💻

SQL Notes by NEIL BAGCHI

🔸 Introduction to Basic Database Concepts

Data is nothing but facts and statistics stored or free-flowing over a network, generally, it's raw and unprocessed. Data becomes **information** when it is processed, turning it into something meaningful.

Collecting and storing data for analysis is a very human activity and we have been doing it for thousands of years. In modern times, we have come up with the term data analysis which is focused on part data discovery, data interpretation, and data communication. In order to effectively work with huge amounts of data in an organized and efficient manner, databases were invented.

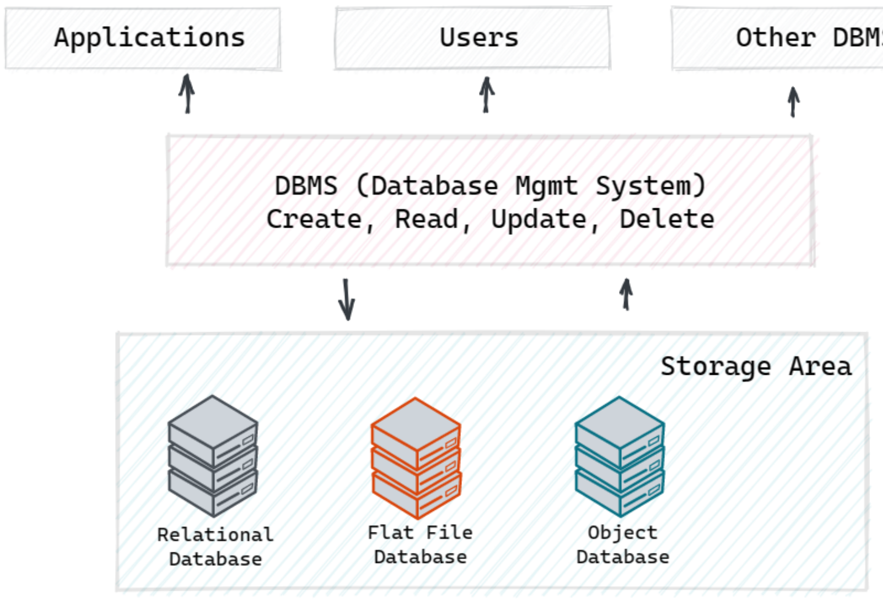
📌 *A Database is a collection of related data/information:*

- *organized in a way that data can be easily accessed, managed, and updated.*
- *captures information about an organization or organizational process*
- *supports the enterprise by continually updating the database*

Thus, a database can be software-based or hardware-based (ledgers, patient records, records in filing cabinets, etc.), with one sole purpose, storing and managing related data.

There is a range of database types—open source to proprietary, row-store to column-store. There are on-premises databases and cloud databases, as well as hybrid databases, where an organization runs the database software on a cloud vendor’s infrastructure. There are also a number of data stores that aren’t databases at all but can be queried with SQL.

📌 *A DBMS is a software system that allows the creation, definition, and manipulation of disk-resident databases, allowing users to store, process, and analyze data easily.*

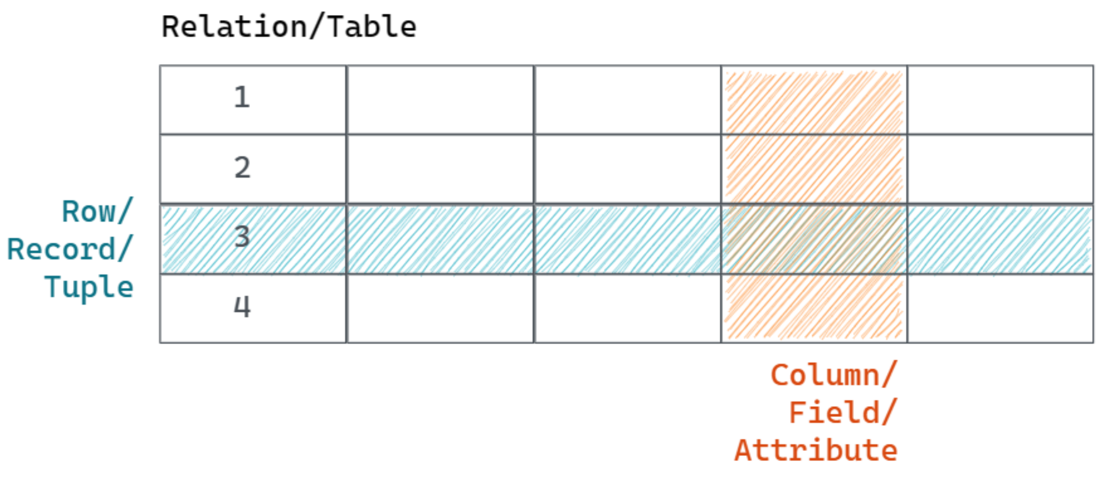


▼ RDBMS (Relational Database Management System)

📌 According to the relational model, data in a relational database is stored in ***relations***, **which are perceived by the user as tables**.

Each relation is composed of *tuples* (records or rows) and *attributes* (fields or columns).

Although tables in general can have duplicate rows of data but a true relation cannot have duplicate data.



A table consists of several records, each record can be broken down into several smaller parts known as **Attributes**. When an attribute is defined in a relation/table, it is defined to hold only certain type of value, also known as the **Attribute Domain**.

A **Relation Key** is an attribute that can uniquely identify each record/row in a relation/table. Every relation/table in an RDBMS should follow few constraints known as **Integrity Constraints** for it to be a valid relation.

These constraints are used to enforce certain rules or conditions that must be met before data can be stored in a database. The purpose of integrity constraints is to maintain the quality of data and prevent any errors or inconsistencies that may arise due to incorrect or inconsistent data entry.

📌 The three main Integrity Constraints are:

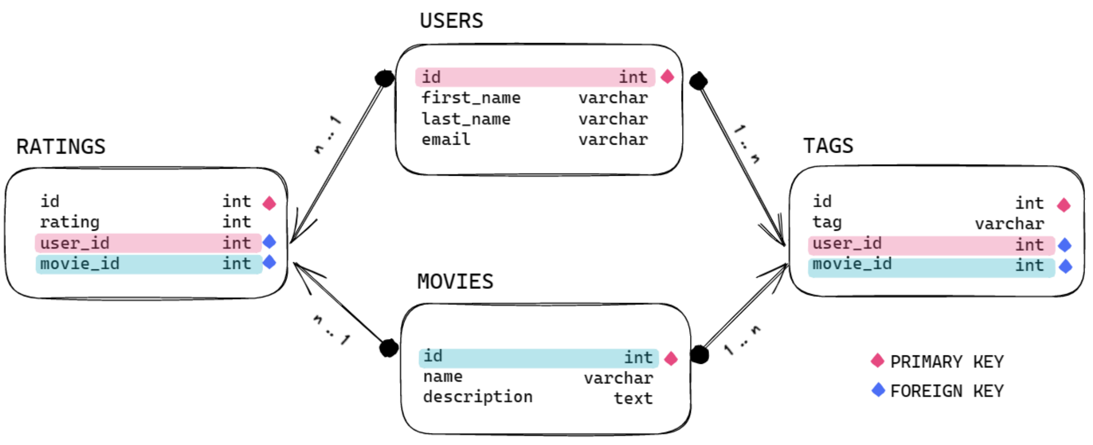
1. **Key Constraints:** specifies that there should be such an attribute (column) in a relation (table), which can be used to fetch data for any tuple (row). The Key attribute should never be **NULL** or same for two different rows of data e.g. PRIMARY KEY.
2. **Domain Constraints:** refer to the rules defined for the values that can be stored for a certain attribute e.g. NOT NULL, UNIQUE, etc.
3. **Referential integrity Constraints:** If a table references some data from another table, then data in that table should be present for the referential integrity constraint to hold true e.g. FOREIGN KEY.

ENTITY-RELATIONSHIP (ER) MODEL

If rows in a given table can be associated in some way with rows in another table, the tables are said to have a relationship between them. The manner in which the relationship is established depends on the type of relationship. Three types of relationships can exist between a pair of tables: **one-to-one (1:1), one-to-many (1:n), and many-to-many (n:n).**

📌 **Relationship diagrams** or **entity-relationship model** depict a high-level overview of how tables are related. **Primary Keys** are unique and used to identify each row of a table. Whereas **Foreign keys** are non-unique and can be used to link records/rows.

The main data objects are termed entities/columns, with their details defined as attributes. For example, a database column that stores employee names might have an attribute that specifies the maximum length of the name, or an attribute that indicates whether the column can contain null values. Some of these attributes are important and are used to identify the entity, and different entities are related using relationships.



What are the Types of Attributes?

1. **Simple attribute:** The attributes with values that are atomic and cannot be broken down further are simple attributes. For example, user's **age**.
2. **Composite attribute:** A composite attribute is made up of more than one simple attribute. For example, a user's **address** will contain, **house no.**, **street name**, **pin code,** etc.
3. **Derived attributes:** These are the attributes that are not present in the whole database but are derived using other attributes. For example, the **average age of users in a store**.

▼ KEYS

If an attribute can uniquely identify an entity, then it will be called a key. A Key can be a single attribute or a group of attributes, where the combination may act as a key. Following are the types of Keys:

student_id	name	phone	age
1	Akon	9876723452	17
2	Akon	9991165674	19
3	Bkon	7898756543	18
4	Ckon	8987867898	19
5	Dkon	9990080080	17

▼ Super Key

Super Key is defined as a set of attributes within a table that can uniquely identify each record within a table. In the table defined above super key would include `student_id`, `(student_id, name)`, `phone` etc.

Confused? The first one is pretty simple as `student_id` is unique for every row of data, hence it can be used to identify each row uniquely.

Next comes, `(student_id, name)`, now names of two students can be the same, but their `student_id` can't be the same hence this combination can also be a key.

Similarly, the phone number for every student will be unique, hence again, `phone` can also be a key. So they all are super keys.

▼ Candidate Key

Candidate keys are defined as the minimal set of fields that can uniquely identify each record in a table. It is an attribute or a set of attributes that can act as a Primary Key for a table to uniquely identify each record in that table. There can be more than one candidate key.

In our example, `student_id` and `phone` both are candidate keys for table **Student**.

- A candidate key can never be NULL or empty. And its value should be unique.
- There can be more than one candidate key for a table.
- A candidate key can be a combination of more than one column (attributes).

▼ Primary Key

A primary key is a candidate key that is most appropriate to become the main key for any table. It is a key that can uniquely identify each record in a table. It can be any column or a group of multiple columns, and there can only be one primary key in a table. The value cannot be repeated or null. It is a required field for concatenating data tables and improving the efficiency of searching data.

▼ Composite Key

Whenever a primary key consists of two or more attributes that uniquely identify any record in a table, it is called a **Composite key**. But the attributes which together form the **Composite key** are not a key independently or individually e.g. `(Student_id, Subject_id)`.

▼ Foreign Key

When it's determined that a pair of tables have a relationship with each other, we can typically establish the relationship by taking a copy of the primary key from the first table and inserting it into the second table, where it becomes a foreign key.

▼ MySQL

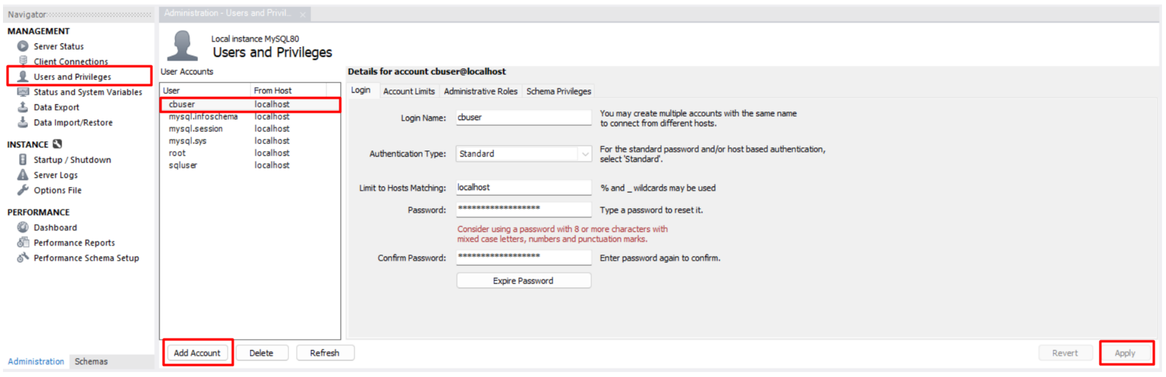
📌 The MySQL database system uses a **client-server** architecture where the **server** is the program that actually manipulates databases and **client programs** communicate an user's intent to the server by means of queries written in SQL.

The client program(s) are installed locally on the machine, but the server can be installed anywhere, as long as clients can connect to it. Thus, DBMS provides **concurrent use of the system** allowing efficient **data retrieval,** providing protection and security to the databases and maintaining data consistency in the case of multiple users.

Create New User in MySQL (2 ways)

```
#Following is typed in MySQL cmd line client
#root has the administrative privileges needed to set up other user accounts
mysql -u root -p  #-u username; -p password
Enter password: ********  #Enter the root password

#Creating a user and password, and Apply access privileges separately
mysql> CREATE USER 'sqluser'@'localhost' IDENTIFIED
       WITH mysql_native_password BY 'password' ;

mysql> GRANT ALL PRIVILEGES ON . TO 'sqluser'@'localhost' ;
mysql> FLUSH PRIVILEGES;

#Creating a user and password, and Provide access to specific database ONLY
mysql> GRANT ALL ON sakila.* TO 'cbuser'@'localhost' IDENTIFIED BY 'cbpass';

#If you plan to make connections to the server from another host, substitute that host in
```

```
  the GRANT statement.
mysql> GRANT ALL ON <database>.* TO 'cbuser'@'xyz.com' IDENTIFIED BY 'cbpass';
```



User	From Host
cbuser	localhost
mysql.infoschema	localhost
mysql.session	localhost
mysql.sys	localhost
root	localhost
sqluser	localhost

Login Name: cbuser — You may create multiple accounts with the same name to connect from different hosts.

Authentication Type: Standard — For the standard password and/or host based authentication, select 'Standard'.

Limit to Hosts Matching: localhost — % and _ wildcards may be used

Password: ****************** — Type a password to reset it.

Consider using a password with 8 or more characters with mixed case letters, numbers and punctuation marks.

Confirm Password: ****************** — Enter password again to confirm.

◆ Intro to SQL: A Non-Procedural Language

📌 A procedural language defines both the desired results and the mechanism, or process, by which the results are generated like Python, Java, etc. Nonprocedural languages also define the desired results, but the process by which the results are generated is left to an external agent.

With SQL, statements define the necessary inputs and outputs, but the manner in which a statement is executed is left to a component of the database engine known as the **optimizer**. The optimizer's job is to look at your SQL statements and, taking into account how your tables are configured and what indexes are available, decide the most efficient execution path (well, not always the *most* efficient). Most database engines will allow you to influence the optimizer's decisions by specifying *optimizer hints*, such as suggesting that a particular index be used.

To communicate with databases, SQL has four sublanguages for tackling different jobs, and these are mostly standard across database types

▼ DDL: Data Definition Language

This includes changes to the structure of the table like the creation of a table, altering the table, deleting a table, etc. All DDL commands are auto-committed. That means it saves all the changes permanently in the database.

▼ `create` command is used to create new objects

```
CREATE DATABASE <DB_NAME>; /*Create database*/

CREATE TABLE <TABLE_NAME> /*Create table*/
(
    column_name1 datatype1,
    column_name2 datatype2,
    column_name3 datatype3,
    column_name4 datatype4
);
```

▼ `alter` command is used for altering the table structure

- to add a column to an existing table

```
ALTER TABLE table_name ADD(
    column_name1 datatype1,
    column_name2 datatype2,
    /*Add a default value to a new column */
    column_name3 datatype3 DEFAULT some_value);
```

- to rename any existing column

```
ALTER TABLE table_name RENAME old_column_name TO new_column_name;
```

- to change the data type of any column or to modify its size

```
ALTER TABLE table_name MODIFY(column_name datatype);
```

- to drop a column from the table.

```
ALTER TABLE table_name DROP(column_name);
```

▼ truncate, drop, rename

The `TRUNCATE` command rem**oves all the records from a table**. But this command will not destroy the table's structure. When we use the `TRUNCATE` command on a table its (auto-increment) primary key is also initialized. Following is its syntax,

```
TRUNCATE TABLE table_name;
```

`DROP` command completely **removes a table from the database**. This command will also destroy the table structure and the data stored in it. Following is its syntax,

```
DROP TABLE table_name;
```

`RENAME` command is used to set a new name for any existing table. Following is the syntax,

```
RENAME TABLE old_table_name to new_table_name;
```

▼ DML: Data Manipulation Language

DML commands are used for manipulating the data stored in the table and not the table itself. DML commands are not auto-committed. It means changes are not permanent to the database, they can be rolled back.

▼ `insert` command is used to insert data into a table.

```
INSERT INTO student(id, name) values(102, 'Alex');
```

▼ `update` command is used to update any record of data in a table

```
/*Don't forget WHERE clause else all values will be updated*/
UPDATE table_name
SET column_name1 = new_value1,
    column_name2 = new_value2
WHERE some_condition;
```

▼ `delete` command is used to delete data from a table

```
DELETE FROM table_name;
```

📌 **Isn't DELETE the same as TRUNCATE ?**

The TRUNCATE command is different from the DELETE command. The delete command will delete all the rows from a table whereas the truncate command not only deletes all the records stored in the table, but it also re-initializes the table (like a newly created table).

For e.g.: If you have a table with 10 rows and an **auto_increment** primary key, and if you use the DELETE command to delete all the rows, it will delete all the rows, but will not re-initialize the primary key, hence if you will insert any row after using the DELETE command, the auto_increment primary key will start from 11. But in the case of the TRUNCATE command, the primary key is re-initialized, and it will again start from 1.

▼ DCL: Data Control Language

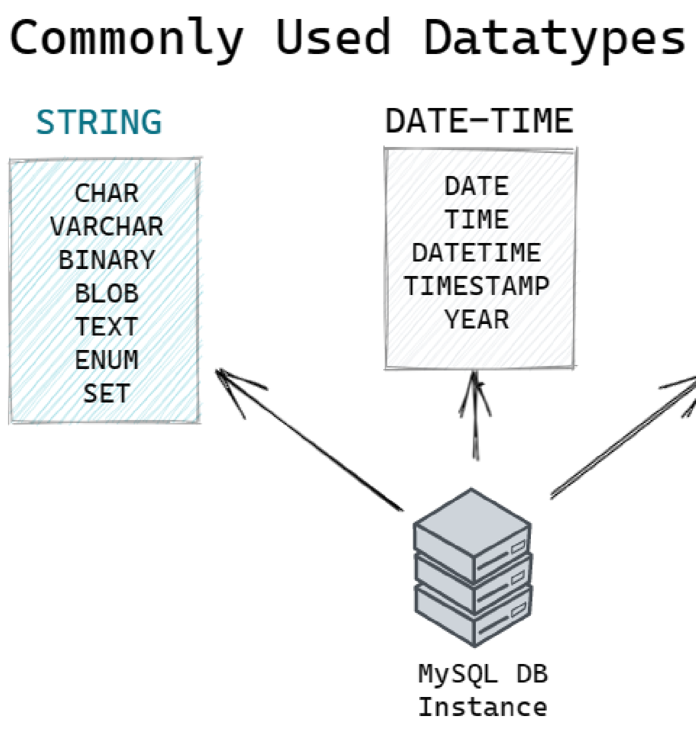
Data control language are the commands to grant and take back authority from any database user.

Command	Description
grant	grant permission of write
revoke	take back permission.

▼ DQL: Data Query Language (Primary Focus going ahead)

Data query language is used to fetch data from tables based on conditions that we can easily apply e.g: SELECT, FROM, WHERE, LIKE, ORDER BY, GROUP BY, HAVING, DISTINCT, AND/OR etc.

▼ Datatypes



▼ String Data Types

String data types are the most versatile. These can hold letters, numbers, and special
characters, including unprintable characters like tabs and newlines. String fields can be defined to hold a fixed or variable number of characters. When data is loaded, if strings arrive that are too big for the defined data type, they may be truncated or rejected entirely.

```
CHAR(size) -- Fixed length string which can contain letters, numbers and special chara
cters. The size parameter sets the maximum string length, from 0 – 255 with a default
 of 1.

VARCHAR(size) -- Variable length string similar to CHAR(), but with a maximum string l
ength range from 0 to 65535.

BINARY(size) -- Similar to CHAR() but stores binary byte strings.

VARBINARY(size) -- Similar to VARCHAR() but for binary byte strings.


/*Text/Blob: Holds longer strings that don’t fit in a VARCHAR. Descriptions or free te
xt entered by survey respondents might be held in these fields*/

TEXT(size) -- Holds a string with a maximum length of 65535 bytes. Again, better to us
e VARCHAR().

BLOB(size) -- Holds Binary Large Objects (BLOBs) with a max length of 65535 bytes.

TINYBLOB -- Holds Binary Large Objects (BLOBs) with a max length of 255 bytes.
```

```
TINYTEXT -- Holds a string with a maximum length of 255 characters. Use VARCHAR() inst
ead, as it’s fetched much faster.

MEDIUMTEXT -- Holds a string with a maximum length of 16,777,215 characters.

MEDIUMBLOB -- Holds Binary Large Objects (BLOBs) with a max length of 16,777,215 byte
s.

LONGTEXT -- Holds a string with a maximum length of 4,294,967,295 characters.

LONGBLOB -- Holds Binary Large Objects (BLOBs) with a max length of 4,294,967,295 byte
s.

ENUM(a, b, c, etc…) -- A string object that only has one value, which is chosen from a
list of values which you define, up to a maximum of 65535 values. If a value is added
 which isn’t on this list, it’s replaced with a blank value instead.

SET(a, b, c, etc…) -- A string object that can have 0 or more values, which is chosen
 from a list of values which you define, up to a maximum of 64 values.
```

CHAR vs VARCHAR

Value	CHAR(4)	Storage	VARCHAR(4)	Storage
‘ ‘	‘ ‘	4 bytes	‘ ‘	1 byte
‘ab’	‘ab ‘	4 bytes	‘ab’	3 byte
‘abcd’	‘abcd’	4 bytes	‘abcd’	5 byte

`CHAR` and `VARCHAR` are both character data types in SQL, but they have some key differences:

1. Storage and Length:

    - `CHAR(n)` has a fixed length, where `n` defines the number of characters. It always stores the same number of characters, even if the actual data is shorter. In this case, the data will be padded with spaces to fill the entire length.

    - `VARCHAR(n)` has a variable length, where `n` defines the maximum number of characters. It stores only the actual number of characters used, plus some additional overhead for storing the length information (usually 1 or 2 bytes).

2. Efficiency:

    - If you're dealing with strings that have a consistent length (e.g., country codes or fixed-format codes), `CHAR` can be more efficient because the storage size is fixed and the database doesn't need to spend extra effort in handling the variable length.

    - If you have strings with varying lengths (e.g., names, addresses, or descriptions), `VARCHAR` is more efficient because it only allocates storage for the actual data used, potentially saving storage space.

3. When to use one over the other:
    - Use `CHAR` when you have data with a consistent, known length, such as codes, abbreviations, or fixed-format data. This can lead to faster retrieval times and reduced storage overhead.
    - Use `VARCHAR` when you have data with varying lengths, or when you're unsure about the maximum length. This can help save storage space and better accommodate different data inputs.

▼ Numeric Data Types

Numeric data types are all the ones that store numbers, both positive and negative. Mathematical functions and operators can be applied to numeric fields. In some databases, such as Postgres, dividing integers results in an integer, rather than a value with decimal places.

```
INT(size) -- A medium integer with a signed range of -2147483648 to 2147483647, and an
unsigned range from 0 to 4294967295. Here, the size parameter specifies the maximum al
lowed display width, which is 255.

BIT(size) -- A bit-value type with a default of 1. The allowed number of bits in a val
ue is set via the size parameter, which can hold values from 1 to 64.

BOOL -- Essentially a quick way of setting the column to TINYINT with a size of 1. 0 i
s considered false, whilst 1 is considered true.

BOOLEAN -- Same as BOOL.

FLOAT(p) -- A floating point number value. If the precision (p) parameter is between 0
to 24, then the data type is set to FLOAT(), whilst if it's from 25 to 53, the data ty
pe is set to DOUBLE(). This behaviour is to make the storage of values more efficient.

DOUBLE(size, d) -- A floating point number value where the total digits are set by the
size parameter, and the number of digits after the decimal point is set by the d param
eter.

DECIMAL(size, d) -- An exact fixed point number where the total number of digits is se
t by the size parameters, and the total number of digits after the decimal point is se
t by the d parameter.

TINYINT(size) -- A very small integer with a signed range of -128 to 127, and an unsig
ned range of 0 to 255. Here, the size parameter specifies the maximum allowed display
 width, which is 255.

SMALLINT(size) -- A small integer with a signed range of -32768 to 32767, and an unsig
ned range from 0 to 65535. Here, the size parameter specifies the maximum allowed disp
lay width, which is 255.

MEDIUMINT(size) -- A medium integer with a signed range of -8388608 to 8388607, and an
unsigned range from 0 to 16777215. Here, the size parameter specifies the maximum allo
wed display width, which is 255.

INTEGER(size) -- Same as INT.
```

```
BIGINT(size) -- A medium integer with a signed range of -9223372036854775808 to 922337
2036854775807, and an unsigned range from 0 to 18446744073709551615. Here, the size pa
rameter specifies the maximum allowed display width, which is 255.

DEC(size, d) -- Same as DECIMAL.
```

FLOAT vs DOUBLE

`FLOAT` and `DOUBLE` are both floating-point data types in SQL, used to store approximate numeric values with decimal points. They differ in terms of precision, storage size, and range:

1. Precision, Storage Size, and Range:
   - `FLOAT`: Usually represented as a 32-bit single-precision floating-point number in most database systems, providing about 7 decimal digits of precision. The range for `FLOAT` is typically ±1.18 x 10^(-38) to ±3.4 x 10^(38).
   - `DOUBLE`: Usually represented as a 64-bit double-precision floating-point number, providing about 15 to 17 decimal digits of precision. The range for `DOUBLE` is typically ±2.23 x 10^(-308) to ±1.8 x 10^(308).
2. Efficiency:
   - `FLOAT` is more space-efficient, as it uses less storage compared to `DOUBLE`. It can also be faster in certain operations due to the reduced storage size and lower precision.
   - `DOUBLE` provides higher precision and a larger range, making it suitable for applications requiring greater accuracy or handling larger numeric values. However, it consumes more storage space and can be slower in certain operations due to the increased precision.
3. When to use one over the other:
   - Use `FLOAT` when you need to store approximate numeric values with moderate precision, and storage space or performance is a concern. It's suitable for most applications where slight inaccuracies are acceptable.
   - Use `DOUBLE` when you need to store approximate numeric values with higher precision or a larger range. It's suitable for scientific or financial applications where greater accuracy is required.

▼ Date/Time Data Types

```
DATE -- A simple date in YYYY-MM-DD format, with a supported range from ‘1000-01-01’ t
o ‘9999-12-31’.

DATETIME(fsp) -- A date time in YYYY-MM-DD hh:mm:ss format, with a supported range fro
m ‘1000-01-01 00:00:00’ to ‘9999-12-31 23:59:59’. By adding DEFAULT and ON UPDATE to t
```

```
he column definition, it automatically sets to the current date/time.

TIMESTAMP(fsp) -- A Unix Timestamp, which is a value relative to the number of seconds
since the Unix epoch (‘1970-01-01 00:00:00’ UTC). This has a supported range from ‘197
0-01-01 00:00:01’ UTC to ‘2038-01-09 03:14:07’ UTC.
-- By adding DEFAULT CURRENT_TIMESTAMP and ON UPDATE CURRENT TIMESTAMP to the column d
efinition, it automatically sets to current date/time.

TIME(fsp) -- A time in hh:mm:ss format, with a supported range from ‘-838:59:59’ to ‘8
38:59:59’.

YEAR -- A year, with a supported range of ‘1901’ to ‘2155’.
```

DATETIME VS TIMESTAMP

`DATETIME` and `TIMESTAMP` are both data types in SQL used to store date and time values. They have some differences in terms of storage format, range, and time zone handling:

1. Storage Format and Range:
    - `DATETIME`: Stores date and time values as a combination of the date and time, usually without time zone information. The typical range for `DATETIME` in most database systems is from '1000-01-01 00:00:00' to '9999-12-31 23:59:59'.
    - `TIMESTAMP`: Stores date and time values as a single value, representing the number of seconds elapsed since the Unix epoch ('1970-01-01 00:00:00 UTC'). The typical range for `TIMESTAMP` in most database systems is from '1970-01-01 00:00:01' UTC to '2038-01-19 03:14:07' UTC.
2. Time Zone Handling:
    - `DATETIME`: Does not store time zone information and is not affected by time zone settings. It represents the same point in time regardless of the time zone.
    - `TIMESTAMP`: Stores date and time values in UTC, but it can be converted to the local time zone when retrieved, based on the database or session time zone settings.
3. Efficiency:
    - `DATETIME`: Can be less efficient in terms of storage, as it stores the date and time separately. However, it can be easier to work with when time zone conversion is not required.
    - `TIMESTAMP`: Usually more storage-efficient, as it stores the date and time as a single value. It can also be more efficient when dealing with time zone conversions, as the conversion is automatically handled by the database system.

4. When to use one over the other:

    - Use `DATETIME` when you need to store date and time values without considering time zones, or when you want to store dates outside the `TIMESTAMP` range. This is useful for applications where time zone conversion is not necessary or where you need to represent historical or future dates beyond the `TIMESTAMP` range.
    - Use `TIMESTAMP` when you need to store date and time values with time zone awareness, or when you require efficient storage and automatic time zone conversion. This is suitable for applications where time zone conversion is important, such as when dealing with international dates and times.

▼ NULL (Special)

In SQL, a **Null** represents a **missing** or **unknown** value; a Null **does not** represent a zero, a character string of one or more blank spaces, or a “zero-length” character string.

The major drawback of Nulls is their adverse effect on mathematical operations. Any operation involving a Null evaluates to Null. This is logically reasonable—if a number is unknown, then the result of the operation is necessarily unknown.

(25 * 3) + 4 = 79

(Null * 3) + 4 = Null

(25 * Null) + 4 = Null

(25 * 3) + Null = Null

▼ Constraints

▼ NOT NULL (Domain Integrity)

The specified entity can’t be left blank/unknown

```
CREATE TABLE contacts (
  name VARCHAR(100) NOT NULL,
);
```

▼ UNIQUE (Domain Integrity)

The specified entity can’t have duplicates

```
CREATE TABLE contacts (
  name VARCHAR(100) NOT NULL,
  phone VARCHAR(15) NOT NULL UNIQUE
);
```

▼ DEFAULT (Domain Integrity)

The specified entity will take the default value if it was left blank while creation.

```
CREATE TABLE Customer (
Custid INT,
CustName VARCHAR(100),
CustAddress VARCHAR(150),
Phno VARCHAR(15) DEFAULT '9999999'
)
```

▼ CHECK & NAMED CONSTRAINTS (Domain Integrity)

A specific logic defined at the time of creation will be checked and entries will only be allowed when this is satisfied.

```
CREATE TABLE users (
  username VARCHAR(20) NOT NULL,
  age INT CHECK (age > 0)
);

--A better way is to name the constraint so that it provides usedful information
--when code raises exception
CREATE TABLE users2 (
    username VARCHAR(20) NOT NULL,
    age INT,
    CONSTRAINT age_not_negative CHECK (age >= 0)
);
```

▼ Primary Key (Entity Integrity)

We will talk about it in the next section

▼ Foreign Key (Referential Integrity)

We will talk about it in the next section

MULTIPLE COLUMN CONSTRAINTS

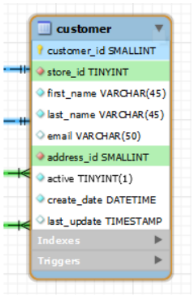
```
--Checks if the combination of name and address is unique
CREATE TABLE companies (
    name VARCHAR(255) NOT NULL,
    address VARCHAR(255) NOT NULL,
    CONSTRAINT name_address UNIQUE (name , address)
);

-- Check if sale price column is greater than purchase price
CREATE TABLE houses (
  purchase_price INT NOT NULL,
  sale_price INT NOT NULL,
  CONSTRAINT sprice_gt_pprice CHECK(sale_price >= purchase_price)
);
```

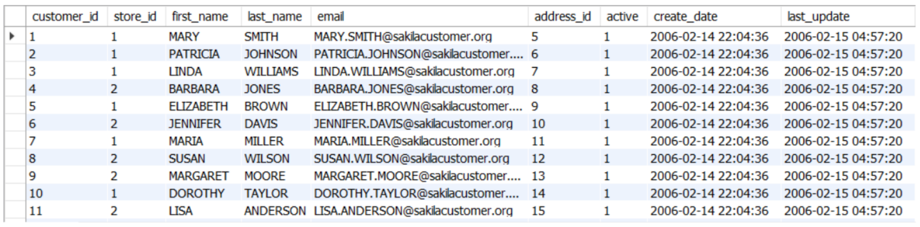
▼ CRUD - Create, Read, Update, Delete

```
/*To view existing databases*/
SHOW DATABASES;

/*To create a database*/
CREATE DATABASE <database_name>;

/*To use a database*/
USE <database_name>;

/*CAREFUL!! Once a dataabse is dropped, it gone forever */
DROP DATABASE <database_name>;

/*Create Tables*/
CREATE TABLE <table_name>
(<column_name1> <data_type1>, /*datatype - what kind of data will populate the col*/
 <column_name2> <data_type2>);


/*Default behavior is to allow NULLs if not specified in a table. Let's define a table wit
h NOT NULL constraint*/
CREATE TABLE <table_name> (
    id INT NOT NULL AUTO_INCREMENT,
    <column_name1> VARCHAR(100) NOT NULL DEFAULT 'Unnamed',
    <column_name2> INT NOT NULL,
    PRIMARY KEY (id) );

*/*NOTE - use single quotes always. Use \ in order to escape a character*/*

/*To view existing tables*/
SHOW TABLES;

/*CAREFUL!! Once a table is dropped, it is gone forever */
DROP TABLE <table_name>;

/*Viewing Table structure*/
DESC <table_name>;

/*Insert data into a table*/
INSERT INTO <table_name> (<column_name1>, <column_name2>)
VALUES (<value1>, <value2>);

/*Update table data*/
UPDATE <table_name>
SET <column_name1> = <value1>
WHERE <column_name2> = <value2>;

/*Delete data from table*/
DELETE FROM <table_name> WHERE <column_name1>=<value1>;
```

A database can contain **many related tables** interlinked amongst each other through relationships. The following shows a table called ‘customer’ that contains 9 columns (fields) and multiple rows (records). Each row corresponds to one specific customer whereas each column contains an **attribute** related to customer details.

```
/*Example*/
CREATE TABLE employee
(
    employee_id     INT PRIMARY KEY AUTO_INCREMENT,
    first_name      VARCHAR(255) NOT NULL,
    last_name       VARCHAR(255) NOT NULL,
    address_id      SMALLINT FOREIGN KEY REFERENCES address(id),
    middle_name     VARCHAR(255),
    age             INT NOT NULL,
    pay_date        DATE,
    amount          DECIMAL(8,2),
    current_status  VARCHAR(255) NOT NULL DEFAULT 'employed'
);
```





customer_id	store_id	first_name	last_name	email	address_id	active	create_date	last_update
1	1	MARY	SMITH	MARY.SMITH@sakilacustomer.org	5	1	2006-02-14 22:04:36	2006-02-15 04:57:20
2	1	PATRICIA	JOHNSON	PATRICIA.JOHNSON@sakilacustomer....	6	1	2006-02-14 22:04:36	2006-02-15 04:57:20
3	1	LINDA	WILLIAMS	LINDA.WILLIAMS@sakilacustomer.org	7	1	2006-02-14 22:04:36	2006-02-15 04:57:20
4	2	BARBARA	JONES	BARBARA.JONES@sakilacustomer.org	8	1	2006-02-14 22:04:36	2006-02-15 04:57:20
5	1	ELIZABETH	BROWN	ELIZABETH.BROWN@sakilacustomer....	9	1	2006-02-14 22:04:36	2006-02-15 04:57:20
6	2	JENNIFER	DAVIS	JENNIFER.DAVIS@sakilacustomer.org	10	1	2006-02-14 22:04:36	2006-02-15 04:57:20
7	1	MARIA	MILLER	MARIA.MILLER@sakilacustomer.org	11	1	2006-02-14 22:04:36	2006-02-15 04:57:20
8	2	SUSAN	WILSON	SUSAN.WILSON@sakilacustomer.org	12	1	2006-02-14 22:04:36	2006-02-15 04:57:20
9	2	MARGARET	MOORE	MARGARET.MOORE@sakilacustomer....	13	1	2006-02-14 22:04:36	2006-02-15 04:57:20
10	1	DOROTHY	TAYLOR	DOROTHY.TAYLOR@sakilacustomer....	14	1	2006-02-14 22:04:36	2006-02-15 04:57:20
11	2	LISA	ANDERSON	LISA.ANDERSON@sakilacustomer.org	15	1	2006-02-14 22:04:36	2006-02-15 04:57:20

▼ ALTERing existing TABLE

```
--Add Column to existing table
ALTER TABLE companies
ADD COLUMN phone VARCHAR(15);

ALTER TABLE companies
ADD COLUMN employee_count INT NOT NULL DEFAULT 1;


--Drop Column from existing table
ALTER TABLE companies DROP COLUMN phone;


--Renaming Columns in existing table
ALTER TABLE suppliers RENAME TO companies;

ALTER TABLE companies
RENAME COLUMN name TO company_name;


--Modifying Column in existing table
ALTER TABLE companies
MODIFY company_name VARCHAR(100) DEFAULT 'unknown';

ALTER TABLE suppliers
CHANGE business biz_name VARCHAR(50);


--Adding and Dropping Constraint
ALTER TABLE houses
```

```
ADD CONSTRAINT positive_pprice CHECK (purchase_price >= 0);

ALTER TABLE houses DROP CONSTRAINT positive_pprice;
```

🔶 Querying in SQL

▼ SELECT…FROM

used in conjunction with other keywords and clauses to find and view the information in an almost limitless number of ways.

A **SELECT** statement is composed of several distinct keywords, known as *clauses*. You define a SELECT statement by using various configurations of these clauses to retrieve the information you require. Some of these clauses are required, although others are optional.

FROM is the second most important clause in the SELECT statement and is also required. You use the FROM clause to specify the **tables** from which to draw the columns you've listed in the SELECT clause

```
SELECT <DISTINCT> <column_names>
                  AS <some_name>
FROM <table_name>
WHERE <logical_conditions> --Optional
GROUP BY <column_names> --Optional
HAVING <logical_conditions> --Optional
ORDER BY <column_names>; --Optional
```

📌 A "**table**" can be a database table, a view (a type of saved query that otherwise functions like a table), or a subquery.

When you execute a SELECT statement, it usually retrieves one or more rows of information—the exact number depends on how you construct the statement. These rows are collectively known as a ***result set.***

Use the **DISTINCT** keyword in your SELECT statement, and the result set will be free and clear of all duplicate rows. DISTINCT is an optional keyword that evaluates the values of all the columns *as a single unit* on a row-by-row basis and eliminates any redundant rows it finds.

SQL **aliases** are used to give a table, or a column in a table, a temporary name. Aliases are often used to make column names more readable. An alias only exists for the duration of that query. An alias is created with the `AS` keyword.

```
--Alias for Columns
SELECT CustomerID AS ID
     , CustomerName AS Customer
FROM Customers;

--Alias for Tables
SELECT o.OrderID
      ,o.OrderDate
      ,c.CustomerName
FROM Customers AS c
    ,Orders AS o
WHERE c.CustomerName = 'Around the Horn'
  AND c.CustomerID = o.CustomerID;
```

▼ WHERE

clause can filter records/rows based on logical operators. The WHERE clause contains a ***search condition*** that it uses as the filter. This search condition provides the mechanism needed to select only the rows you need or exclude the ones you don't want. Your database system applies the search condition to each row in the logical table defined by the FROM clause.

Operator	Condition	SQL Example
=, !=, < <=, >, >=	Standard numerical/string/date operators = equal to <>, != not equal to < less than > greater than <= less than or equal to >= greater than or equal to	`col_name != 4` `col_name = "abc"` `payment_date>'2006-01-01'` -- yyyy-mm-dd
BETWEEN … AND …	The number/string/date is within the range of two values (inclusive)	`col_name BETWEEN 1.5 AND 10.5`
NOT BETWEEN … AND …	The number/string/date is not within the range of two values (inclusive)	`col_name NOT BETWEEN 1 AND 10`
IN (…)	The number/string exists in a list	`col_name IN (2, 4, 6)` `col_name IN ("A", "B", "C")`
NOT IN (…)	The number/string does not exist in a list	`col_name NOT IN (1, 3, 5)` `col_name NOT IN ("D", "E", "F")`
LIKE	Case-insensitive exact string comparison	`col_name LIKE "ABC"`
NOT LIKE	Case insensitive exact string inequality comparison	`col_name NOT LIKE "ABCD"`
%	Used anywhere in a string to match a **sequence of zero or more characters** (only with LIKE or NOT LIKE)	`col_name LIKE "%AT%"` (matches "AT", "ATTIC", "CAT" or even "BATS")

`_`	Used anywhere in a string to match a **single character** (only with LIKE or NOT LIKE)	`col_name LIKE "AN_"` (matches "AND", but not "AN")

`'Sha%'`	Character string can be any length but must begin with "Sha"	Shannon, Sharon, Shawn
`'%son'`	Character string can be any length but must end with "son"	Benson, Johnson, Morrison
`'%han%'`	Character string can be any length but must contain "han"	Buchanan, handel, Johansen, Nathanson
`'Ro_'`	Character string can be only three characters in length and must have "Ro" as the first and second letters	Rob, Ron, Roy
`'_im'`	Character string can be only three characters in length and must have "im" as the second and third letters	Jim, Kim, Tim
`'_ar_'`	Character string can be only four characters in length and must have "ar" as the second and third letters	Bart, Gary, Mark
`'_at%'’`	Character string can be any length but must have "at" as the second and third letters	Gates, Matthews, Patterson
`'%ac_'`	Character string can be any length but must have "ac" as the second and third letters from the end of the string	Apodaca, Tracy, Wallace

You can combine two or more conditions by using the **AND** and **OR** operators, and the complete set of conditions you've combined to answer a given request constitutes a single search condition.

```
SELECT
    DISTINCT customer_id,
    rental_id,
    amount,
    payment_date
FROM payment
WHERE customer_id IN (42,53,60,75)
        OR amount>5;

SELECT
    customer_id,
    rental_id,
    amount,
    payment_date
FROM payment
WHERE customer_id<101
      AND amount>5
      AND payment_date>'2006-01-01'; -- yyyy-mm-dd
```

```
SELECT * FROM people WHERE HOUR(birthtime)
BETWEEN 12 AND 16;
```

📌 **Note**: Unlike specifying multiple columns in our SELECT statement, we cannot list multiple tables in our FROM statement. In order to use multiple tables, we'll need to use a JOIN

The **ESCAPE(\)** option allows you to designate a *single* character string literal—known as an *escape character*—to indicate how the database system should interpret a percent sign or underscore character within a pattern string. Place the escape character after the ESCAPE keyword and enclose it within single quotes, as you would any character string literal. When the escape character precedes a wildcard character in a pattern string, the database system interprets that wildcard character *literally* within the pattern string.

```
-- To select books with '%' in their title:
SELECT * FROM books
WHERE title LIKE '%\%%';

-- To select books with an underscore '_' in the title:
SELECT * FROM books
WHERE title LIKE '%G\_00_%';
```

To retrieve **Null** values from a value expression, we use the *Null* condition.

Note: A condition such as `<ValueExpression> = Null` is invalid because the value of the value expression cannot be compared to something that is, by definition, unknown.

```
SELECT CONCAT(CustFirstName, ' ', CustLastName) AS Customer
FROM Customers
WHERE CustCounty IS NULL;
```

📌 **Note:** Virtually all commercial database systems include a query optimizer that looks at your entire request and tries to figure out the fastest way to return the answer. The indexes that your database administrator has defined on columns in your tables have the biggest influence on what most optimizers choose to do. But it doesn't hurt to make it a practice to include the most exclusive search condition first to further influence your database system's optimizer.

▼ GROUP BY

used to specify how to group the data in our results and comes before HAVING and ORDER BY clauses in our query. It works similarly to Pivot Tables in Excel.

📌 Think 'dimensions' vs 'metrics'. **Make sure all the dimensions are in our GROUP BY.**

*'**dimensions**' are the column headings*

*'**metrics**' are the aggregate of the columns for e.g. COUNT(__)*

Multiple columns can be specified at once using GROUP BY to create sub-groups in our result set (similar to multiple row or column labels in a PivotTable!!).

TIP: Group by customer segment and add a time dimension to create cohort trends.

```
SELECT
    replacement_cost,
    COUNT DISTINCT(film_id) AS Number_of_films,
    AVG(rental_rate) AS Avg_rental_rate,
    MIN(rental_rate) AS Min_rental_rate,
    MAX(rental_rate) AS Max_rental_rate
FROM film
GROUP BY replacement_cost;
```

WITH ROLLUP

the modifier used in SQL's `GROUP BY` clause to generate summary and subtotal rows for grouped data, providing a quick way to analyze hierarchical or aggregated data. It is particularly useful when you want to calculate subtotals and grand totals along with the grouped data.

```
SELECT region, product, SUM(sales_amount) as total_sales
FROM sales_data
GROUP BY region, product;
```

```
+--------+---------+-------------------+
| region | product | SUM(sales_amount) |
+--------+---------+-------------------+
| EAST    | Mobile |              42   |
| EAST    | IPads  |              50   |
| WEST    | Mobile |              60   |
+---------+--------+-------------------+

SELECT region, product, SUM(sales_amount) as total_sales
FROM sales_data
GROUP BY region, product WITH ROLLUP;

+--------+---------+-------------------+
| region | product | SUM(sales_amount) |
+--------+---------+-------------------+
| EAST    | Mobile |              42   |
| EAST    | IPads  |              50   |
| EAST    | NULL   |              46   |
| WEST    | Mobile |              60   |
| WEST    | NULL   |              60   |
| NULL    | NULL   |           50.66   |
+---------+--------+-------------------+
```

▼ HAVING

can only be used with GROUP BY!! HAVING lets us specify the filtering logic the we want to apply to our group-level aggregated **metrics**.

```
SELECT
    customer_id,
    COUNT(rental_id) AS total_rentals
FROM rental
GROUP BY customer_id
HAVING COUNT(rental_id)<15;
```

▼ ORDER BY

specifies the order (ascending/descending) in which query results are displayed.

```
SELECT
    title,
    length,
    rental_rate
FROM film
ORDER BY length DESC, title ASC;
```

Is any importance placed on the sequence of the columns in the ORDER BY clause? The answer is *"**Yes**!"* The sequence is important because your database system will evaluate the columns in the ORDER BY clause from left to right. Also, the importance of the sequence grows in direct proportion to the number of columns you use.

The manner in which the ORDER BY clause sorts the information depends on the collating sequence used by your database software. The collating sequence determines the order of precedence for every character listed in the current language character set specified by your operating system. For example, it identifies whether lowercase letters will be sorted before uppercase letters, or whether a case will even matter.

▼ LIMIT

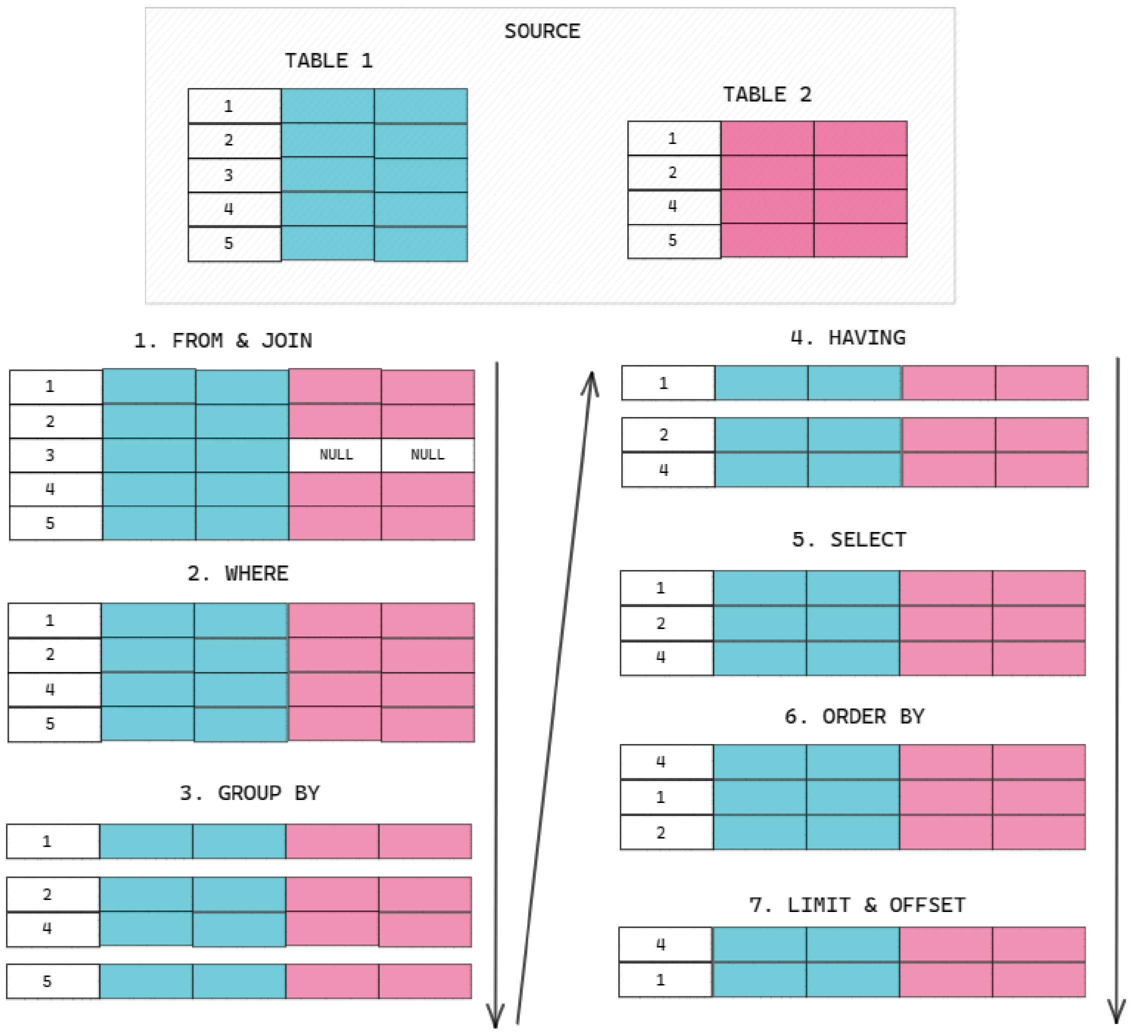
`LIMIT` *`n`* tells the server to return the first *`n`* rows of a result set. `LIMIT` also has a two-argument form that allows you to pick out any arbitrary section of rows from a result. The arguments indicate how many rows to skip and how many to return. This means that you can use `LIMIT` to do such things as skip two rows and return the next, thus answering questions such as "what is the *third*-smallest or *third*-largest value?

```
SELECT title, pages
FROM books
ORDER BY pages DESC LIMIT 1;

#Third smallest birth entry
SELECT * FROM profile ORDER BY birth LIMIT 2,1;

+----+------+------------+-------+--------------+------+
| id | name | birth      | color | foods        | cats |
+----+------+------------+-------+--------------+------+
| 10 | Tony | 1960-05-01 | white | burrito,pizza |    0 |
+----+------+------------+-------+--------------+------+

#Third largest birth entry
SELECT * FROM profile ORDER BY birth DESC LIMIT 2,1;
+----+------+------------+-------+---------------------+------+
| id | name | birth      | color | foods               | cats |
+----+------+------------+-------+---------------------+------+
|  1 | Fred | 1970-04-13 | black | lutefisk,fadge,pizza |    0 |
+----+------+------------+-------+---------------------+------+
```

Another alternative to limit the amount of data is to use Sampling technique. Sampling can be accomplished by using a function on an ID field that has a random distribution of digits at the beginning or end.

```
--If the ID field is an integer, mod can be used to find the last one, two, or more digits
and filter on the result:
WHERE mod(integer_order_id,100) = 6

--If the field is alphanumeric, you can use a right() function to find a certain number of
digits at the end
WHERE right(alphanum_order_id,1) = 'B'
```

Order of Execution



▼ Pivot Tables (COUNT..CASE)

allows us to process a series of IF/THEN logical operations in a specific order. When you need to test the data in one column to determine how to handle data in another column, you need to be able to say something like "If the value in column 'a' is 'x', then return expression 'y', else return expression 'z'." The SQL Standard provides a handy syntax to accomplish this: CASE

📌 Every CASE statement begins with **CASE**, ends with **END,** and contains at least one **WHEN/THEN** pair.

```
SELECT
  CONCAT(first_name,' ',last_name) AS name_customer,
  CASE
        WHEN store_id=1 AND active=1 THEN 'store 1 active'
        WHEN store_id=1 AND active=0 THEN 'store 1 inactive'
        WHEN store_id=2 AND active=1 THEN 'store 2 active'
        WHEN store_id=2 AND active=0 THEN 'store 2 inactive'
        ELSE 'check logic'
  END AS store_status
FROM customer;


SELECT StaffID, StfFirstName, StfLastname,
YEAR(CAST('2017-10-01' As Date))- YEAR(DateHired)
- CASE
    WHEN Month(DateHired) < 10 THEN 0
    WHEN Month(DateHired) > 10 THEN 1
    WHEN Day(DateHired) > 1 THEN 1
    ELSE 0
  END) AS LengthOfService
FROM Staff
ORDER BY StfLastName, StfFirstName;
```

We can use COUNT and CASE to replicate Excel’s ability to pivot to columns

```
SELECT
  store_id,
  COUNT(CASE WHEN active=1 THEN customer_id ELSE NULL END) AS 'active customer',
  COUNT(CASE WHEN active=0 THEN customer_id ELSE NULL END) AS 'inactive customer'
FROM customer
GROUP BY store_id;
```

▼ Unpivoting using UNION

Sometimes we have the opposite problem and need to move data stored in columns into rows instead to create tidy data. This operation is called unpivoting. Data sets that may need unpivoting are those that are in a pivot table format.

order date	shirts amount	shoes amount	hats amount
2020-05-01	5268.56	1211.65	562.25
2020-05-02	5533.84	522.25	325.62
2020-05-03	5986.85	1088.62	858.35

UNION removes duplicates from the result set, whereas UNION ALL retains all records, whether duplicates or not. UNION ALL is faster since the database doesn’t have to go over the data to find duplicates. It also ensures that every record ends up in the result set.

```
SELECT order_date
,CAST('2020-05-01' as date) as order_date
,shirts_amount as shirts
,shoes_amount as shoes
,hats_amount as hats
FROM orders
```

```
 UNION ALL

SELECT order_date
,CAST('2020-05-02' as date) as order_date
,shirts_amount as shirts
,shoes_amount as shoes
,hats_amount as hats
FROM orders
...
```

UNION combines tables by stacking new data from a different table below the existing one. UNION deduplicates records and keeps only distinct values in the result set. **UNION ALL** is used if we wish to keep all the records including duplicate records.

```
SELECT c.first_name, c.last_name
    -> FROM customer c
    -> WHERE c.first_name LIKE 'J%' AND c.last_name LIKE 'D%'
    -> UNION
    -> SELECT a.first_name, a.last_name
    -> FROM actor a
    -> WHERE a.first_name LIKE 'J%' AND a.last_name LIKE 'D%';
+------------+-----------+
| first_name | last_name |
+------------+-----------+
| JENNIFER   | DAVIS     |
| JUDY       | DEAN      |
| JODIE      | DEGENERES |
| JULIANNE   | DENCH     |
+------------+-----------+

SELECT c.first_name, c.last_name
    -> FROM customer c
    -> WHERE c.first_name LIKE 'J%' AND c.last_name LIKE 'D%'
    -> UNION ALL
    -> SELECT a.first_name, a.last_name
    -> FROM actor a
    -> WHERE a.first_name LIKE 'J%' AND a.last_name LIKE 'D%';
+------------+-----------+
| first_name | last_name |
+------------+-----------+
| JENNIFER   | DAVIS     |
| JENNIFER   | DAVIS     |
| JUDY       | DEAN      |
| JODIE      | DEGENERES |
| JULIANNE   | DENCH     |
+------------+-----------+
```

▼ Tips and Tricks

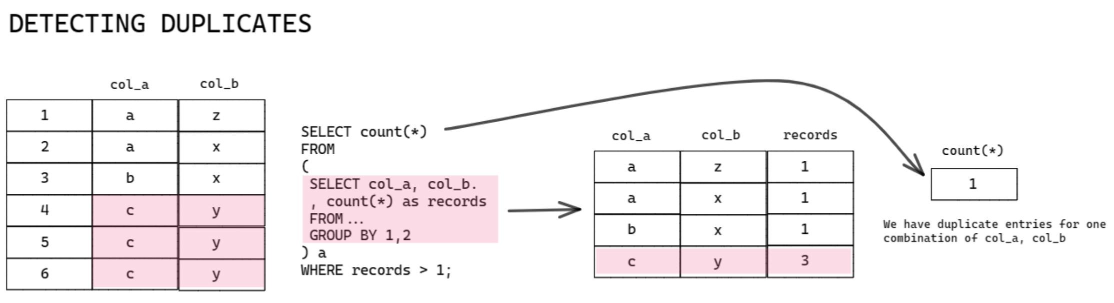
Mapping NULL to Other Values

```
SELECT name,
      IF(id IS NULL,'Unknown', id) AS 'id'
FROM taxpayer;
```

```
+---------+---------+
| name    | id      |
+---------+---------+
| bernina | 198-48  |
| bertha  | Unknown |
| ben     | Unknown |
| bill    | 475-83  |
+---------+---------+
```

Detecting Duplicates

```
SELECT count(*)
FROM
(
 SELECT column_a, column_b, column_c...
 , count(*) as records
 FROM...
 GROUP BY 1,2,3...
) a
WHERE records > 1;
```



	col_a	col_b
1	a	z
2	a	x
3	b	x
4	c	y
5	c	y
6	c	y

```
SELECT count(*)
FROM
(
 SELECT col_a, col_b.
 , count(*) as records
 FROM ...
 GROUP BY 1,2
) a
WHERE records > 1;
```

col_a	col_b	records
a	z	1
a	x	1
b	x	1
c	y	3

count(*)
1

We have duplicate entries for one combination of col_a, col_b

```
--Alternate way of finding duplicates
SELECT column_a, column_b, column_c...
 , count(*) as records
 FROM...
 GROUP BY 1,2,3...
HAVING count(*)>1;
```

CAST to avoid datatype mismatch

You must be careful when you create an expression to make sure that the data types of the columns and literals are compatible with the operation you are requesting. The CAST function converts a literal value or the value of a column into a specific data type. This helps to ensure that the data types of the values in the expression are *compatible.*

```
CASE WHEN order_items <= 3 THEN CAST(order_items as varchar(4))
 ELSE '4+'
 END
```

▼ NUMERIC FUNCTIONS

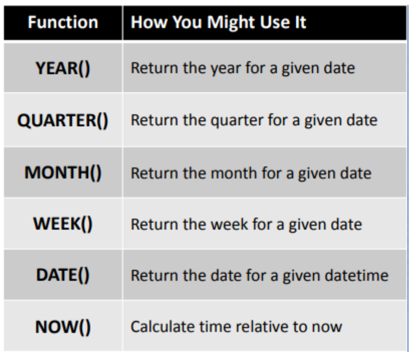
```
ABS -- Returns the absolute value of the given number.

AVG -- Returns the average value of the given expression.

CEIL -- Returns the closest whole number (integer) upwards from a given decimal point numb
er.

COUNT -- Returns the amount of records that are returned by a SELECT query.

DIV -- Allows you to divide integers.

EXP -- Returns e to the power of the given number.

FLOOR -- Returns the closest whole number (integer) downwards from a given decimal point n
umber.

GREATEST -- Returns the highest value in a list of arguments.

LEAST -- Returns the smallest value in a list of arguments.

MAX -- Returns the highest value from a set of values.

MIN -- Returns the lowest value from a set of values.

MOD -- Returns the remainder of the given number divided by the other given number.

POW -- Returns the value of the given number raised to the power of the other given numbe
r.

RAND -- Returns a random number.

ROUND -- Rounds the given number to the given amount of decimal places.

SQRT -- Returns the square root of the given number.

SUM -- Returns the value of the given set of values combined.

TRUNCATE -- Returns a number truncated to the given number of decimal places.
```

Round	Number	Output
2	123456.789	123456.79
1	123456.789	123456.8
0	123456.789	123457
-1	123456.789	123460
-2	123456.789	123500
-3	123456.789	123000

▼ DATE FUNCTIONS

used with GROUP BY and aggregate functions like COUNT() and SUM() to perform trend analysis. Some of the common ones are

```
SELECT
    birthdate,
    MONTHNAME(birthdate),
    YEAR(birthdate)
FROM people;
```



Function	How You Might Use It
YEAR()	Return the year for a given date
QUARTER()	Return the quarter for a given date
MONTH()	Return the month for a given date
WEEK()	Return the week for a given date
DATE()	Return the date for a given datetime
NOW()	Calculate time relative to now

DATE_FORMAT (*date*, *format*)

```
/*%a Abbreviated weekday name (Sun to Sat)
%b Abbreviated month name (Jan to Dec)
%D Day of the month as a numeric value, followed by suffix (1st, 2nd, 3rd, ...)*/
SELECT birthdate, DATE_FORMAT(birthdate, '%a %b %D') FROM people;

/*%H Hour (00 to 23)
%i Minutes (00 to 59)*/
SELECT birthdt, DATE_FORMAT(birthdt, '%H:%i') FROM people;

/*%r Time in 12 hour AM or PM format (hh:mm:ss AM/PM)*/
SELECT birthdt, DATE_FORMAT(birthdt, 'BORN ON: %r') FROM people;
```

DATEDIFF()

returns the **number of days** between two date values

```
SELECT DATEDIFF("2017-01-01", "2016-12-24");

SELECT DATEDIFF(CURDATE(), birthdate) AS current_age FROM people;
```

There are DATE_ADD and DATE_SUB for more granular control

```
/*After your device-level analysis of conversion rates, we
realized desktop was doing well, so we bid our gsearch
nonbrand desktop campaigns up on 2012-05-19.
Could you pull weekly trends for both desktop and mobile
so we can see the impact on volume?
You can use 2012-04-15 until the bid change as a baseline.*/

SELECT
  MIN(DATE (created_at)) AS week_start_date,
  COUNT(CASE WHEN device_type='desktop'
            THEN website_session_id
            ELSE null END) AS dtop_sessions,
  COUNT(CASE WHEN device_type='mobile'
              THEN website_session_id
              ELSE null END) AS mob_sessions
FROM website_sessions
WHERE
  created_at BETWEEN '2012-04-15' AND '2012-06-9'
    AND utm_source='gsearch'
    AND utm_campaign='nonbrand'
GROUP BY WEEK(created_at);
```

▼ STRING FUNCTIONS

Like most data types, strings can be compared for equality or inequality or relative ordering. However, strings have some additional properties to consider:

- Strings can be case sensitive (or not), which can affect the outcome of string operations.
- You can compare entire strings, or just parts of them by extracting substrings.
- You can apply pattern-matching operations to look for strings that have a certain structure.

CONCAT

joins two or more strings together

```
SELECT CONCAT(author_fname,' ', author_lname) AS author_name FROM books;

/*CONCAT_WS adds two or more strings together with a separator*/
SELECT CONCAT_WS('-',title, author_fname, author_lname) FROM books;

+-----------------------------------+
| title - author_fname - author_lname |
+-----------------------------------+


SELECT name, LEFT(name,2), MID(name,3,1), RIGHT(name,3) FROM metal;
+----------+--------------+---------------+---------------+
| name     | LEFT(name,2) | MID(name,3,1) | RIGHT(name,3) |
+----------+--------------+---------------+---------------+
| copper   | co           | p             | per           |
| gold     | go           | l             | old           |
| iron     | ir           | o             | ron           |
| lead     | le           | a             | ead           |
| mercury  | me           | r             | ury           |
| platinum | pl           | a             | num           |
| silver   | si           | l             | ver           |
| tin      | ti           | n             | tin           |
+----------+--------------+---------------+---------------+
```

SUBSTRING

extracts user-provided characters from a string, index starts at 1

To return everything to the right or left of a given character, use `SUBSTRING_INDEX( str , c , n )`. It searches into a string `str` for the `n`-th occurrence of the character `c` and returns everything to its left. If `n` is negative, the search for `c` starts from the right and returns everything to the right of the character

```
SELECT SUBSTRING('Hello World', 1, 4); --Hell

SELECT SUBSTRING('Hello World', 7); --Hello W

SELECT SUBSTRING(title, 1, 10) AS 'short title' FROM books;


SELECT name,
      SUBSTRING_INDEX(name,'r',2),
      SUBSTRING_INDEX(name,'i',-1)
FROM metal;
+----------+---------------------------+----------------------------+
| name     | SUBSTRING_INDEX(name,'r',2) | SUBSTRING_INDEX(name,'i',-1) |
+----------+---------------------------+----------------------------+
| copper   | copper                    | copper                     |
| gold     | gold                      | gold                       |
| iron     | iron                      | ron                        |
| lead     | lead                      | lead                       |
| mercury  | mercu                     | mercury                    |
| platinum | platinum                  | num                        |
| silver   | silver                    | lver                       |
| tin      | tin                       | n                          |
+----------+---------------------------+----------------------------+
```

REPLACE

replaces all occurrences of a substring within a string, with a new substring

```
SELECT REPLACE('Hello World', 'Hell', '%$#@');

SELECT REPLACE(title, 'e ', '3') FROM books;

SELECT REPLACE(title, ' ', '-') FROM books;
```

CHAR_LENGTH

return the length of a string (in characters)

```
SELECT CHAR_LENGTH('Hello World');

SELECT title, CHAR_LENGTH(title) as length FROM books;

SELECT CONCAT(author_lname, ' is ', CHAR_LENGTH(author_lname), ' characters long')
FROM books;
```

LOCATE & POSITION

```
SELECT LOCATE('is', vchar_fld, 5) #start at position 5
    -> FROM string_tbl;
#'This string is 28 characters'
+---------------------------+
| LOCATE('is', vchar_fld, 5) |
+---------------------------+
|                         13 |
+---------------------------+

#No option to provide starting position
SELECT POSITION('characters' IN vchar_fld)
    -> FROM string_tbl;
+------------------------------------+
| POSITION('characters' IN vchar_fld) |
+------------------------------------+
|                                  19 |
+------------------------------------+
```

Pattern Matching Using REGEXP (Using LIKE is discussed previously)

Pattern	What the Pattern matches
*	Zero or more instances of string preceding it
+	One or more instances of strings preceding it
.	Any single character
?	Match zero or one instances of the strings preceding it.
^	caret(^) matches Beginning of string
$	End of string
[abc]	Any character listed between the square brackets
[^abc]	Any character not listed between the square brackets
[A-Z]	match any upper case letter.
[a-z]	match any lower case letter
[0-9]	match any digit from 0 through to 9.
[[:<:]]	matches the beginning of words.
[[:>:]]	matches the end of words.
[:class:]	matches a character class i.e. [:alpha:] to match letters, [:space:] to match white space, [:punct:] is match punctuations and [:upper:] for upper class letters.
p1|p2|p3	Alternation; matches any of the patterns p1, p2, or p3
{n}	n instances of preceding element

{m,n}	m through n instances of preceding element

```
#Strings that begin with a particular substring
SELECT name FROM metal WHERE name REGEXP '^co';
+--------+
| name   |
+--------+
| copper |
+--------+

#Strings that end with a particular substring
SELECT name FROM metal WHERE name REGEXP 'er$';
+--------+
| name   |
+--------+
| copper |
| silver |
+--------+

#Strings that contain a particular substring at any position
SELECT name FROM metal WHERE name REGEXP 'er';
+---------+
| name    |
+---------+
| copper  |
| mercury |
| silver  |
+---------+

#Strings that contain a particular substring at a specific position
SELECT name FROM metal WHERE name REGEXP '^..pp';
+--------+
| name   |
+--------+
| copper |
+--------+

#Alternation example
SELECT name FROM metal WHERE name REGEXP '^[aeiou]|er$';
+--------+
| name   |
+--------+
| copper |
| iron   |
| silver |
+--------+

#Parentheses may be used to group alternations. For example, if you want to match strings
 that consist entirely of digits or entirely of letters, you might try this pattern, using
an alternation
SELECT '0m' REGEXP '^[[:digit:]]+|[[:alpha:]]+$';
+-------------------------------------------+
| '0m' REGEXP '^[[:digit:]]+|[[:alpha:]]+$' |
+-------------------------------------------+
|                                         1 |
+-------------------------------------------+
```

A pattern consisting only of either SQL metacharacter matches all the values in the table, not just the metacharacter itself:

```
SELECT c, c LIKE '%', c LIKE '_' FROM metachar;
+------+------------+------------+
| c    | c LIKE '%' | c LIKE '_' |
+------+------------+------------+
| %    |          1 |          1 |
| _    |          1 |          1 |
| .    |          1 |          1 |
| ^    |          1 |          1 |
| $    |          1 |          1 |
| \    |          1 |          1 |
+------+------------+------------+
```

To match a literal instance of a SQL pattern metacharacter, precede it with a backslash:

```
SELECT c, c LIKE '\%', c LIKE '\_' FROM metachar;
+------+-------------+-------------+
| c    | c LIKE '\%' | c LIKE '\_' |
+------+-------------+-------------+
| %    |           1 |           0 |
| _    |           0 |           1 |
| .    |           0 |           0 |
| ^    |           0 |           0 |
| $    |           0 |           0 |
| \    |           0 |           0 |
+------+-------------+-------------+
```

With `REGEXP`, you need a double backslash to match a metacharacter literally:

```
SELECT c, c REGEXP '\\.', c REGEXP '\\^', c REGEXP '\\$' FROM metachar;
+------+----------------+----------------+----------------+
| c    | c REGEXP '\\.' | c REGEXP '\\^' | c REGEXP '\\$' |
+------+----------------+----------------+----------------+
| %    |              0 |              0 |              0 |
| _    |              0 |              0 |              0 |
| .    |              1 |              0 |              0 |
| ^    |              0 |              1 |              0 |
| $    |              0 |              0 |              1 |
| \    |              0 |              0 |              0 |
+------+----------------+----------------+----------------+
```

Full list of MySQL Functions

◆ Advanced SQL

▼ Joins

Database normalization is useful because it minimizes duplicate data in any single table, and allows for data in the database to grow independently of each other. As a trade-off, queries get slightly more complex since they have to be able to find data from different parts of the database. In order to answer questions about an entity that has data spanning multiple tables in a normalized database, we use JOINs.

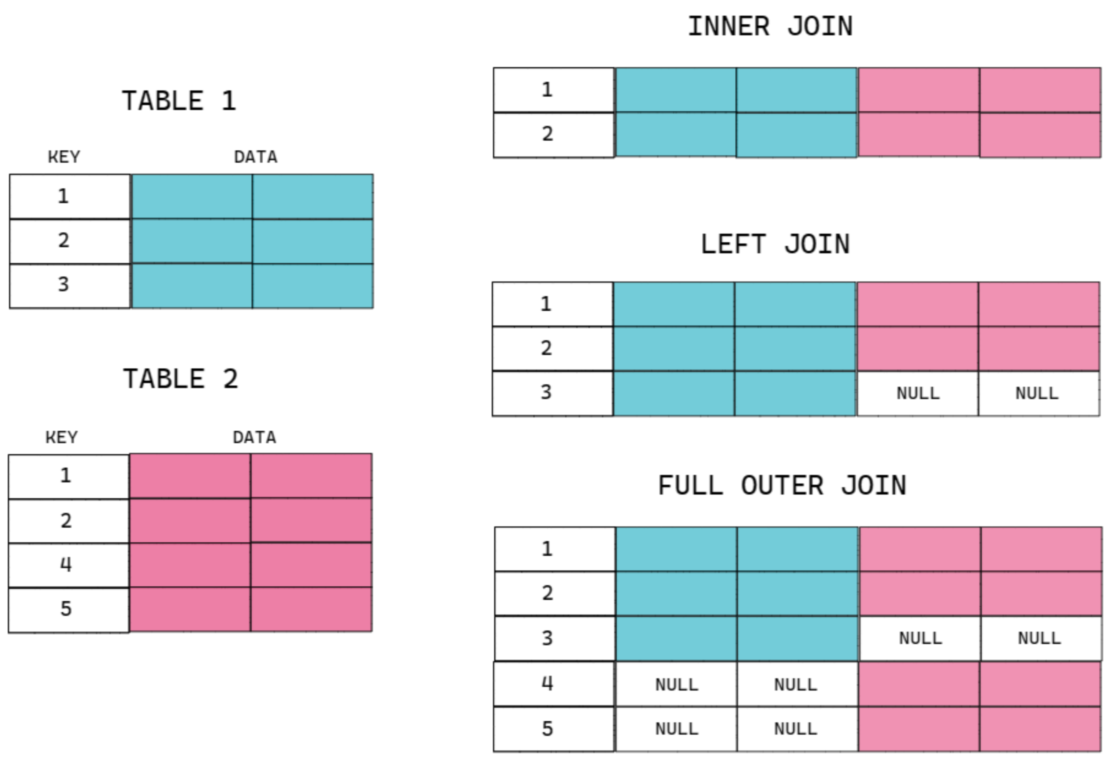
The following are the common join types:

An **INNER JOIN** returns only those rows where the linking values match in both of the tables or in result sets.

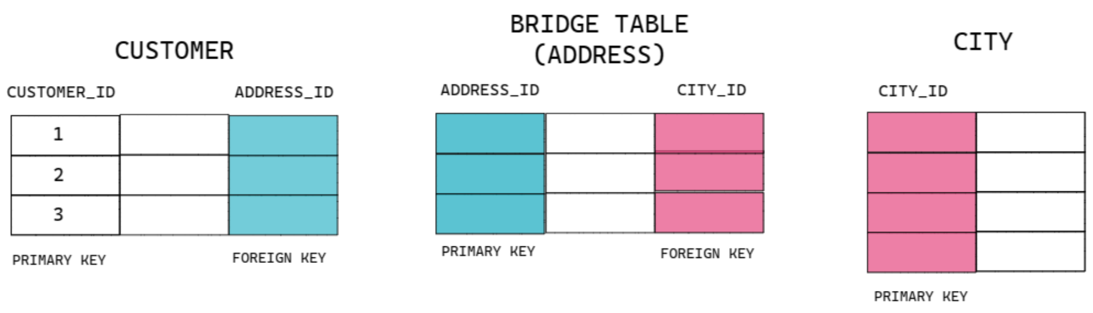
```
SELECT ColA, ColB, ColX, ColY
FROM Table1 INNER JOIN Table2
  ON Table1.KeyID = Table2.KeyID

--or

SELECT t1.ColA, t1.ColB, t2.ColX, t2.ColY
FROM Table1 t1 INNER JOIN Table2 t2
  ON t1.KeyID = t2.KeyID
```

A **FULL OUTER JOIN** or **FULL JOIN** asks your database system to return not only the rows that match the criteria you specify but also the unmatched rows from either one or both of the two sets you want to link

What’s the difference between LEFT JOIN and RIGHT JOIN? Remember that to specify an INNER JOIN on two tables, you name the first table, include the JOIN keyword, and then name the second table. When you begin building queries using OUTER JOIN, the SQL Standard considers the first table you name as the one on the “left,” and the second table as the one on the “right.” So, if you want all the rows from the first table and any matching rows from the second table, you’ll use a **LEFT OUTER JOIN** or **LEFT JOIN**. Conversely, if you want all the rows from the second table and any matching rows from the first table, you’ll specify a **RIGHT OUTER JOIN** or **RIGHT JOIN**.



Bridging unrelated tables:



```
SELECT
        customer.name AS customer_name,
        city.name AS city_name
FROM customer
INNER JOIN address
        ON customer.address_id = address.address_id
INNER JOIN city
        ON address.city_id = city.city_id
```

Multi-condition Joins:

```
SELECT
  film.film_id,
  film.title,
  category.name AS category_name
```

```
FROM film
INNER JOIN film category
  ON film.film_id = film.category_id
INNER JOIN category
  ON film_category.category_id = category.category_id
WHERE category.name = 'horror' --This can be included in the join statements
ORDER BY film_id

SELECT
  film.film_id,
  film.title,
  category.name AS category_name
FROM film
INNER JOIN film category
  ON film.film_id = film.category_id
INNER JOIN category
  ON film_category.category_id = category.category_id
  AND category.name = 'horror'
ORDER BY film_id

#SELF JOIN
SELECT f.title, f_prnt.title prequel
    FROM film f
    INNER JOIN film f_prnt
    ON f_prnt.film_id = f.prequel_film_id
    WHERE f.prequel_film_id IS NOT NULL;

#Using the Same table twice
SELECT f.title
    FROM film f
    INNER JOIN film_actor fa1
      ON f.film_id = fa1.film_id
    INNER JOIN actor a1
      ON fa1.actor_id = a1.actor_id
    INNER JOIN film_actor fa2
      ON f.film_id = fa2.film_id
    INNER JOIN actor a2
      ON fa2.actor_id = a2.actor_id
    WHERE (a1.first_name = 'CATE' AND a1.last_name = 'MCQUEEN')
        AND (a2.first_name = 'CUBA' AND a2.last_name = 'BIRCH');

+------------------+
| title            |
+------------------+
| BLOOD ARGONAUTS  |
| TOWERS HURRICANE |
+------------------+
```

▼ Subqueries

allows us to query another query. Simply put, a *subquery* is a SELECT expression that you embed inside one of the clauses of a SELECT statement to form your final query statement.

📌 Subqueries can be used in different parts of a SQL statement such as SELECT, FROM, WHERE, and HAVING.
An alias must always be provided when using with FROM statement.

📌 When you use a subquery in the `FROM` clause, the result set returned from a subquery is used as a **temporary table**. This table is referred to as a **derived table** or materialized subquery.

The SQL Standard defines three types of subqueries:

1. **Table subquery -** an embedded SELECT expression that returns one or more columns and zero to many rows

```
--Example: Find the names of the employees who work in the same department as John.

SELECT emp_name
FROM employees
WHERE dept_id = (SELECT dept_id FROM employees WHERE emp_name = 'John');

--In this example, the subquery (SELECT dept_id FROM employees WHERE emp_name = 'Joh
n') returns multiple rows (all the departments where an employee named John works) wh
ich is used to compare with the department ID of each employee in the main query.

--Another example of a multiple-row subquery is using a subquery as a table in a FROM
clause:

SELECT * FROM
  (SELECT * FROM website_sessions
    WHERE website_session_id <= 100) AS first_hundred;
--Alias must always be provided when using with FROM statement

--Example: Find the total sales for each employee in the company.

SELECT emp_name, total_sales
FROM employees
INNER JOIN (SELECT emp_id, SUM(sales_amount) as total_sales
            FROM sales GROUP BY emp_id) as sales
ON employees.emp_id = sales.emp_id;

--In this example, the subquery (SELECT emp_id, SUM(sales_amount) as total_sales FROM
sales GROUP BY emp_id) returns a table with multiple rows (the total sales for each e
mployee) which is used as a table in the main query to join with the employees table.

--Another example of a multiple-row subquery is using a subquery as a table in a SELE
CT clause:

SELECT Orders.OrderNumber
      ,Orders.OrderDate
      ,Orders.ShipDate
      ,(SELECT Customers.CustLastName
        FROM Customers
        WHERE Customers.CustomerID = Orders.CustomerID)
```

```
FROM Orders
WHERE Orders.ShipDate = '2017-10-03'
```

2. **Row subquery -** an embedded SELECT expression that returns more than one column but no more than one row

```
SELECT SKUClass, SKUNumber, ProductName
FROM Products
WHERE (SKUClass, SKUNumber) = ('DSK', 9775)
```

3. **Scalar subquery -** an embedded SELECT expression that returns only one column and no more than one row

```
--Example: Find the name of the employee who has the highest salary in the company.

SELECT emp_name
FROM employees
WHERE salary = (SELECT MAX(salary) FROM employees);

--The subquery (SELECT MAX(salary) FROM employees) returns a single value (the maximu
m salary) which is used to compare with the salary of each employee in the main quer
y.
```

▼ Temporary Tables

transient tables that exist only for the duration of the current session or transaction. Temporary tables are useful for storing intermediate results, breaking down complex operations into smaller steps.

```
CREATE TEMPORARY TABLE full_reviews
SELECT title, released_year, genre, rating, first_name, last_name FROM reviews
JOIN series ON series.id = reviews.series_id
JOIN reviewers ON reviewers.id = reviews.reviewer_id;
```

Some APIs support persistent connections in a web environment. Use of these prevents temporary tables from being dropped as you expect when your script ends, because the web server keeps the connection open for reuse by other scripts. (The server may close the connection eventually, but you have no control over when that happens.) This means it can be prudent to issue the following statement prior to creating a temporary table, just in case it's still hanging around from the previous execution of the script: `DROP TABLE IF EXISTS` `tbl_name`

▼ Views

a virtual table based on the result set of an SQL statement. The tables that comprise the view are known as *base tables.* A view contains rows and columns, just like a real table, but there is no data associated with a view, it draws data from base tables rather than storing any data on its own (this is why it's called a *virtual* table). When you issue a query against a view, your query is merged with the view definition to create a final query to be executed.

```
-- INSTEAD OF TYPING THIS QUERY ALL THE TIME...
SELECT title, released_year, genre, rating, first_name, last_name
FROM reviews
JOIN series ON series.id = reviews.series_id
JOIN reviewers ON reviewers.id = reviews.reviewer_id;

-- WE CAN CREATE A VIEW:
CREATE VIEW full_reviews AS
SELECT title, released_year, genre, rating, first_name, last_name FROM reviews
JOIN series ON series.id = reviews.series_id
JOIN reviewers ON reviewers.id = reviews.reviewer_id;

-- NOW WE CAN TREAT THAT VIEW AS A VIRTUAL TABLE
-- (AT LEAST WHEN IT COMES TO SELECTING)
SELECT * FROM full_reviews;
```

For a view to be updatable, there must be a one-to-one relationship between the rows in the view and the rows in the underlying table. There are also certain other constructs that make a view non-updatable. To be more specific, a view is not updatable if it contains any of the following:

- Aggregate functions or window functions (`SUM()`, `MIN()`, `MAX()`, `COUNT()`, and so forth)
- `DISTINCT`
- `GROUP BY`
- `HAVING`
- `UNION` or `UNION ALL`
- Subquery in the select list

▼ VIEWS vs TEMP TABLES

`VIEW` and `TEMPORARY TABLE` are both database objects in SQL that serve different purposes. Here's a comparison of their characteristics and use cases:

1. Definition and Purpose:
    - `VIEW`: A virtual table based on the result of a SELECT query. It does not store data itself but instead represents a predefined query on one or more existing tables. Views can simplify complex queries, provide data security by restricting access to specific columns, and improve code maintainability.

- `TEMPORARY TABLE`: A temporary, non-persistent table that exists only for the duration of the current session or transaction. Temporary tables are useful for storing intermediate results, breaking down complex operations into smaller steps, or optimizing performance for certain tasks.

2. Data Storage:

   - `VIEW`: Does not store data; instead, it reflects the data stored in the underlying tables. When you query a view, the database system executes the underlying SELECT query and fetches the data from the base tables.
   - `TEMPORARY TABLE`: Stores data temporarily in memory or on disk, depending on the database system. The data in a temporary table is only available to the session or transaction that created it and is automatically deleted when the session or transaction ends.

3. Efficiency:

   - `VIEW`: The efficiency of a view depends on the complexity of the underlying query and the database system's query optimization. Querying a view can be slower than querying the base tables directly, as the database needs to execute the underlying SELECT query each time the view is accessed.
   - `TEMPORARY TABLE`: Can improve performance in certain scenarios, such as when intermediate results need to be used multiple times or when complex operations can be divided into smaller steps. However, the overhead of creating, populating, and deleting temporary tables can impact performance.

▼ Window Functions

Window functions perform calculations across a group of rows but produce a result `FOR EACH ROW`, allowing you to create more advanced and flexible data analyses. They are called "window functions" because they operate on a "window" of rows defined by an `OVER()` clause.

Window functions can be used to calculate running totals, cumulative sums, moving averages, rankings, percentiles, and other complex calculations that depend on the context of the current row in relation to other rows in the result set.

The general structure of a window function is as follows:

```
<function>(<expression>) OVER (
    [PARTITION BY <partition_expression>]
    [ORDER BY <order_expression>]
    [ROWS <frame_specification>]
)
```

1. `<function>`: A window function, such as ROW_NUMBER(), RANK(), DENSE_RANK(), NTILE(), LEAD(), LAG(), SUM(), AVG(), MIN(), MAX(), etc.
2. `<expression>`: The column or expression on which the window function operates.
3. `PARTITION BY`: Optional. It divides the result set into partitions to which the window function is applied. If not specified, the function treats the whole result set as a single partition.
4. `ORDER BY`: Optional. It orders the rows within each partition. The window function is applied based on this order.
5. `ROWS <frame_specification>`: Optional. It defines the set of rows, or "frame", within the partition that the window function operates on, relative to the current row.

```
SELECT
    emp_no,
    department,
    salary,
    MIN(salary) OVER(),
    MAX(salary) OVER()
FROM employees;

SELECT
    emp_no,
    department,
    salary,
    AVG(salary) OVER(PARTITION BY department) AS dept_avg,
FROM employees;

SELECT
    region,
    product,
    sales_amount,
    SUM(sales_amount) OVER (PARTITION BY region ORDER BY product) as running_total
FROM sales_data;
```

▼ RANK, DENSE_RANK & ROW_NUMBER

`RANK`, `DENSE_RANK`, and `ROW_NUMBER` are window functions in SQL used to assign a unique rank or number to each row within a result set, based on the values in one or more columns. They are often used in scenarios where you need to rank or number rows, such as leaderboard generation, competition scoring, or top-N analysis.

1. `RANK`: Assigns a unique rank to each row within the result set, with the same rank assigned to rows with equal values. When multiple rows share the same rank, the next rank will be skipped. The syntax for `RANK` is:
2. `DENSE_RANK`: Assigns a unique rank to each row within the result set, with the same rank assigned to rows with equal values. Unlike `RANK`, `DENSE_RANK` does not skip any rank numbers when there are multiple rows with the same rank. The syntax for `DENSE_RANK` is:

3. `ROW_NUMBER`: Assigns a unique number to each row within the result set, regardless of the values in the columns. Even rows with equal values will receive different row numbers. The syntax for `ROW_NUMBER` is:

Here's an example that demonstrates the difference between `RANK`, `DENSE_RANK`, and `ROW_NUMBER`:

Assume we have a `sales_data` table with the following records:

salesperson	sales_amount
Alice	1000
Bob	2000
Carol	2000
Dave	3000

The following query will calculate the `RANK`, `DENSE_RANK`, and `ROW_NUMBER` for each salesperson based on their `sales_amount`:

```
SELECT
    salesperson,
    sales_amount,
    RANK() OVER (ORDER BY sales_amount DESC) as rank,
    DENSE_RANK() OVER (ORDER BY sales_amount DESC) as dense_rank,
    ROW_NUMBER() OVER (ORDER BY sales_amount DESC) as row_number
FROM sales_data;

| salesperson | sales_amount | rank | dense_rank | row_number |
| --- | --- | --- | --- | --- |
| Dave | 3000 | 1 | 1 | 1 |
| Bob | 2000 | 2 | 2 | 2 |
| Carol | 2000 | 2 | 2 | 3 |
| Alice | 1000 | 4 | 3 | 4 |
```

As you can see, `RANK` skips the rank 3 (due to Bob and Carol sharing rank 2), while `DENSE_RANK` does not skip any rank. `ROW_NUMBER` assigns a unique number to each row, even though Bob and Carol have the same `sales_amount`.

```
SELECT
    emp_no,
    department,
    salary,
    ROW_NUMBER() OVER(PARTITION BY department ORDER BY SALARY DESC) as dept_row_numbe
r,
    RANK() OVER(PARTITION BY department ORDER BY SALARY DESC) as dept_salary_rank,
    RANK() OVER(ORDER BY salary DESC) as overall_rank,
    DENSE_RANK() OVER(ORDER BY salary DESC) as overall_dense_rank,
    ROW_NUMBER() OVER(ORDER BY salary DESC) as overall_num
FROM employees ORDER BY overall_rank;
```

▼ LEAD vs LAG

`LEAD` and `LAG` are window functions in SQL that allow you to access the value of a row ahead or behind the current row within a result set, making it easier to compare values between rows or calculate differences, growth rates, or other relative metrics.

1. `LEAD`: Retrieves the value of a given expression for the row that is N rows *after* the current row, within the same result set. The syntax for `LEAD` is:

```
LEAD(<expression>, <offset>, <default>) OVER (
    [PARTITION BY <partition_expression>]
    ORDER BY <order_expression>
)
```

- `<expression>`: The column or expression for which the value will be fetched.
- `<offset>`: Optional. The number of rows ahead of the current row to fetch the value from. Defaults to 1.
- `<default>`: Optional. The value to return if the lead value is not available (e.g., when the current row is the last row).

1. `LAG`: Retrieves the value of a given expression for the row that is N rows *before* the current row, within the same result set. The syntax for `LAG` is:

```
LAG(<expression>, <offset>, <default>) OVER (
    [PARTITION BY <partition_expression>]
    ORDER BY <order_expression>
)
```

- `<expression>`: The column or expression for which the value will be fetched.
- `<offset>`: Optional. The number of rows behind the current row to fetch the value from. Defaults to 1.
- `<default>`: Optional. The value to return if the lag value is not available (e.g., when the current row is the first row).

Both `LEAD` and `LAG` require an `ORDER BY` clause within the `OVER()` clause to define the order of rows within the result set.

Here's an example using `LEAD` and `LAG` to calculate the difference between the current sales amount and the previous and next sales amounts:

```
SELECT
    region,
    product,
    sales_amount,
    sales_amount - LAG(sales_amount, 1) OVER (PARTITION BY region ORDER BY product) as
```

```
prev_difference,
    LEAD(sales_amount, 1) OVER (PARTITION BY region ORDER BY product) - sales_amount a
s next_difference
FROM sales_data;
```

In this example, the `LAG` function calculates the difference between the current `sales_amount` and the previous `sales_amount` within each `region`, while the `LEAD` function calculates the difference between the current `sales_amount` and the next `sales_amount`.

▼ NTILE

`NTILE()` is a window function in SQL that is used to divide the result set into a specified number of groups or "tiles" with roughly equal numbers of rows. It assigns each row a tile number based on the ordering specified in the `OVER()` clause. `NTILE()` is useful for scenarios such as dividing data into quartiles, percentiles, or other equal-sized groups for further analysis.

```
SELECT
    emp_no,
    department,
    salary,
    NTILE(4) OVER(PARTITION BY department ORDER BY salary DESC) AS dept_salary_quartil
e,
    NTILE(4) OVER(ORDER BY salary DESC) AS salary_quartile
FROM employees;
```

▼ FIRST_VALUE, LAST_VALUE and NTH_VALUE

`FIRST_VALUE`, `LAST_VALUE`, and `NTH_VALUE` are window functions in SQL that allow you to access the first, last, or Nth value of a specified column within a window frame. These functions are useful for comparing values between rows, calculating differences, or retrieving specific values based on ordering.

1. `FIRST_VALUE`: Retrieves the first value of the specified column in the window frame. The syntax for `FIRST_VALUE` is:

```
FIRST_VALUE(<expression>) OVER (
    [PARTITION BY <partition_expression>]
    ORDER BY <order_expression>
    [ROWS <frame_specification>]
)
```

1. `LAST_VALUE`: Retrieves the last value of the specified column in the window frame. The syntax for `LAST_VALUE` is:

```
LAST_VALUE(<expression>) OVER (
    [PARTITION BY <partition_expression>]
    ORDER BY <order_expression>
    [ROWS <frame_specification>]
)
```

1. `NTH_VALUE`: Retrieves the Nth value of the specified column in the window frame. The syntax for `NTH_VALUE` is:

```
NTH_VALUE(<expression>, <N>) OVER (
    [PARTITION BY <partition_expression>]
    ORDER BY <order_expression>
    [ROWS <frame_specification>]
)
```

- `<expression>`: The column or expression for which the value will be fetched.
- `<partition_expression>`: Optional. It divides the result set into partitions to which the window function is applied. If not specified, the function treats the whole result set as a single partition.
- `<order_expression>`: Orders the rows within each partition. The window function is applied based on this order.
- `<frame_specification>`: Optional. It defines the set of rows, or "frame", within the partition that the window function operates on, relative to the current row.
- `<N>`: The position of the value to be retrieved within the window frame, where N is a positive integer.

Here's an example that demonstrates the use of `FIRST_VALUE`, `LAST_VALUE`, and `NTH_VALUE` to retrieve the first, last, and second sales amounts for each region:

```
SELECT
    region,
    product,
    sales_amount,
    FIRST_VALUE(sales_amount) OVER (PARTITION BY region ORDER BY product) as first_val
ue,
    LAST_VALUE(sales_amount) OVER (PARTITION BY region ORDER BY product ROWS BETWEEN U
NBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) as last_value,
    NTH_VALUE(sales_amount, 2) OVER (PARTITION BY region ORDER BY product) as nth_valu
e
FROM sales_data;
```

In this example, the `FIRST_VALUE`, `LAST_VALUE`, and `NTH_VALUE` functions retrieve the first, last, and second sales amounts within each `region` as the rows are ordered by `product`. Note the frame specification `ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING` for `LAST_VALUE`, which ensures that the entire window frame is considered when retrieving the last value.

▼ Variables

A common situation in which SQL variables come in handy is when you need to issue successive queries on multiple tables that are related by a common key value. Suppose you have a customers table with a `cust_id` column that identifies each customer, and an orders table that also has a `cust_id` column to indicate which customer each order is associated with. If you have a customer name and you want to delete the customer record as well as all the customer's orders, you need to determine the proper `cust_id` value for that customer, then delete records from both the customers and orders tables that match the ID. One way to do this is to first save the ID value in a variable, then refer to the variable in the DELETE statements.

```
SELECT @id := cust_id
FROM customers
WHERE cust_id='customer name';

DELETE FROM customers
WHERE cust_id = @id;

DELETE FROM orders
WHERE cust_id = @id;
```

SQL variables hold single values. If you assign a value to a variable using a statement that returns multiple rows, the value from the last row is used. Also, Variable names are case sensitive

```
SET @x = 1; SELECT @x, @X;

--+------+------+
--| @x   | @X   |
--+------+------+
--|    1 | NULL |
--+------+------+
```