# General Architecture

Client: Where the request initiated.

Query: SQL query which is high level language.

Logical Units: Keywords, expressions and operators, etc.

N/W Packets: Network related code.

Protocols: In SQL Server we have 4 protocols.

Shared memory (for local connections and troubleshooting purpose).

Named pipes (for connections which are in LAN connectivity).

TCP/IP (for connections which are in WAN connectivity).

VIA-Virtual Interface Adapter (requires special hardware to set up by vendor).

Server: Where SQL Services got installed and databases reside.

Relational Engine: This is where real execution will be done. It contains Query parser, Query optimizer and Query executor.

Query Parser (Command Parser) and Compiler (Translator): This will check syntax of the query and it will convert the query to machine language.

Query Optimizer: It will prepare the execution plan as output by taking query, statistics and Algebrizer tree as input.

Execution Plan: It is like a roadmap, which contains the order of all the steps to be performed as part of the query execution.

Query Executor: This is where the query will be executed step by step with the help of execution plan and also the storage engine will be contacted.

Storage Engine: It is responsible for storage and retrieval of data on the storage system (disk, SAN, etc.,), data manipulation, locking and managing transaction

In MS SQL Server, two types of databases are available.

System databases - System databases are created automatically when we install MS SQL Server. Following is a list of system databases: · Master · Model · MSDB · Tempdb

User Databases -

Data Manipulation Language (DML for short) includes operations such as INSERT, UPDATE and DELETE:

SELECT DISTINCT column1, column2, ...FROM table\_name WHERE condition ORDER BY column1, column2, ... ASC|DESC;

The ORDER BY keyword sorts the records in ascending order by default. To sort the records in descending order, use the DESC keyword.

INSERT INTO table\_name (column1, column2, column3, ...) VALUES (value1, value2, value3, ...);

If you are adding values for all the columns of the table, you do not need to specify the column names in the SQL query

To insert multiple rows of data, we use the same INSERT INTO statement, but with multiple values with comma separated.

It is not possible to test for NULL values with comparison operators, such as =, <, or <>. We will have to use the IS NULL and IS NOT NULL operators instead.

UPDATE table\_name SET column1 = value1, column2 = value2, ... WHERE condition;

DELETE FROM table\_name WHERE condition;

SELECT TOP number|*percent* column\_name(s) FROM table\_name WHERE condition;

The MIN() function returns the smallest value of the selected column. The MAX() function returns the largest value of the selected column. The COUNT() function returns the number of rows. The SUM() function returns the total sum of a numeric column. The AVG() function returns the average value of a numeric column.

SELECT COUNT(DISTINCT Price) FROM Products;

The LIKE operator is used in a WHERE clause to search for a specified pattern in a column. There are two wildcards often used in conjunction with the LIKE operator:

* The percent sign (%) represents zero, one, or multiple characters
* The underscore sign (\_) represents one, single character

A wildcard character is used to substitute one or more characters in a string. Wildcard characters are used with the [LIKE](https://www.w3schools.com/sql/sql_like.asp) operator.

% Represents zero or more characters

\_ Represents a single character --> h\_t finds hot, hat, and hit

[] any single char within the brackets --> h[oa]t finds hot and hat, but not hit

^ any character not in the brackets --> h[^oa]t finds hit, but not hot and hat

- any single character within the specified range --> c[a-b]t finds cat and cbt

The IN operator allows you to specify multiple values in a WHERE clause. The IN operator is a shorthand for multiple OR conditions.

The BETWEEN operator selects values within a given range. The values can be numbers, text, or dates. The BETWEEN operator is inclusive: begin and end values are included.

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.

JOINS

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

* (INNER) JOIN: Returns records that have matching values in both tables
* LEFT (OUTER) JOIN: Returns all records from the left table, and the matched records from the right table
* RIGHT (OUTER) JOIN: Returns all records from the right table, and the matched records from the left table
* FULL (OUTER) JOIN: Returns all records when there is a match in either left or right table
* A self join is a regular join, but the table is joined with itself.

**Inner joins** can be specified in either the FROM or WHERE clauses.

**Outer joins** and **cross joins** can be specified in the FROM clause only.

* Tables cannot be joined directly on ntext, text, or image columns. However, tables can be joined indirectly on ntext, text, or image columns by using SUBSTRING.
  + SELECT \* FROM t1 JOIN t2 ON SUBSTRING(t1.textcolumn, 1, 20) = SUBSTRING(t2.textcolumn, 1, 20)
* Another possibility for comparing ntext or text columns from two tables is to compare the lengths of the columns with a WHERE clause, for example:
  + WHERE DATALENGTH(p1.pr\_info) = DATALENGTH(p2.pr\_info)

The UNION operator is used to combine the result-set of two or more SELECT statements.

* Every SELECT statement within UNION must have the same number of columns
* The columns must also have similar data types
* The columns in every SELECT statement must also be in the same order

The UNION operator selects only distinct values by default. To allow duplicate values, use UNION ALL:

SELECT column\_name(s) FROM table1  
UNION ALL  
SELECT column\_name(s) FROM table2;

The GROUP BY statement groups rows that have the same values into summary rows, like "find the number of customers in each country".

The GROUP BY statement is often used with aggregate functions (COUNT(), MAX(), MIN(), SUM(), AVG()) to group the result-set by one or more columns.

The HAVING clause was added to SQL because the WHERE keyword cannot be used with aggregate functions.

SELECT COUNT(CustomerID), Country  
FROM Customers  
GROUP BY Country  
HAVING COUNT(CustomerID) > 5  
ORDER BY COUNT(CustomerID) DESC;

The EXISTS operator is used to test for the existence of any record in a subquery. The EXISTS operator returns TRUE if the subquery returns one or more records.

SELECT column\_name(s) FROM table\_name WHERE EXISTS  
(SELECT column\_name FROM table\_name WHERE condition);

The ANY and ALL operators allow you to perform a comparison between a single column value and a range of other values.

* ANY returns TRUE if ANY of the subquery values meet the condition
* ALL returns TRUE if ALL of the subquery values meet the condition

SELECT column\_name(s) FROM table\_name  
WHERE column\_name operator ANY/ALL  
  (SELECT column\_name FROM table\_name WHERE condition);

The SELECT INTO statement copies data from one table into a new table.

SELECT column1, column2, column3, ... (or \*)  
INTO newtable [IN externaldb]  
FROM oldtableWHERE condition;

SELECT INTO can also be used to create a new, empty table using the schema of another. Just add a WHERE clause that causes the query to return no data:

SELECT \* INTO newtable FROM oldtable WHERE 1 = 0;

The INSERT INTO SELECT statement copies data from one table and inserts it into another table. The INSERT INTO SELECT statement requires that the data types in source and target tables match. The existing records in the target table are unaffected.

INSERT INTO table2 (column1, column2, column3, ...)  
SELECT column1, column2, column3, ...  
FROM table1  
WHERE condition;

The CASE expression goes through conditions and returns a value when the first condition is met (like an if-then-else statement). So, once a condition is true, it will stop reading and return the result. If no conditions are true, it returns the value in the ELSE clause. If there is no ELSE part and no conditions are true, it returns NULL.

CASE  
    WHEN condition1 THEN result1  
    WHEN condition2 THEN result2  
    WHEN conditionN THEN resultN  
    ELSE result  
END;

SELECT CustomerName, City, Country  
FROM Customers ORDER BY (CASE  
    WHEN City IS NULL THEN Country  
    ELSE City  
END);

[ISNULL()](https://www.w3schools.com/sql/func_sqlserver_isnull.asp) function lets you return an alternative value when an expression is NULL. COALESCE() Return the first non-null value in a list.

SELECT ProductName, UnitPrice \* (UnitsInStock + ISNULL(UnitsOnOrder, 0))  
FROM Products;

SELECT ProductName, UnitPrice \* (UnitsInStock + COALESCE(UnitsOnOrder, 0))  
FROM Products;

A stored procedure is a prepared SQL code that you can save, so the code can be reused over and over again.

CREATE PROCEDURE SelectAllCustomers @City nvarchar(30), @PostalCode nvarchar(10) AS  
SELECT \* FROM Customers WHERE City = @City AND PostalCode = @PostalCode  
GO;

EXEC SelectAllCustomers @City = 'London', @PostalCode = 'WA1 1DP';

VIEWS:

A view is a virtual table created according to the result set of an SQL statement. A view contains rows and columns, just like a real table. The columns in the view are the columns from one or more real tables in the database. SQL functions, WHERE, and JOIN statements can also be added to the view.

Databases 🡪 Views 🡪 New View

CREATE VIEW view\_name AS  
SELECT column1, column2, ...FROM table\_name WHERE condition;

SELECT \* FROM view\_name;

CREATE OR REPLACE VIEW view\_name AS SELECT ………………

DROP VIEW view\_name;

Sub String:

**SELECT** SUBSTRING('javatpoint', 1, 50) **AS** ExtractString;

**SELECT** person\_name, email,  SUBSTRING(email, LEN(Email) - 2, 3) **AS** EmailString

**FROM** persons;

**SELECT** person\_name, email,  SUBSTRING(email, CHARINDEX('@', email) + 1, LEN(email)) **AS** DomainName  **FROM** persons;

CTE:

**A CTE (Common Table Expression) is a one-time result set that only exists for the duration of the query**. It allows us to refer to data within a single SELECT, INSERT, UPDATE, DELETE, CREATE VIEW, or MERGE statement's execution scope. It is temporary because its result cannot be stored anywhere and will be lost as soon as a query's execution is completed

**WITH** customers\_in\_NewYork **AS**

(**SELECT** \* **FROM** customer **WHERE** state = 'New York')

**SELECT** c\_name, email, state **FROM** customers\_in\_NewYork;

**WITH**  customers\_in\_NewYork  **AS**

(**SELECT** \* **FROM** customer **WHERE** state 'NewYork'),

customers\_in\_California  **AS**

(**SELECT** \* **FROM** customer **WHERE** state = 'California')

**SELECT** c\_name, email, state **FROM** customers\_in\_NewYork

**UNION** ALL

**SELECT** c\_name, email, state **FROM** customers\_in\_California;

WITH avg\_order(total) AS (

SELECT AVG(total)

FROM orders

WHERE created\_at > '2016-04-21'

)

SELECT o.id,o.total

FROM orders AS o

LEFT JOIN avg\_order AS a

WHERE o.total > a.total

ORDER BY o.total DESC

Recursive CTE

**WITH**  odd\_num\_cte (id, n) **AS**

(    **SELECT** 1, 1

**UNION** ALL

**SELECT** id+1, n+2 **from** odd\_num\_cte **where** id < 5     )

**SELECT** \* **FROM** odd\_num\_cte;

Cursor:

A cursor in SQL Server is a d**atabase object that allows us to retrieve each row at a time and manipulate its data**. A cursor is nothing more than a pointer to a row. It's always used in conjunction with a SELECT statement. **@@FETCHSTATUS function** get the status of the most recent FETCH statement that was executed against the cursor.

**DECLARE** cursor\_name **CURSOR**

**FOR** select\_statement;

**OPEN** cursor\_name;

WHILE @@FETCH\_STATUS = 0

**BEGIN**

**FETCH** **NEXT** **FROM** cursor\_name;

**END**;

**CLOSE** cursor\_name;

**DEALLOCATE** cursor\_name;

The result set shown by the static cursor is always the same as when the cursor was first opened. Since the static cursor will store the result in **tempdb**, they are always **read-only**.

Dynamic cursors allow us to perform the data updation, deletion, and insertion operations while the cursor is open. It is **scrollable by default**.

It is the default and fastest cursor type among all cursors. It is called a forward-only cursor because it **moves only forward through the result set**. It can only retrieve rows from the beginning to the end of the result set.

Index:

An index is one of the important paths to make the performance of SQL Server database high. It **makes the** **querying process fast** by providing easy access to rows in data table. **An index is a set of keys made up of single or multiple columns in a table or view**. They are stored in a structure (B-tree) that helps SQL Server users quickly and efficiently find the rows or rows associated with the key values.

There are four types of logins provided by SQL Server:

* A login based on Windows credentials.
* A login specific to SQL Server.
* A login mapped to a certificate.
* A login mapped to asymmetric key.

create login yourloginname with password='yourpassword'

Create user <name> for login <login name>

Grant <permission name> on <object Name>  to <user name>

CREATE DATABASE *databasename*;

SHOW DATABASES;

USE *databasename;*

DROP DATABASE *databasename*;

BACKUP DATABASE *databasename* TO DISK = '*filepath*';

BACKUP LOG *databasename* TO DISK = '*filepath*';

BACKUP DATABASE *databasename* TO DISK = '*filepath*' WITH DIFFERENTIAL;

A differential back up only backs up the parts of the database that have changed since the last full database backup.

CREATE TABLE *table\_name*(*column1 datatype*, *column2 datatype*,....);

CREATE TABLE *new\_table\_name* AS SELECT *column1, column2,...*

FROM *existing\_table\_name* WHERE ....;

The new table gets the same column definitions. All columns or specific columns can be selected.

DROP TABLE *table\_name*;

TRUNCATE TABLE *table\_name*;

This statement is used to delete the data inside a table, but not the table.

The ALTER TABLE statement is used to add, delete, or modify columns in an existing table. The ALTER TABLE statement is also used to add and drop various constraints on an existing table.

ALTER TABLE *table\_name* ADD *column\_name datatype*;

ALTER TABLE *table\_name* DROP COLUMN *column\_name*;

ALTER TABLE *table\_name* RENAME COLUMN *old\_name* to *new\_name*;

ALTER TABLE *table\_name* ALTER COLUMN *column\_name datatype*;

# SQL constraints

SQL constraints are used to specify rules for the data in a table. Constraints are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the table. If there is any violation between the constraint and the data action, the action is aborted. Constraints can be column level or table level. Column level constraints apply to a column, and table level constraints apply to the whole table.

The following constraints are commonly used in SQL:

* [NOT NULL](https://www.w3schools.com/SQL/sql_notnull.asp) - Ensures that a column cannot have a NULL value
* [UNIQUE](https://www.w3schools.com/SQL/sql_unique.asp) - Ensures that all values in a column are different
* [PRIMARY KEY](https://www.w3schools.com/SQL/sql_primarykey.asp) - A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table
* [FOREIGN KEY](https://www.w3schools.com/SQL/sql_foreignkey.asp) - Prevents actions that would destroy links between tables
* [CHECK](https://www.w3schools.com/SQL/sql_check.asp) - Ensures that the values in a column satisfies a specific condition
* [DEFAULT](https://www.w3schools.com/SQL/sql_default.asp) - Sets a default value for a column if no value is specified
* [CREATE INDEX](https://www.w3schools.com/SQL/sql_create_index.asp) - Used to create and retrieve data from the database very quickly

Constraints can be specified when the table is created with the CREATE TABLE statement, or after the table is created with the ALTER TABLE statement.

CREATE TABLE *table\_name*(*column1 datatype* *constraint*,   
*column2 datatype* *constraint*, *column3 datatype* *constraint*, ....);

ALTER TABLE *table\_name* ALTER COLUMN *column\_name datatype constraint*;

By default, a column can hold NULL values. The NOT NULL constraint enforces a column to NOT accept NULL values.

The UNIQUE constraint ensures that all values in a column are different. Both the UNIQUE and PRIMARY KEY constraints provide a guarantee for uniqueness for a column or set of columns. A PRIMARY KEY constraint automatically has a UNIQUE constraint. However, you can have many UNIQUE constraints per table, but only one PRIMARY KEY constraint per table.

CREATE TABLE Persons (ID int NOT NULL UNIQUE,……);

CREATE TABLE Persons (ID int NOT NULL, LastName varchar(255) NOT NULL,   
    …………, CONSTRAINT UC\_Person UNIQUE (ID,LastName) );

ALTER TABLE Persons ADD UNIQUE (ID)

ALTER TABLE Persons ADD CONSTRAINT UC\_Person UNIQUE (ID,LastName);

ALTER TABLE Persons DROP CONSTRAINT UC\_Person;

The PRIMARY KEY constraint uniquely identifies each record in a table. Primary keys must contain UNIQUE values, and cannot contain NULL values. A table can have only ONE primary key; and in the table, this primary key can consist of single or multiple columns (fields).

NOTE – Same example PRIMARY KEY instead of UNIQUE

The FOREIGN KEY constraint is used to prevent actions that would destroy links between tables. A FOREIGN KEY is a field (or collection of fields) in one table, that refers to the [PRIMARY KEY](https://www.w3schools.com/SQL/sql_primarykey.asp) in another table. The table with the foreign key is called the child table, and the table with the primary key is called the referenced or parent table. The FOREIGN KEY constraint prevents invalid data from being inserted into the foreign key column, because it has to be one of the values contained in the parent table.

CREATE TABLE Orders (   
    OrderID int NOT NULL PRIMARY KEY,   
    OrderNumber int NOT NULL,   
    PersonID int FOREIGN KEY REFERENCES Persons(PersonID) );

ALTER TABLE Orders   
ADD FOREIGN KEY (PersonID) REFERENCES Persons(PersonID);

ALTER TABLE Orders   
ADD CONSTRAINT FK\_PersonOrder   
FOREIGN KEY (PersonID) REFERENCES Persons(PersonID);

The CHECK constraint is used to limit the value range that can be placed in a column. If you define a CHECK constraint on a column it will allow only certain values for this column. If you define a CHECK constraint on a table it can limit the values in certain columns based on values in other columns in the row.

CREATE TABLE Persons (ID int NOT NULL,……,Age int CHECK (Age>=18));

CREATE TABLE Persons (ID int NOT NULL,……,Age int,City varchar(255),   
    CONSTRAINT CHK\_Person CHECK (Age>=18 AND City='Sandnes'));

The DEFAULT constraint is used to set a default value for a column. The default value will be added to all new records, if no other value is specified.

CREATE TABLE Orders (ID int NOT NULL, OrderNumber int NOT NULL,   
    OrderDate date DEFAULT GETDATE());

ALTER TABLE Persons ADD CONSTRAINT df\_City DEFAULT 'Sandnes' FOR City;

ALTER TABLE Persons ALTER COLUMN City DROP DEFAULT;

Indexes are used to retrieve data from the database more quickly than otherwise. The users cannot see the indexes, they are just used to speed up searches/queries.

CREATE INDEX index\_name ON table\_name (column1, column2, ...);

DROP INDEX table\_name.index\_name;

Auto-increment allows a unique number to be generated automatically when a new record is inserted into a table. Often this is the primary key field that we would like to be created automatically every time a new record is inserted.

CREATE TABLE Persons (Personid int IDENTITY(1,1) PRIMARY KEY,  
    LastName varchar(255) NOT NULL, FirstName varchar(255), Age int);

By default, the starting value for AUTO\_INCREMENT is 1, and it will increment by 1 for each new record. To let the AUTO\_INCREMENT sequence start with another value, use the following SQL statement:

To specify that the "Personid" column should start at value 10 and increment by 5, change it to IDENTITY(10,5).

Data types for storing a date or a date/time value in the database:

* DATE - format YYYY-MM-DD
* DATETIME - format: YYYY-MM-DD HH:MI:SS
* SMALLDATETIME - format: YYYY-MM-DD HH:MI:SS
* TIMESTAMP - format: a unique number

# Aggregate Functions:

Aggregate functions operate on sets of values. They are often used with a GROUP BY clause to group values into subsets.

Functions: COUNT, COUNT(DISTINCT), AVG, MIN, MAX, SUM, STD ,BIT\_AND - Return bitwise AND, BIT\_OR - Return bitwise OR, BIT\_XOR - Return bitwise XOR, GROUP\_CONCAT - Return a concatenated string

If you use an aggregate function in a statement containing no GROUP BY clause, it is equivalent to grouping on all rows

The [SUM()](https://dev.mysql.com/doc/refman/8.0/en/aggregate-functions.html#function_sum) and [AVG()](https://dev.mysql.com/doc/refman/8.0/en/aggregate-functions.html#function_avg) aggregate functions do not work with temporal (date time) values. The work around this problem:

SELECT SEC\_TO\_TIME(SUM(TIME\_TO\_SEC(*time\_col*))) FROM *tbl\_name*;

SELECT FROM\_DAYS(SUM(TO\_DAYS(*date\_col*))) FROM *tbl\_name*;

AVG([DISTINCT] ***expr***)

Returns the average value of ***expr***. The DISTINCT option can be used to return the average of the distinct values of ***expr***.

If there are no matching rows, [AVG()](https://dev.mysql.com/doc/refman/8.0/en/aggregate-functions.html#function_avg) returns NULL. Returns NULL if ***expr*** is NULL.

SELECT student\_name, AVG(score)

FROM student

GROUP BY student\_name;

[COUNT([DISTINCT] ***expr***)](https://dev.mysql.com/doc/refman/8.0/en/aggregate-functions.html#function_count)

Returns a count of the number of non-NULL values of ***expr*** in the rows retrieved by a [SELECT](https://dev.mysql.com/doc/refman/8.0/en/select.html) statement. The result is a [BIGINT](https://dev.mysql.com/doc/refman/8.0/en/integer-types.html) value.

SELECT student.student\_name,COUNT(\*)

FROM student,course

WHERE student.student\_id=course.student\_id

GROUP BY student\_name;

[COUNT(\*)](https://dev.mysql.com/doc/refman/8.0/en/aggregate-functions.html#function_count) is somewhat different in that it returns a count of the number of rows retrieved, whether or not they contain NULL values.

InnoDB handles SELECT COUNT(\*) and SELECT COUNT(1) operations in the same way. There is no performance difference.

For MyISAM tables, [COUNT(\*)](https://dev.mysql.com/doc/refman/8.0/en/aggregate-functions.html#function_count) is optimized to return very quickly if the [SELECT](https://dev.mysql.com/doc/refman/8.0/en/select.html) retrieves from one table, no other columns are retrieved, and there is no WHERE clause. For example:

This optimization only applies to MyISAM tables, because an exact row count is stored for this storage engine and can be accessed very quickly. COUNT(1) is only subject to the same optimization if the first column is defined as NOT NULL

[GROUP\_CONCAT(***expr***)](https://dev.mysql.com/doc/refman/8.0/en/aggregate-functions.html#function_group-concat)

This function returns a string result with the concatenated non-NULL values from a group. It returns NULL if there are no non-NULL values. The full syntax is as follows:

GROUP\_CONCAT([DISTINCT] expr [,expr ...]

[ORDER BY {unsigned\_integer | col\_name | expr}

[ASC | DESC] [,col\_name ...]]

[SEPARATOR str\_val])

mysql> SELECT student\_name,

GROUP\_CONCAT(DISTINCT score

ORDER BY score DESC SEPARATOR ' ')

FROM student

GROUP BY student\_name;

[MAX([DISTINCT] ***expr***) [***over\_clause***]](https://dev.mysql.com/doc/refman/8.0/en/aggregate-functions.html#function_max)

Returns the maximum value of ***expr***. [MAX()](https://dev.mysql.com/doc/refman/8.0/en/aggregate-functions.html#function_max) may take a string argument; in such cases, it returns the maximum string value. The DISTINCT keyword can be used to find the maximum of the distinct values of ***expr***, however, this produces the same result as omitting DISTINCT.

If there are no matching rows, or if ***expr*** is NULL, [MAX()](https://dev.mysql.com/doc/refman/8.0/en/aggregate-functions.html#function_max) returns NULL.

mysql> SELECT student\_name, MIN(score), MAX(score), SUM(score)

FROM student

GROUP BY student\_name;

[MIN([DISTINCT] ***expr***) [***over\_clause***]](https://dev.mysql.com/doc/refman/8.0/en/aggregate-functions.html#function_min)

Returns the minimum value of ***expr***. [MIN()](https://dev.mysql.com/doc/refman/8.0/en/aggregate-functions.html#function_min) may take a string argument; in such cases, it returns the minimum string value The DISTINCT keyword can be used to find the minimum of the distinct values of ***expr***, however, this produces the same result as omitting DISTINCT.

If there are no matching rows, or if ***expr*** is NULL, [MIN()](https://dev.mysql.com/doc/refman/8.0/en/aggregate-functions.html#function_min) returns NULL.

[SUM([DISTINCT] ***expr***) [***over\_clause***]](https://dev.mysql.com/doc/refman/8.0/en/aggregate-functions.html#function_sum)

Returns the sum of ***expr***. If the return set has no rows, [SUM()](https://dev.mysql.com/doc/refman/8.0/en/aggregate-functions.html#function_sum) returns NULL. The DISTINCT keyword can be used to sum only the distinct values of ***expr***.

If there are no matching rows, or if ***expr*** is NULL, [SUM()](https://dev.mysql.com/doc/refman/8.0/en/aggregate-functions.html#function_sum) returns NULL.

Super aggregate:

SELECT year, SUM(profit) AS profit

FROM sales

GROUP BY year;

+------+--------+

| year | profit |

+------+--------+

| 2000 | 4525 |

| 2001 | 3010 |

+------+--------+

The output shows the total profit for each year. To also determine the total profit summed over all years, you must add up the individual values yourself or run an additional query. Or you can use ROLLUP, which provides both levels of analysis with a single query. Adding a WITH ROLLUP modifier to the GROUP BY clause causes the query to produce another (super-aggregate) row that shows the grand total over all year values:

mysql> SELECT year, SUM(profit) AS profit

FROM sales

GROUP BY year WITH ROLLUP;

+------+--------+

| year | profit |

+------+--------+

| 2000 | 4525 |

| 2001 | 3010 |

| NULL | 7535 |

+------+--------+

The NULL value in the year column identifies the grand total super-aggregate line.

# Window functions

[ROW\_NUMBER()](https://dev.mysql.com/doc/refman/8.0/en/window-function-descriptions.html#function_row-number) ***over\_clause***

Returns the number of the current row within its partition. Rows numbers range from 1 to the number of partition rows. ORDER BY affects the order in which rows are numbered. Without ORDER BY, row numbering is nondeterministic.

[RANK()](https://dev.mysql.com/doc/refman/8.0/en/window-function-descriptions.html#function_rank) ***over\_clause***

Returns the rank of the current row within its partition, with gaps. Peers are considered ties and receive the same rank. This function does not assign consecutive ranks to peer groups if groups of size greater than one exist; the result is noncontiguous rank numbers. This function should be used with ORDER BY to sort partition rows into the desired order. Without ORDER BY, all rows are peers.

[DENSE\_RANK()](https://dev.mysql.com/doc/refman/8.0/en/window-function-descriptions.html#function_dense-rank) ***over\_clause***

Returns the rank of the current row within its partition, without gaps. Peers are considered ties and receive the same rank. This function assigns consecutive ranks to peer groups; the result is that groups of size greater than one do not produce noncontiguous rank numbers. This function should be used with ORDER BY to sort partition rows into the desired order. Without ORDER BY, all rows are peers.

SELECT

val,

ROW\_NUMBER() OVER w AS 'row\_number',

RANK() OVER w AS 'rank',

DENSE\_RANK() OVER w AS 'dense\_rank'

FROM numbers

WINDOW w AS (ORDER BY val);

+------+------------+------+------------+

| val | row\_number | rank | dense\_rank |

+------+------------+------+------------+

| 1 | 1 | 1 | 1 |

| 1 | 2 | 1 | 1 |

| 2 | 3 | 3 | 2 |

| 3 | 4 | 4 | 3 |

| 3 | 5 | 4 | 3 |

| 3 | 6 | 4 | 3 |

| 4 | 7 | 7 | 4 |

| 4 | 8 | 7 | 4 |

| 5 | 9 | 9 | 5 |

+------+------------+------+------------+

SELECT

ROW\_NUMBER() OVER(ORDER BY name ASC) AS Row#,

name, recovery\_model\_desc

FROM sys.databases

WHERE database\_id < 5;

SELECT

ROW\_NUMBER() OVER(PARTITION BY recovery\_model ORDER BY name ASC)

AS Row#,

name, recovery\_model\_desc

FROM sys.databases WHERE database\_id < 5;

[PERCENT\_RANK()](https://dev.mysql.com/doc/refman/8.0/en/window-function-descriptions.html#function_percent-rank) ***over\_clause***

Returns the percentage of partition values less than the value in the current row, excluding the highest value. Return values range from 0 to 1 and represent the row relative rank, calculated as the result of this formula, where ***rank*** is the row rank and ***rows*** is the number of partition rows:

(rank - 1) / (rows - 1)

[CUME\_DIST()](https://dev.mysql.com/doc/refman/8.0/en/window-function-descriptions.html#function_cume-dist) ***over\_clause***

Returns the cumulative distribution of a value within a group of values; that is, the percentage of partition values less than or equal to the value in the current row. Return values range from 0 to 1. This function should be used with ORDER BY to sort partition rows into the desired order. Without ORDER BY, all rows are peers and have value ***N***/***N*** = 1, where ***N*** is the partition size.

SELECT

val,

ROW\_NUMBER() OVER w AS 'row\_number',

CUME\_DIST() OVER w AS 'cume\_dist',

PERCENT\_RANK() OVER w AS 'percent\_rank'

FROM numbers

WINDOW w AS (ORDER BY val);

+------+------------+--------------------+--------------+

| val | row\_number | cume\_dist | percent\_rank |

+------+------------+--------------------+--------------+

| 1 | 1 | 0.2222222222222222 | 0 |

| 1 | 2 | 0.2222222222222222 | 0 |

| 2 | 3 | 0.3333333333333333 | 0.25 |

| 3 | 4 | 0.6666666666666666 | 0.375 |

| 3 | 5 | 0.6666666666666666 | 0.375 |

| 3 | 6 | 0.6666666666666666 | 0.375 |

| 4 | 7 | 0.8888888888888888 | 0.75 |

| 4 | 8 | 0.8888888888888888 | 0.75 |

| 5 | 9 | 1 | 1 |

+------+------------+--------------------+--------------+

ROW\_NUMBER ( )

OVER ( [ PARTITION BY value\_expression , ... [ n ] ] order\_by\_clause )

SELECT

ROW\_NUMBER() OVER(PARTITION BY recovery\_model\_desc ORDER BY name ASC)

AS Row#,

name, recovery\_model\_desc

FROM sys.databases WHERE database\_id < 5;

SELECT

ROW\_NUMBER() OVER(ORDER BY name ASC) AS Row#,

name, recovery\_model\_desc

FROM sys.databases

WHERE database\_id < 5;

RANK() OVER (PARTITION BY ContenderNum ORDER BY totals ASC) AS xRank

RANK() OVER (ORDER BY totals DESC) AS xRank

SELECT TOP(10) EmpId, Salary,

DENSE\_RANK() OVER (ORDER BY Salary DESC) AS RankBySalary

FROM EmployeePayHistory;

[FIRST\_VALUE(***expr***)](https://dev.mysql.com/doc/refman/8.0/en/window-function-descriptions.html#function_first-value) [***null\_treatment***] ***over\_clause***

Returns the value of ***expr*** from the first row of the window frame.

[LAST\_VALUE(***expr***)](https://dev.mysql.com/doc/refman/8.0/en/window-function-descriptions.html#function_last-value) [***null\_treatment***] ***over\_clause***

Returns the value of ***expr*** from the last row of the window frame.

[NTH\_VALUE(***expr***, ***N***)](https://dev.mysql.com/doc/refman/8.0/en/window-function-descriptions.html#function_nth-value) [***from\_first\_last***] [***null\_treatment***] ***over\_clause***

Returns the value of ***expr*** from the ***N***-th row of the window frame. If there is no such row, the return value is NULL. ***N*** must be a literal positive integer. ***from\_first\_last*** is part of the SQL standard, but the MySQL implementation permits only FROM FIRST (which is also the default). FROM LAST is parsed, but produces an error. To obtain the same effect as FROM LAST (begin calculations at the last row of the window), use ORDER BY to sort in reverse order.

SELECT

time, subject, val,

FIRST\_VALUE(val) OVER w AS 'first',

LAST\_VALUE(val) OVER w AS 'last',

NTH\_VALUE(val, 2) OVER w AS 'second',

NTH\_VALUE(val, 4) OVER w AS 'fourth'

FROM observations

WINDOW w AS (PARTITION BY subject ORDER BY time

ROWS UNBOUNDED PRECEDING);

+----------+---------+------+-------+------+--------+--------+

| time | subject | val | first | last | second | fourth |

+----------+---------+------+-------+------+--------+--------+

| 07:00:00 | st113 | 10 | 10 | 10 | NULL | NULL |

| 07:15:00 | st113 | 9 | 10 | 9 | 9 | NULL |

| 07:30:00 | st113 | 25 | 10 | 25 | 9 | NULL |

| 07:45:00 | st113 | 20 | 10 | 20 | 9 | 20 |

| 07:00:00 | xh458 | 0 | 0 | 0 | NULL | NULL |

| 07:15:00 | xh458 | 10 | 0 | 10 | 10 | NULL |

| 07:30:00 | xh458 | 5 | 0 | 5 | 10 | NULL |

| 07:45:00 | xh458 | 30 | 0 | 30 | 10 | 30 |

| 08:00:00 | xh458 | 25 | 0 | 25 | 10 | 30 |

+----------+---------+------+-------+------+--------+--------+

[LAG(***expr*** [, ***N***[, ***default***]])](https://dev.mysql.com/doc/refman/8.0/en/window-function-descriptions.html#function_lag) [***null\_treatment***] ***over\_clause***

Returns the value of ***expr*** from the row that lags (precedes) the current row by ***N*** rows within its partition. If there is no such row, the return value is ***default***. For example, if ***N*** is 3, the return value is ***default*** for the first three rows. If ***N*** or ***default*** are missing, the defaults are 1 and NULL, respectively.

[LEAD(***expr*** [, ***N***[, ***default***]])](https://dev.mysql.com/doc/refman/8.0/en/window-function-descriptions.html#function_lead) [***null\_treatment***] ***over\_clause***

Returns the value of ***expr*** from the row that leads (follows) the current row by ***N*** rows within its partition. If there is no such row, the return value is ***default***. For example, if ***N*** is 3, the return value is ***default*** for the last three rows. If ***N*** or ***default*** are missing, the defaults are 1 and NULL, respectively.

SELECT

t, val,

LAG(val) OVER w AS 'lag',

LEAD(val) OVER w AS 'lead',

val - LAG(val) OVER w AS 'lag diff',

val - LEAD(val) OVER w AS 'lead diff'

FROM series

WINDOW w AS (ORDER BY t);

+----------+------+------+------+----------+-----------+

| t | val | lag | lead | lag diff | lead diff |

+----------+------+------+------+----------+-----------+

| 12:00:00 | 100 | NULL | 125 | NULL | -25 |

| 13:00:00 | 125 | 100 | 132 | 25 | -7 |

| 14:00:00 | 132 | 125 | 145 | 7 | -13 |

| 15:00:00 | 145 | 132 | 140 | 13 | 5 |

| 16:00:00 | 140 | 145 | 150 | -5 | -10 |

| 17:00:00 | 150 | 140 | 200 | 10 | -50 |

| 18:00:00 | 200 | 150 | NULL | 50 | NULL |

+----------+------+------+------+----------+-----------+

[NTILE(***N***)](https://dev.mysql.com/doc/refman/8.0/en/window-function-descriptions.html#function_ntile) ***over\_clause***

Divides a partition into ***N*** groups (buckets), assigns each row in the partition its bucket number, and returns the bucket number of the current row within its partition. For example, if ***N*** is 4, NTILE() divides rows into four buckets. If ***N*** is 100, NTILE() divides rows into 100 buckets.

***N*** must be a literal positive integer. Bucket number return values range from 1 to ***N***.

SELECT

val,

ROW\_NUMBER() OVER w AS 'row\_number',

NTILE(2) OVER w AS 'ntile2',

NTILE(4) OVER w AS 'ntile4'

FROM numbers

WINDOW w AS (ORDER BY val);

+------+------------+--------+--------+

| val | row\_number | ntile2 | ntile4 |

+------+------------+--------+--------+

| 1 | 1 | 1 | 1 |

| 1 | 2 | 1 | 1 |

| 2 | 3 | 1 | 1 |

| 3 | 4 | 1 | 2 |

| 3 | 5 | 1 | 2 |

| 3 | 6 | 2 | 3 |

| 4 | 7 | 2 | 3 |

| 4 | 8 | 2 | 4 |

| 5 | 9 | 2 | 4 |

+------+------------+--------+--------+

SELECT SUM(profit) AS total\_profit

FROM sales;

+--------------+

| total\_profit |

+--------------+

| 7535 |

+--------------+

mysql> SELECT country, SUM(profit) AS country\_profit

FROM sales

GROUP BY country

ORDER BY country;

+---------+----------------+

| country | country\_profit |

+---------+----------------+

| Finland | 1610 |

| India | 1350 |

| USA | 4575 |

+---------+----------------+

mysql> SELECT

year, country, product, profit,

SUM(profit) OVER() AS total\_profit,

SUM(profit) OVER(PARTITION BY country) AS country\_profit

FROM sales

ORDER BY country, year, product, profit;

+------+---------+------------+--------+--------------+----------------+

| year | country | product | profit | total\_profit | country\_profit |

+------+---------+------------+--------+--------------+----------------+

| 2000 | Finland | Computer | 1500 | 7535 | 1610 |

| 2000 | Finland | Phone | 100 | 7535 | 1610 |

| 2001 | Finland | Phone | 10 | 7535 | 1610 |

| 2000 | India | Calculator | 75 | 7535 | 1350 |

| 2000 | India | Calculator | 75 | 7535 | 1350 |

| 2000 | India | Computer | 1200 | 7535 | 1350 |

| 2000 | USA | Calculator | 75 | 7535 | 4575 |

| 2000 | USA | Computer | 1500 | 7535 | 4575 |

| 2001 | USA | Calculator | 50 | 7535 | 4575 |

| 2001 | USA | Computer | 1200 | 7535 | 4575 |

| 2001 | USA | Computer | 1500 | 7535 | 4575 |

| 2001 | USA | TV | 100 | 7535 | 4575 |

| 2001 | USA | TV | 150 | 7535 | 4575 |

+------+---------+------------+--------+--------------+----------------+

# Frames:

The definition of a window used with a window function can include a frame clause. A frame is a subset of the current partition and the frame clause specifies how to define the subset.

Frames are determined with respect to the current row, which enables a frame to move within a partition depending on the location of the current row within its partition. Examples:

* By defining a frame to be all rows from the partition start to the current row, you can compute running totals for each row.
* By defining a frame as extending ***N*** rows on either side of the current row, you can compute rolling averages.

The ROW clause does it by specifying a fixed number of rows that precede or follow the current row.

The RANGE clause, on the other hand, limits the rows logically; it specifies the range of values in relation to the value of the current row.

The following query demonstrates the use of moving frames to compute running totals within each group of time-ordered level values, as well as rolling averages computed from the current row and the rows that immediately precede and follow it:

frame\_clause:

frame\_units frame\_extent

frame\_units: The ***frame\_units*** value indicates the type of relationship between the current row and frame rows:

{ROWS | RANGE}

* ROWS: The frame is defined by beginning and ending row positions. Offsets are differences in row numbers from the current row number.
* RANGE: The frame is defined by rows within a value range. Offsets are differences in row values from the current row value.

frame\_extent: The ***frame\_extent*** value indicates the start and end points of the frame. You can specify just the start of the frame (in which case the current row is implicitly the end) or use BETWEEN to specify both frame endpoints:

{frame\_start | BETWEEN frame\_start AND frame\_end }

Finally…. 🡪 {ROWS | RANGE} {frame\_start | BETWEEN frame\_start AND frame\_end }

frame\_start, frame\_end: {

CURRENT ROW

| UNBOUNDED PRECEDING

| UNBOUNDED FOLLOWING

| expr PRECEDING

| expr FOLLOWING

}

With BETWEEN syntax, ***frame\_start*** must not occur later than ***frame\_end***.

The permitted ***frame\_start*** and ***frame\_end*** values have these meanings:

* CURRENT ROW: For ROWS, the bound is the current row. For RANGE, the bound is the peers of the current row.
* UNBOUNDED PRECEDING: The bound is the first partition row.
* UNBOUNDED FOLLOWING: The bound is the last partition row.
* ***expr*** PRECEDING: For ROWS, the bound is ***expr*** rows before the current row. For RANGE, the bound is the rows with values equal to the current row value minus ***expr***; if the current row value is NULL, the bound is the peers of the row.
* ***expr*** FOLLOWING: For ROWS, the bound is ***expr*** rows after the current row. For RANGE, the bound is the rows with values equal to the current row value plus ***expr***; if the current row value is NULL, the bound is the peers of the row.

SELECT

time, subject, val,

SUM(val) OVER (PARTITION BY subject ORDER BY time

ROWS UNBOUNDED PRECEDING)

AS running\_total,

AVG(val) OVER (PARTITION BY subject ORDER BY time

ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)

AS rolling\_average

FROM observations;

+----------+---------+------+---------------+-----------------+

| time | subject | val | running\_total | rolling\_average |

+----------+---------+------+---------------+-----------------+

| 07:00:00 | st113 | 10 | 10 | 9.5000 |

| 07:15:00 | st113 | 9 | 19 | 14.6667 |

| 07:30:00 | st113 | 25 | 44 | 18.0000 |

| 07:45:00 | st113 | 20 | 64 | 22.5000 |

| 07:00:00 | xh458 | 0 | 0 | 5.0000 |

| 07:15:00 | xh458 | 10 | 10 | 5.0000 |

| 07:30:00 | xh458 | 5 | 15 | 15.0000 |

| 07:45:00 | xh458 | 30 | 45 | 20.0000 |

| 08:00:00 | xh458 | 25 | 70 | 27.5000 |

+----------+---------+------+---------------+-----------------+

| **id** | **date** | **revenue\_amount** | **running\_total** |
| --- | --- | --- | --- |
| 1 | 2021-05-01 | 12,573.25 | 12,573.25 |
| 2 | 2021-05-02 | 11,348.22 | 23,921.47 |
| 3 | 2021-05-02 | 14,895.13 | 38,816.60 |
| 4 | 2021-05-03 | 14,388.14 | 53,204.74 |
| 5 | 2021-05-04 | 18,847.54 | 72,052.28 |

Notice there is another row with a different amount of revenue for 2021-05-02. Maybe this is for another branch, country, product or whatever. It works the same way: 14,895.13 + 23,921.47 = 38,816.60. (The RANGE clause will work even if there are multiple rows with the same date.)

SELECT

  id,

  date,

  revenue\_amount,

  SUM(revenue\_amount) OVER (

    ORDER BY date

    RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

  ) AS running\_total

FROM revenue;

SELECT

  shop,

  date,

  revenue\_amount,

  AVG(revenue\_amount) OVER (

    PARTITION BY shop

    ORDER BY date ASC

    RANGE BETWEEN INTERVAL '1' DAY PRECEDING AND CURRENT ROW

  ) AS moving\_avg

FROM revenue\_per\_shop;

# Database Architecture:

# Normal Forms

Functional Dependency:

The functional dependency is a relationship that exists between two attributes. It typically exists between the primary key and non-key attribute within a table.

Assume we have an employee table with attributes: Emp\_Id, Emp\_Name, Emp\_Address.

Emp\_Id → Emp\_Name

Here Emp\_Id attribute can uniquely identify the Emp\_Name attribute of employee table because if we know the Emp\_Id, we can tell that employee name associated with it.

The left side of FD is known as a determinant, the right side of the production is known as a dependent.

* A → B has trivial functional dependency if B is a subset of A.

{Employee\_id, Employee\_Name}   →    Employee\_Id

* A → B has a non-trivial functional dependency if B is not a subset of A.

Employee\_id  →    Employee\_Name

Rules:

In the reflexive rule, if Y is a subset of X, then X determines Y.

If X ⊇ Y then X  →    Y

The augmentation is also called as a partial dependency. In augmentation rule, if X determines Y, then XZ determines YZ for any Z.

If X    →  Y then XZ   →   YZ

In the transitive rule, if X determines Y and Y determine Z, then X must also determine Z.

If X   →   Y and Y  →  Z then X  →   Z

In union rule, if X determines Y and X determines Z, then X must also determine Y and Z.

If X    →  Y and X   →  Z then X  →    YZ

Decomposition rule is also known as project rule. It is the reverse of union rule.

This Rule says, if X determines Y and Z, then X determines Y and X determines Z separately.

If X   →   YZ then X   →   Y and X  →    Z

Normalization

* Normalization is the process of organizing the data in the database.
* Normalization is used to minimize the redundancy from a relation or set of relations.

1NF

* A relation will be 1NF if it contains an atomic value.
* It states that an attribute of a table cannot hold multiple values. It must hold only single-valued attribute.
* ID Name Courses  
  ------------------  
  1 A c1, c2  
  2 E c3  
  3 M C2, c3

ID Name Course  
------------------  
1 A c1  
1 A c2  
2 E c3  
3 M c2  
3 M c3

2NF

* In the 2NF, relational must be in 1NF.
* In the second normal form, all non-key attributes are fully functional dependent on the primary key
* STUD\_NO COURSE\_NO COURSE\_FEE  
  1 C1 1000  
  2 C2 1500  
  1 C4 2000  
  4 C3 1000  
  4 C1 1000  
  2 C5 2000

**Table 1** Table 2  
STUD\_NO COURSE\_NO COURSE\_NO COURSE\_FEE   
1 C1 C1 1000  
2 C2 C2 1500  
1 C4 C3 1000  
4 C3 C4 2000  
4 C1 C5 2000

3NF

* A relation will be in 3NF if it is in 2NF and not contain any transitive partial dependency.
* 3NF is used to reduce the data duplication. It is also used to achieve the data integrity.

STUDENTDATA (STUD\_NO, STUD\_NAME, STUD\_PHONE, STUD\_STATE, STUD\_ COUNTRY, STUD\_AGE)

STUD\_NO 🡪 STUD\_STATE, STUD\_STATE 🡪 STUD\_ COUNTRY

STUDENT (STUD\_NO, STUD\_NAME, STUD\_PHONE, STUD\_STATE, STUD\_AGE)   
STATE\_COUNTRY (STATE, COUNTRY)

 BCNF

* BCNF is the advance version of 3NF. It is stricter than 3NF.
* A table is in BCNF if every functional dependency X → Y, X is the super key of the table.

STUDENTDATA (STUD\_NO, STUD\_NAME, BRANCH\_NO, BRANCH\_NAME)

The table is not in BCNF because neither STUD\_NO nor BRANCH\_NO alone are keys.

To convert the given table into BCNF, we decompose it into three tables:

STUDENT (STUD\_NO, STUD\_NAME)

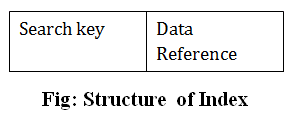
BRANCH (BRANCH\_NO, BRANCH\_NAME)

STUDENT\_BRANCH (STUD\_NO, BRANCH\_NO)

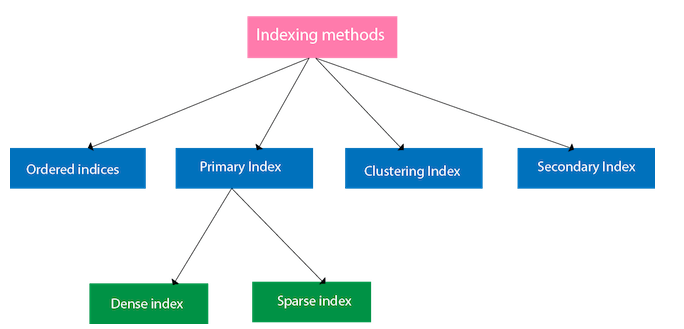
# Indexing:

* Indexing is used to optimize the performance of a database by minimizing the number of disk accesses required when a query is processed.
* The index is a type of data structure. It is used to locate and access the data in a database table quickly.

Indexes can be created using some database columns.



* The first column of the database is the search key that contains a copy of the primary key or candidate key of the table. The values of the primary key are stored in sorted order so that the corresponding data can be accessed easily.
* The second column of the database is the data reference. It contains a set of pointers holding the address of the disk block where the value of the particular key can be found.



## **Ordered indices**

The indices are usually sorted to make searching faster. The indices which are sorted are known as ordered indices.

## **Primary Index**

* If the index is created on the basis of the primary key of the table, then it is known as primary indexing. These primary keys are unique to each record and contain 1:1 relation between the records.
* As primary keys are stored in sorted order, the performance of the searching operation is quite efficient.

## **Dense index**

* The dense index contains an index record for every search key value in the data file. It makes searching faster.
* In this, the number of records in the index table is same as the number of records in the main table.
* It needs more space to store index record itself. The index records have the search key and a pointer to the actual record on the disk.

## **Sparse index**

* In the data file, index record appears only for a few items. Each item points to a block.
* In this, instead of pointing to each record in the main table, the index points to the records in the main table in a gap.

## **Clustering Index**

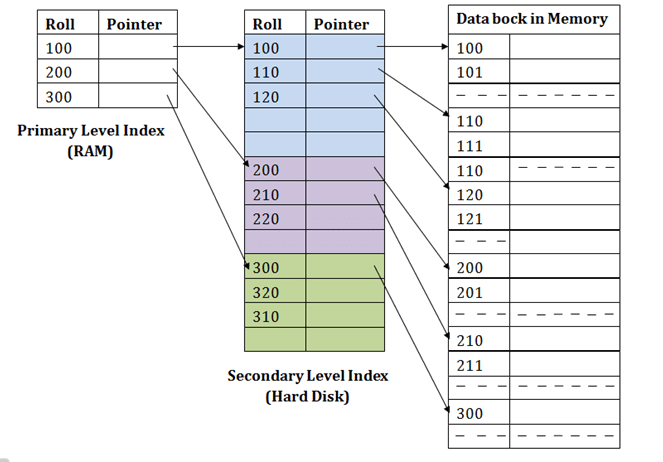
* A clustered index can be defined as an ordered data file. Sometimes the index is created on non-primary key columns which may not be unique for each record.
* In this case, to identify the record faster, we will group two or more columns to get the unique value and create index out of them. This method is called a clustering index.
* The records which have similar characteristics are grouped, and indexes are created for these group.

**Example**: suppose a company contains several employees in each department. Suppose we use a clustering index, where all employees which belong to the same Dept\_ID are considered within a single cluster, and index pointers point to the cluster as a whole. Here Dept\_Id is a non-unique key.

## **Secondary Index**

In the sparse indexing, as the size of the table grows, the size of mapping also grows. These mappings are usually kept in the primary memory so that address fetch should be faster. Then the secondary memory searches the actual data based on the address got from mapping. If the mapping size grows then fetching the address itself becomes slower. In this case, the sparse index will not be efficient. To overcome this problem, secondary indexing is introduced.

In secondary indexing, to reduce the size of mapping, another level of indexing is introduced.



SQL Server has two types of indexes: clustered index and non-clustered index.

A clustered index stores data rows in a sorted structure based on its key values. Each table has only one clustered index because data rows can be only sorted in one order. A table that has a clustered index is called a clustered table.

A clustered index organizes data using a special structured so-called [B-tree](https://en.wikipedia.org/wiki/B-tree) (or balanced tree). In this structure, the top node of the B-tree is called the **root node**. The nodes at the bottom level are called the **leaf nodes**. Any index levels between the root and the leaf nodes are known as intermediate levels. In the B-Tree, the root node and intermediate-level nodes contain index pages that hold index rows. The leaf nodes contain the data pages of the underlying table. The pages in each level of the index are linked using another structure called a doubly-linked list.

When you create a table with a [primary key](https://www.sqlservertutorial.net/sql-server-basics/sql-server-primary-key/), SQL Server automatically creates a corresponding clustered index that includes primary key columns.

ACID Properties

## **Atomicity**

* It states that all operations of the transaction take place at once if not, the transaction is aborted.
* There is no midway, i.e., the transaction cannot occur partially. Each transaction is treated as one unit and either run to completion or is not executed at all.

Atomicity involves the following two operations:

**Abort:** If a transaction aborts then all the changes made are not visible.

**Commit:** If a transaction commits then all the changes made are visible.

## **Consistency**

* The integrity constraints are maintained so that the database is consistent before and after the transaction.
* The execution of a transaction will leave a database in either its prior stable state or a new stable state

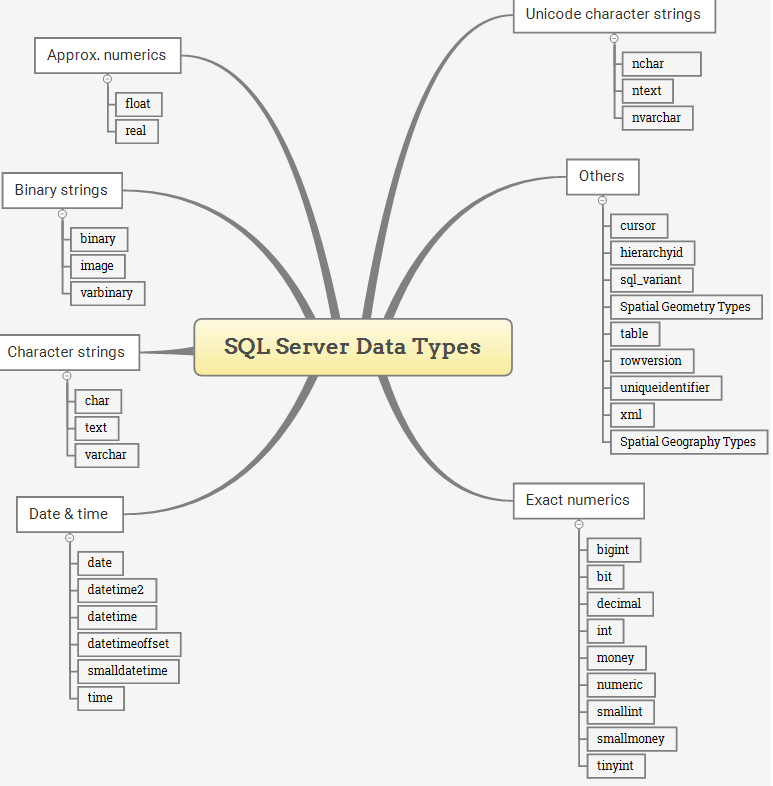
## **Isolation**

* It shows that the data which is used at the time of execution of a transaction cannot be used by the second transaction until the first one is completed.
* In isolation, if the transaction T1 is being executed and using the data item X, then that data item can't be accessed by any other transaction T2 until the transaction T1 ends.

## **Durability**

* The durability property is used to indicate the performance of the database's consistent state. It states that the transaction made the permanent changes.
* They cannot be lost by the erroneous operation of a faulty transaction or by the system failure. When a transaction is completed, then the database reaches a state known as the consistent state. That consistent state cannot be lost, even in the event of a system's failure.

DATA TYPES



CHAR

To store fixed length, non-Unicode string data, you use the SQL Server CHAR data type

CHAR(n) 🡪 n is from 1 to 8,000. Default 1. Require 1 byte to store a character.

If the length of the inserted value is less than the length specified, SQL Server will add trailing spaces. However, when you select this string value, SQL Server removes the trailing spaces before returning it.

In SQL Server, LEN function returns the number of characters in a specified column that excludes the trailing blanks and the DATALENGTH function returns the number of bytes.

SELECT val,LEN(val), DATALENGTH(val)

FROM sql\_server\_char;

Output is ABC 3 10 for CHAR(10)

NCHAR

To store fixed-length, Unicode character string data in the database, you use the SQL Server NCHAR data type:

NCHAR(n) 🡪 n is from 1 to 4,000. Default 1. Require 2 bytes to store a character.

To insert data to column, must prefix the Unicode character string with the letter N.

INSERT INTO test.sql\_server\_nchar (val)

VALUES (N'あ');

VARCHAR

To store variable-length, non-Unicode string data.

VARCHAR(n) 🡪 n is from 1 to 8,000. Default 1. Require 1 byte to store a character.

VARCHAR(max) 🡪 2GB

NVARCHAR

To store variable-length, Unicode string data.

NVARCHAR(n) 🡪 n is from 1 to 4,000. Default 1. Require 2 bytes to store a character.

DATE

It takes 3 bytes to store a DATE value. The default literal string format of a DATE value is YYYY-MM-DD. The range of a DATE value is from 0001-01-01 through 9999-12-31.

TIME

The SQL Server TIME data type defines a time of a day based on 24-hour clock.

TIME[ (n) ] 🡪 n - the number of digits for the fractional part of the seconds. Valid values are 0 to 7. Default value is 7.

The default literal format for a TIME value is 🡪 hh:mm:ss[.nnnnnnn]

DATETIME

To store both date and time in the database

DATETIME

YYYY-MM-DD hh:mm:ss[.nnn] 🡪Valid range: 1753-01-01.000 to 9999-12-31.999

Default value - 1900-01-01 00:00:00

DATETIME2

To store both date and time in the database

DATETIME2(fractional seconds precision)

YYYY-MM-DD hh:mm:ss[.nnnnnnn] 🡪Valid range: 0001-01-01 to 9999-12-31.9999999

SMALLDATETIME

Defines a date that is combined with a time of day, with seconds always zero (:00) and without fractional seconds.

SMALLDATETIME

Valid range: 1900-01-01 through 2079-06-06

DATETIMEOFFSET

Date is combined with a time of a day based on a 24-hour clock like [datetime2](https://learn.microsoft.com/en-us/sql/t-sql/data-types/datetime2-transact-sql?view=sql-server-ver16), and adds time zone awareness based on Coordinated Universal Time (UTC).

yyyy-MM-dd HH:mm:ss[.nnnnnnn] [{+|-}hh:mm]

SELECT

In this case, SQL Server processes the clauses in the following sequence: FROM, [WHERE](https://www.sqlservertutorial.net/sql-server-basics/sql-server-where/), [GROUP BY](https://www.sqlservertutorial.net/sql-server-basics/sql-server-group-by/),  HAVING, SELECT, and [ORDER BY](https://www.sqlservertutorial.net/sql-server-basics/sql-server-order-by/)

SELECT city, COUNT (\*)

FROM sales.customers

WHERE state = 'CA'

GROUP BY city

HAVING COUNT (\*) > 10

ORDER BY city;

To filter groups based on one or more conditions, you use the [HAVING](https://www.sqlservertutorial.net/sql-server-basics/sql-server-having/) clause. [WHERE](https://www.sqlservertutorial.net/sql-server-basics/sql-server-where/) clause filters rows while the HAVING clause filter groups.

ORDER BY

SQL Server return a result set with rows in an unspecified order. The only way to guarantee that the rows in the result set are sorted is to use the ORDER BY clause.

SELECT select\_list

FROM table\_name

ORDER BY column\_name | expression [ASC | DESC ];

SELECT first\_name,last\_name

FROM sales.customers

ORDER BY LEN(first\_name) DESC; //Expression example

SELECT first\_name,last\_name

FROM sales.customers

ORDER BY 1, 2; //Ordinal positions of the columns

OFFSET and FETCH

The OFFSET and FETCH clauses are the options of the [ORDER BY](https://www.sqlservertutorial.net/sql-server-basics/sql-server-order-by/) clause. They allow you to limit the number of rows to be returned by a [query](https://www.sqlservertutorial.net/sql-server-basics/sql-server-select/).

ORDER BY column\_list [ASC | DESC]

OFFSET offset\_row\_count {ROW | ROWS}

FETCH {FIRST | NEXT} fetch\_row\_count {ROW | ROWS} ONLY

row\_count 🡪 can be a constant, variable, or parameter that is greater or equal to zero.

SELECT

product\_name,list\_price

FROM

production.products

ORDER BY

list\_price,product\_name

OFFSET 10 ROWS

FETCH NEXT 10 ROWS ONLY;

SELECT

product\_name,list\_price

FROM

production.products

ORDER BY

list\_price DESC,product\_name

OFFSET 0 ROWS

FETCH FIRST 10 ROWS ONLY;

TOP

Because the order of rows stored in a table is unspecified, the SELECT TOP statement is always used in conjunction with the [ORDER BY](https://www.sqlservertutorial.net/sql-server-basics/sql-server-order-by/) clause.

SELECT TOP (expression) [PERCENT] [WITH TIES]

FROM table\_name

ORDER BY column\_name;

SELECT TOP 10 product\_name, list\_price

FROM production.products

ORDER BY list\_price DESC;

The production.products table has 321 rows, therefore, one percent ( 3.21), after rounds it up to the next whole number which is four ( 4) in this case.

SELECT TOP 1 PERCENT product\_name, list\_price

FROM production.products

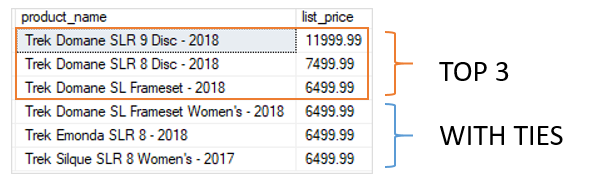
ORDER BY list\_price DESC;

The WITH TIES allows you to return additional rows with values that match those of the last row in the limited result set.

SELECT TOP 3 WITH TIES product\_name, list\_price

FROM production.products

ORDER BY list\_price DESC;



DISTINCT

* Use the SQL Server SELECT DISTINCT clause to retrieve the distinct values from one or more columns.

The DISTINCT clause keeps only one NULL in the column and removes other NULLs.

Both DISTINCT and GROUP BY clause reduces the number of returned rows in the result set by removing the duplicates.

SELECT city, state, zip\_code

FROM sales.customers

GROUP BY city, state, zip\_code

ORDER BY city, state, zip\_code

SELECT

DISTINCT city, state, zip\_code

FROM

sales.customers;

Both are equivalent.