when you only know partially the records you are searching because it helps you perform a flexible search that considers variations in the data.

Case sensitivity varies depending on the specific database platform and configuration. (Consult the documentation for case sensitivity behavior within your specific database platform. See notes below for LIKE case sensitivity within RedShift, BigQuery, and Snowflake.)

Searching for a specific string:

**SELECT \***

**FROM SANDBOX.tv\_show**

**WHERE SHOW\_NAME LIKE 'The Office';**

Note that without the ‘%’ wildcard character, LIKE is equivalent to using the equals operator.

Searching for a pattern:

**SELECT \***

**FROM SANDBOX.tv\_show**

**WHERE SHOW\_NAME LIKE 'The%';**

Searching for records that contain a specific word pattern:

**SELECT \***

**FROM SANDBOX.tv\_show**

**WHERE SHOW\_NAME LIKE '%Twil%';**

Searching for records that match multiple conditions:

**SELECT \***

**FROM SANDBOX.tv\_show**

**WHERE SHOW\_NAME LIKE 'Th%' AND GENRE LIKE 'Doc%';**

Note that this one helps you find results with typos. You can combine it with other operators and wildcard characters to perform more complex searches and find patterns in your data.

*Note: LIKE operator is case sensitive by default.*

*What is the difference between LIKE and ‘=’ Operator?*

* LIKE operator is mostly used for pattern matching while ‘=’ operator is used for exact matches.
* Use = wherever you can and LIKE wherever you must.
* ‘=’ operator is much faster compared to LIKE.
* LIKE can interpret wildcard\* characters, whereas ‘=’ operator cannot.

## \*Wildcards

A wildcard character is used to substitute one or more characters in a string. Wildcard characters are used with the LIKE operator. The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.

Mostly two wildcards are used in Oracle SQL.

‘**%**’ Wildcard and ‘**\_**’ Wildcard.

**Using the % Wildcard**

The % wildcard represents any number of characters, even zero characters.

Example

Return all customers that ends with the pattern ‘es’:

**SELECT \* FROM Customers  
WHERE CustomerName LIKE ‘%es’;**

Example

Return all customers that *contains* the pattern 'mer':

**SELECT \* FROM Customers  
WHERE CustomerName LIKE '%mer%';**

**Using the \_Wildcard**

The \_ wildcard represents a single character.

It can be any character or number, but each \_ represents one, and only one, character.

Example

Return all customers with a City starting with any character, followed by “ondon”:

**SELECT \* FROM Customers  
WHERE City LIKE ‘\_ondon’;**

Example

Return all customers with a City starting with "L", followed by any 3 characters, ending with "on":

**SELECT \* FROM Customers  
WHERE City LIKE 'L\_\_\_on';**

# ERRORS

**ORA-00942: table or view does not exist:** Error returned when the Table we are referring (through SELECT statement usually) is not created yet, or when there might be a typo in using the table name.

**ORA-00905: missing keyword:** Either a missing keyword or there is a typo mistake.

**ORA-00933: SQL command not properly ended:** When a Clause or Statement is partially ended.

**ORA-00920: invalid relational operator:** When a relational operator (<,>, =) is missing after a relational clause (WHERE).

**ORA-00970: missing WITH keyword:** Occurs when you miss WITH keyword in Command eg ‘GENERATED BY DEFAULT AS IDENTITY (START **WITH** 500 INCREMENT BY 1)’ when declaring a column as identity during table creation.