SQL Questions interview?

☺ What is a database?

A database is a collection of information that is organized so that it can be easily accessed, managed and updated.

☺ Evolution of Database

1. **File System**: In the early days, user data was stored in flat files.
2. **Relational Database**: In this model, data is organized in tables.
3. **NoSQL Database**: In this model, data is stored in a non-tabular format.
4. **Graph Database**: In this model, data is stored in graph format.
5. **NewSQL Database**: In this model, data is stored in a relational format but with new features.

☺ Database vs DBMS?

Database: A database is a collection of information that is organized so that it can be easily accessed, managed and updated.

DBMS: A DBMS (Database Management System) is software that is used to manage databases e.g. MySQL, Oracle, SQL Server.

☺ ER Diagram Entity Types and Attributes

In an Entity-Relationship (ER) diagram, entities represent objects or concepts, while attributes provide details about these entities. The main types of entities and attributes are as follows:

Entities

1. Strong Entity: Exists independently (e.g., Customer).

Weak Entity: Depends on a strong entity (e.g., Order Item).

Attributes

1. Simple Attribute: Cannot be divided further (e.g., First Name).
2. Composite Attribute: Can be divided into smaller sub-parts (e.g., Full Name as First Name + Last Name).
3. Single-Valued Attribute: Holds a single value (e.g., Age).
4. Multi-Valued Attribute: Can hold multiple values (e.g., Phone Numbers).
5. Derived Attribute: Can be derived from other attributes (e.g., Age from Date of Birth)

☺ Relationship in SQL: Degree, Cardinality, and Participation

 Degree of a Relationship:

- The degree of a relationship refers to the **number of entities** involved in a relationship.
  - **Unary**: Involves one entity type (e.g., employee supervising another employee).
  - **Binary**: Involves two entity types (e.g., customer and order).
  - **Ternary**: Involves three entity types (e.g., supplier, product, and warehouse).

 Cardinality of a Relationship:

- Cardinality describes the **number of instances** of one entity that can or must be associated with each instance of another entity.
  - **One-to-One (1:1)**: Each entity in A is related to at most one entity in B and vice versa.
  - **One-to-Many (1)**: Each entity in A can be related to many entities in B, but each entity in B is related to at most one entity in A.
  - **Many-to-Many (M)**: Entities in A can relate to multiple entities in B, and vice versa.

 Participation:

- Participation refers to whether **all entities** in a set participate in the relationship. It can be:
  - **Total Participation**: Every entity in the entity set must participate in the relationship (e.g., every employee must be assigned to a department).
  - **Partial Participation**: Some entities may not participate in the relationship (e.g., some departments might not have employees).

☺ ER Mapping Rules

Mapping an Entity-Relationship (ER) diagram to a relational database involves several key rules to ensure that entities, attributes, and relationships are accurately represented. Here are the primary mapping rules:

1. **Entities to Tables**: Each strong entity in the ER diagram becomes a separate table in the relational model.
2. **Primary Keys**: Identify a primary key for each table, typically using a unique attribute or a combination of attributes from the entity.
3. **Attributes to Columns**: Attributes of each entity are mapped to columns in the corresponding table. Simple attributes become individual columns, while composite attributes can be split into multiple columns.
4. **Weak Entities**: A weak entity is mapped to a table that includes its attributes and a foreign key referencing the strong entity it depends on.
5. **Relationships**:

- **One-to-One**: Implemented by adding a foreign key to either of the tables involved.
- **One-to-Many**: The primary key of the "one" side is added as a foreign key in the "many" side.
- **Many-to-Many**: Requires a new table that contains foreign keys referencing both entities, often along with additional attributes specific to the relationship

☺ **What is Normalization?**

Normalization is the process of organizing data in a database to eliminate redundancy and dependency issues. It involves splitting tables into smaller, more manageable entities.

☺ **What is Denormalization?**

Denormalization is the inverse process of normalization, where the normalized schema is converted into a schema that has redundant information. The performance is improved by using redundancy and keeping the redundant data consistent. The reason for performing denormalization is the overheads produced in the query processor by an over-normalized structure.

☺ **What are the various forms of Normalization?**

The different normal forms are:

i. First Normal Form (1NF): Eliminates duplicate rows and ensures atomicity of values.
ii. Second Normal Form (2NF): Ensures that each non-key column depends on the entire primary key.
iii. Third Normal Form (3NF): Ensures that each non-key column depends only on the primary key and not on other non-key columns.
iv. Fourth Normal Form (4NF): Eliminates multi-valued dependencies.
v. Fifth Normal Form (5NF): Eliminates join dependencies.

☺ **What methods do you use to ensure data normalization and avoid redundancy in complex relational databases?**

To ensure data normalization and avoid redundancy in complex relational databases, the following methods are applied:

1. Apply Normal Forms:
   - First Normal Form (1NF)
   - Second Normal Form (2NF)
   - Third Normal Form (3NF)
2. Minimize Redundancy: Break down large, complex tables into smaller, well-defined tables to minimize duplication of data across the database.
3. Establish Clear Relationships: Use foreign keys to create relationships between normalized tables, linking related data while keeping it stored efficiently and separately.
4. Normalize with Functional Dependencies: Ensure that each column is functionally dependent on the primary key, thus eliminating redundant data storage.
5. Monitor and Refine Design: Continuously monitor the database design, refactoring, when necessary, as the system grows, or business requirements change.

☺ Database development steps

1. **Requirement Analysis**: Understand the requirements of the database.
2. **Design**: Design the ERD.
3. **Mapping**: Map the ERD to schema.
4. **Implementation**: Implement the database using RDBMS.

☺ SQL vs MySQL?

SQL: SQL stands for Structured Query Language. It is a language used to communicate with databases.
MySQL: MySQL is a type of RDBMS that uses SQL as its language

☺ SQL is a declarative language, what does it mean?

SQL is a declarative language, which means that you specify what you want to retrieve, insert, update, or delete from the database, but you do not specify how to do it also it doesn't have control flow statements like if, else, loops.

☺ What are SQL commands categories? What is the difference between DML, DDL and DCL?

1. **DDL**: Data Definition Language It is used to **define the structure of the database** e.g. **Create, Alter, Drop, Truncate**.
2. **DML**: Data Manipulation Language. It is used to **manipulate the data in the database** e.g. **Insert, Update, Delete.**
3. **DCL**: Data Control Language. It is used to **control the access to the database** e.g. **Grant, Revoke.**
4. **TCL**: Transaction Control Language. It is used to **manage transactions in the database** e.g. **Commit, Rollback.**
5. **DQL**: Data Query Language. It is used to **query the database** e.g. Select.

☺ Drop, Truncate and Delete Statement.

Feature	Drop	Truncate	Delete
Definition	Removes a table from the database.	Removes all rows from a table but keeps the table structure.	Removes rows from a table based on a condition.
Rollback	No	No	Yes
Type	DDL (Data Definition Language)	DDL (Data Definition Language)	DML (Data Manipulation Language)

☺ Select and Select into Statement.

Select is primarily used for querying and displaying data from existing tables.

Select Into allows for both data retrieval and the creation of a new table, making it useful for data backup or transformation tasks.

☺ SQL Server has control flow statements, despite being a declarative language, how?

SQL Server has control flow statements like if, else, loops, etc. in the form of T-SQL (Transact-SQL) which is an extension of SQL.

☺ What is the rules in SQL?

Rules are used to enforce business rules in the database. They are similar to constraints but are more flexible. Rules are related to the column and **can be shared between tables** using sp_bindrule, can apply **only one rule to a column**.

☺ What are constraints in SQL and types of Constraints?

Constraints are objects inside a table that applies restrictions to enforce the integrity of the data in the database, constraints are either column level or table level. Constraints are related to the table and **can't be shared between tables, can apply multiple constraints to a column**

1. **Primary Key**: Uniquely identifies each record in a table.
2. **Foreign Key**: Ensures referential integrity between two tables.
3. **Unique**: Ensures that all values in a column are different. This provides uniqueness for the column(s) and helps identify each row uniquely.
4. **Check**: Ensures that all values in a column satisfy a specific condition. A CHECK constraint is used to limit the values or type of data that can be stored in a column. They are used to enforce domain integrity.
5. **Default**: Provides a default value for a column when none is specified.
6. **Not Null**: Ensures that a column cannot have a NULL value.

☺ What is UNIQUE constraint?

A UNIQUE constraint ensures that all values in a column are different. It is similar to the PRIMARY KEY constraint, except that it can contain NULL values. A table in SQL can have multiple UNIQUE constraints.

☺ What is the difference between a rule and a constraint?

Feature	Rule	Constraint
Definition	A rule is a guideline or policy that dictates how data should be handled.	A constraint is a restriction applied to data to ensure its integrity and accuracy.
Application	Often used for validation or business logic.	Enforced automatically by the database system.
Types	Can be custom-defined based on business needs.	Common types include primary key, foreign key, unique, and check constraints.
New vs. Old Data	Can be applied to both new and existing data, often requiring manual enforcement.	Typically applies automatically to new data; existing data must comply with constraints or be corrected.
Enforcement	Generally requires explicit execution or trigger mechanisms.	Automatically enforced by the database engine upon data modification.

☺ What is the purpose of the CASCADE DELETE constraint?

The CASCADE DELETE constraint is used to automatically delete related rows in child tables when a row in the parent table is deleted.

☺ What are Types of Keys in SQL? What is the difference between a primary key and a candidate key?

In a database management system (DBMS), keys are used to identify unique rows of data in a table and establish relationships between tables. The common types of keys are:

Primary Key: Uniquely identifies each row in a table. It cannot contain NULL values.
Candidate Key: A set of one or more columns that can uniquely identify rows. The primary key is selected from the candidate keys.
Super Key: A set of one or more columns that, together, uniquely identify rows in a table.
Alternate Key: The candidate keys that were not chosen as the primary key.
Foreign Key: A field in one table that refers to the primary key in another table, used to establish relationships.
Composite Key: A composite key is a primary key composed of two or more columns. Together, these columns uniquely identify each record in a table.
Artificial Key: A key created manually, often when no natural primary key exists.

☺ What is the primary key?

A primary key constraint is a unique identifier for a record in a table. It must contain UNIQUE values and has an implicit NOT NULL constraint. A table in SQL is strictly restricted to having one and only one primary key, which is comprised of single or multiple fields (columns).

☺ What is the foreign key?

A foreign key is a column or combination of columns that establishes a link between data in two tables. It ensures referential integrity by enforcing relationships between tables.

☺ What is a self-referencing foreign key?

A self-referencing foreign key is a foreign key that references the primary key of the same table. It is used to establish hierarchical relationships within a single table.

☺ What is the purpose of the CASCADE keyword in a FOREIGN KEY constraint?

The CASCADE keyword is used to specify that changes made to the primary key values in the referenced table should be propagated to the foreign key values in the referring table. This ensures that the relationship remains valid.

☺ Difference between Primary Key and Secondary Key?

Primary Key

- Definition: Uniquely identifies each record in a table.
- Constraints: Cannot contain NULL values; all values must be unique.
- Usage: Each table can have only one primary key, often referenced by foreign keys in other tables

Secondary Key

- Definition: A key not selected as the primary key but can still retrieve records.
- Characteristics: Typically, non-unique and used for faster search operations.
- Flexibility: A table can have multiple secondary keys, which do not have the same uniqueness constraints as primary keys.

☺ What is the difference between keys in SQL? What is the difference between a primary key and a unique key?

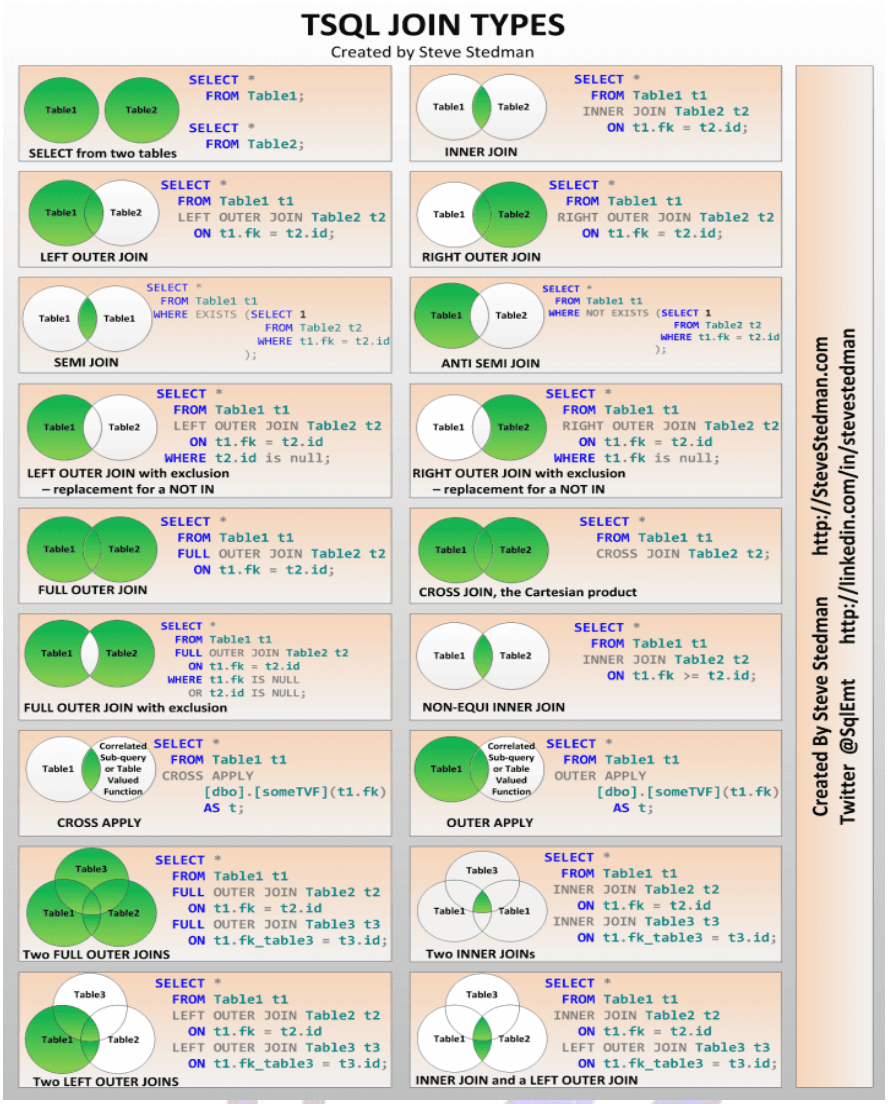
Key Type	Definition	Characteristics	Usage
Primary Key	Uniquely identifies each row in a table.	Must be unique, cannot be NULL.	Enforces entity integrity.
Foreign Key	Links records from one table to another.	Points to a primary key in another table; can be NULL.	Maintains referential integrity.
Unique Key	Ensures all values in a column are distinct.	Can accept NULLs, but only one NULL value per column.	Guarantees uniqueness of data.
Candidate Key	A set of fields that can uniquely identify rows.	Any candidate key can become a primary key.	Identifies potential primary keys.
Composite Key	A key made up of two or more columns.	Uniquely identifies rows through a combination of columns.	Used when no single column is sufficient.
Super Key	A set of one or more attributes that uniquely identify a row.	Can contain extra attributes that are not necessary for uniqueness.	General concept for any combination of columns.

☺ What is join and its types?

- A join is used to combine rows from two or more tables based on a related column between them e.g. primary key and foreign key.
- When to Use:
  - Combining Related Tables: Use joins to retrieve data from multiple related tables (e.g., customer orders and product details).
  - Performance: Joins are often more efficient for retrieving and combining related data in a single query, especially with large datasets.

Types of joins:

1. **INNER JOIN**: Returns rows that have matching values in both tables involved in the join.
2. **LEFT JOIN**: Returns all rows from the left table, and the matched rows from the right table.
3. **RIGHT JOIN**: Returns all rows from the right table, and the matched rows from the left table.
4. **FULL JOIN**: Returns rows when there is a match in one of the tables.
5. **CROSS JOIN**: Returns the Cartesian product of the two tables.
6. **SELF JOIN**: Joining a table by itself.
7. **full outer join** returns all rows from both tables, including unmatched rows, and
8. combines them based on the join condition.
9. **NATURAL JOIN**: Joining two tables based on the same column names.



☺ What is the purpose of the CROSS APPLY operator?

Definition: The CROSS APPLY operator is used to join a table with a table-valued function or subquery that takes parameters from each row of the outer table. It functions similarly to an INNER JOIN but allows for more complex transformations based on outer table values.

Key Features:

Row-wise Execution: Executes the table-valued function for each row of the outer table, making it suitable for scenarios where the function's output depends on the outer row data.

Results Filtering: Returns only those rows from the outer table for which the function produces a result. If no results are returned for a specific outer row, that row is excluded from the final output.

Use Cases: Ideal for applying complex calculations or fetching related data based on the context of each row.

Summary: CROSS APPLY enhances SQL query capabilities, particularly when working with functions that require individual row data from a primary table.

Example:

Explain

sql Copy code

```sql
SELECT
    e.EmployeeID,
    e.EmployeeName,
    d.DepartmentName
FROM
    Employees e
CROSS APPLY
    (SELECT DepartmentName
     FROM Departments d
     WHERE d.DepartmentID = e.DepartmentID) AS dept;
```

Explanation:

- The query retrieves each employee and their corresponding department name.
- `CROSS APPLY` is used to apply the subquery (`Departments`) to each row of the `Employees` table. It evaluates the subquery for each row of the `Employees` table, allowing you to perform operations that depend on the columns from the outer query.

☺ What is a recursive SQL query?

a. A recursive SQL query is a query that refers to its own output in order to perform additional operations. It is commonly used for hierarchical or tree-like data structures.

☺ What is a subquery? What is the difference between a correlated subquery and a nested subquery? What is a correlated subquery?

A subquery is a query within another query. It is used to return data that will be used in the main query.

Types of subqueries:

1. **A correlated subquery** depends on the outer query for its values. It cannot be executed independently because it references columns from the outer query's tables.
2. **A non-correlated subquery** is an Independent of the outer query that can be run separately from the outer query. It does not reference any columns from the outer query.

☺ What are the differences between using subqueries and joins in SQL, and when would you choose one over the other?

- Subqueries return data to be used by the outer query, while joins combine data from multiple tables in a single result set.
- Joins are generally preferred for combining related tables, whereas subqueries are ideal for filtering or when you need to break down complex logic into smaller steps.

☺ What is a self-referencing table?

a. A self-referencing table is a table that has a foreign key column referencing its own primary key. It is used to represent hierarchical relationships within a single table.

☺ What is A Common Table Expression (CTE)?

A Common Table Expression (CTE) is a temporary result set in SQL that you can reference within a SELECT, INSERT, UPDATE, or DELETE statement. It simplifies complex queries by breaking them down into more manageable parts, improving readability and maintainability.

Usage:

1. **Readability**: CTEs make SQL queries easier to read and understand, especially when dealing with complex logic.
2. **Reusability**: A CTE can be used multiple times within the same query, which reduces redundancy.
3. **Recursion**: CTEs can be recursive, allowing you to perform operations like hierarchical data queries (e.g., organizational charts).
4. **Modularity**: They help separate different parts of a query, making it easier to debug and modify.

CTEs are defined at the beginning of the query with the WITH keyword, followed by the CTE name and its query definition

- **How do you manage and optimize SQL queries that involve multiple correlated subqueries?**

Managing and optimizing SQL queries with multiple correlated subqueries is crucial for improving query performance. Correlated subqueries can be costly because they execute the inner query repeatedly for each row in the outer query. Here are key techniques to optimize them:

1. Rewrite Subqueries as Joins
2. Use Indexes
3. Optimize with EXISTS/NOT EXISTS:
4. Limit Subquery Execution CTE or temporary tables.
5. Simplify Subquery Logic

- **What is the difference between CTEs, and subqueries? Which is faster between CTE and Subquery?**

Factor	CTEs	Subqueries
Readability	Often more readable, especially in complex queries.	Can become complex and harder to read in nested cases.
Reusability	Can be referenced multiple times in the main query.	Typically evaluated each time it is referenced.
Optimization	May allow for better optimization by the query engine.	Depends heavily on how the database handles nested queries.
Performance	Generally performs similarly to subqueries but can be slower in some cases due to re-evaluation.	Can be faster for simpler queries, but performance may degrade with complexity.
Execution Plan	The execution plan may differ based on the optimizer's treatment of CTEs.	Execution plan usually straightforward, but complexity can lead to inefficiencies.

☺ How many types of clauses are there in SQL? What is the difference between GROUP BY and HAVING clauses

SQL contains several types of clauses that serve various functions within queries. Here are some of the main types:

1. **SELECT**: Specifies the columns to be returned.
2. **FROM**: Indicates the tables from which to retrieve data.
3. **WHERE**: Filters records based on specific conditions.
4. **GROUP BY**: is used to group rows based on one or more columns.
5. **HAVING**: Filters groups based on aggregate conditions.
6. **ORDER BY:** Sorts the result set based on one or more columns
7. in ascending or descending order.
8. **JOIN**: Combines rows from two or more tables based on related columns.
9. **LIMIT / TOP**: Restricts the number of rows returned.

☺ What is the difference between WHERE and HAVING clause?

- **WHERE** clause is used to filter rows before grouping.
- **HAVING** clause is used to filter groups after grouping.

☺ What is the purpose of the LIKE operator?

The LIKE operator is used in a WHERE clause to search for a specified pattern in a column. It allows wildcard characters like % (matches any sequence of characters) and _ (matches any single character).

☺ What is the difference between the IN and EXISTS operators?

IN: The IN operator is used to check if a value exists in a list of values or in the result set of a subquery. IN performs better when dealing with a small set of values in the subquery. It compares values directly, which can be faster for smaller datasets.

EXISTS: The EXISTS operator is used to check if a subquery returns any rows. It stops searching once it finds the first matching row, making it ideal for checking the existence of data. EXISTS performs better for larger datasets since it stops execution as soon as a match is found, reducing unnecessary comparisons.

☺ What are the types of functions in SQL? What are Aggregate and Scalar functions?

SQL functions can be categorized into several types, each serving different purposes in data manipulation and querying. The main types include:
SQL functions can be categorized into several types, each serving different purposes:

1. **A NULL Functions:** Handle NULL values e.g. ISNULL, COALESCE.
2. **Conversion Functions:** Convert data types e.g. CAST, CONVERT. Both functions serve to change data types in SQL, but CONVERT allows for additional formatting options, making it useful for date and time conversions.
3. **Aggregate Functions**: These functions perform calculations on a set of values and return a single value. They are commonly used in conjunction with the GROUP BY and HAVING clauses of the SELECT statement. Note: All aggregate functions described above ignore NULL values except for the COUNT function. Examples include:
    - SUM(): Adds up all values.
    - COUNT(): Counts the number of rows.
    - AVG(): Calculates the average value.

- MAX(): Finds the maximum value.
- MIN(): Finds the minimum value.

4. **Scalar Functions**: These return a single value based on the input value and can be used anywhere in a SQL statement. They include:
   - **String Functions**: Manipulate string values
     - LEN() - Calculates the total length of the given field (column).
     - UCASE() - Converts a collection of string values to uppercase characters.
     - LCASE() - Converts a collection of string values to lowercase characters.
     - MID() - Extracts substrings from a collection of string values in a table.
     - CONCAT() - Concatenates two or more strings.
   - **Numeric Functions**: Perform operations on numbers (e.g., ROUND (), ABS()).
     - RAND() - Generates a random collection of numbers of a given length.
     - ROUND() - Calculates the round-off integer value for a numeric field (or decimal point values).
   - **Date Functions**: Handle date and time values (e.g., GETDATE (), DATEDIFF ())
     - NOW() - Returns the current date & time.
     - FORMAT() - Sets the format to display a collection of values.
5. **User-Defined Functions**: Users can create their own functions to encapsulate complex logic. These can be scalar (returning a single value) or table-valued (returning a table).
6. **System Functions**: These built-in functions provide information about the database or perform specific tasks, including:
   - **Metadata Functions**: Retrieve information about database objects.
   - **Security Functions**: Provide details about users and role
7. **Ranking Functions:** Assign a rank to each row in a result set e.g. RANK, DENSE_RANK, ROW_NUMBER, NTILE.
8. **Window Functions**: Perform calculations across a set of rows related to the current row e.g. LAG, LEAD, FIRST_VALUE, LAST_VALUE.
9. **Logical Functions:** Perform logical operations e.g. IIF, CHOOSE, SWITCH.

☻ What are the ranking functions in SQL?

Ranking functions in SQL are used to assign a rank to each row within a result set based on specified criteria. Here's a brief overview of some common ranking functions:

1. **RANK ()**: Assigns a unique rank number to each row within a partition. If two rows share the same rank, the next rank will skip the next number (e.g., 1, 1, 3).

```
SELECT
  name,
  department,
  salary,
  RANK() OVER (PARTITION BY department ORDER BY salary DESC) AS rank
FROM employees;
```

2. **DENSE_RANK ()**: Similar to RANK (), but it does not skip rank numbers. If two rows are tied, the next rank is consecutive (e.g., 1, 1, 2).

```
SELECT
  name,
  department,
  salary,
  DENSE_RANK() OVER (PARTITION BY department ORDER BY salary DESC) AS
dense_rank
FROM employees;
```

3. **ROW_NUMBER ()**: Assigns a unique sequential integer to rows within a partition, starting at 1 for the first row. Unlike RANK () and DENSE_RANK (), there are no ties.
4. **NTILE(n)**: Divides the result set into 'n' number of approximately equal parts and assigns a bucket number to each row.

```
SELECT
  name,
  department,
  salary,
  NTILE(4) OVER (PARTITION BY department ORDER BY salary DESC) AS bucket
FROM employees;
```

These functions are essential for data analysis and reporting, allowing for efficient ranking and categorization of data.

☺ What is the difference between RANK, DENSE_RANK, ROW_NUMBER?

Function	Description	Handling of Ties
RANK()	Assigns a rank to each row within a partition. If two rows share the same rank, the next rank is skipped (e.g., 1, 1, 3).	Ties receive the same rank, but the next rank is skipped. [1]
DENSE_RANK()	Similar to RANK(), but does not skip ranks after ties (e.g., 1, 1, 2).	Ties receive the same rank, and the next rank is consecutive. [2]
ROW_NUMBER()	Assigns a unique sequential integer to each row, starting at 1 for the first row in each partition.	No ties; every row gets a unique number regardless of values. [4]

☺ What are the window functions in SQL?

Window functions in SQL enable calculations across a specific set of rows related to the current row while retaining the original row structure. Key functions include:

1. **LAG**: Retrieves a value from a previous row in the result set, allowing comparisons across rows. Provides access to a row at a specified physical offset before the current row in the result set. This will return the salary from the previous row for each employee.

```
SELECT
  name,
  department,
  salary,
  LAG(salary, 1) OVER (PARTITION BY department ORDER BY salary) AS
previous_salary
FROM employees;
```

2. **LEAD**: Like LAG, but accesses data from a subsequent row, but it accesses the next row's data.

```
SELECT
  name,
  department,
  salary,
  LEAD(salary, 1) OVER (PARTITION BY department ORDER BY salary) AS
next_salary
FROM employees;
```

3. **FIRST_VALUE**: Returns the first value in an ordered set of values for each row.

FIRST_VALUE: Returns the first value in a set.

SELECT employee_id, salary, FIRST_VALUE (salary) OVER (ORDER BY salary) AS FirstSalary FROM employees;

This will return the first salary in the set (the minimum salary in that case).

4. **LAST_VALUE**: Returns the last value in the defined window for each row.

LAST_VALUE: Returns the last value in a set.

SELECT employee_id, salary, LAST_VALUE (salary) OVER (ORDER BY salary) AS LastSalary FROM employees.

This will return the last salary in the set (the maximum salary in that case).

These functions enhance analytical capabilities by allowing complex computations, such as running totals, moving averages, and ranking, without grouping rows into a single output.

☺ **what is the difference between FIRST_VALUE and LAST_VALUE?**

FIRST_VALUE: Returns the first value in a set.

LAST_VALUE: Returns the last value in a set.

☺ **What is the difference between LAG and LEAD?**

LAG: Returns the value of a column in the previous row.

LEAD: Returns the value of a column in the next row.

☺ **What is ROWS/RANGE window function?**

ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW: Includes all rows from the start of the partition to the current row, allowing cumulative calculations (e.g., running totals).

RANGE BETWEEN 1 PRECEDING AND 1 FOLLOWING: Includes rows that fall within one value before and one value after the current row, useful for calculations based on a specific range of values rather than specific row counts.

```
SELECT
  name,
  department,
  salary,
  SUM(salary) OVER (PARTITION BY department ORDER BY salary ROWS BETWEEN
UNBOUNDED PRECEDING AND CURRENT ROW) AS running_total
FROM employees;
```

☺ **What is User-defined function? What are its various types?**

The user-defined functions in SQL are like functions in any other programming language that accept parameters, perform complex calculations, and return a value. They are written to use

the logic repetitively whenever required. There are two types of SQL user-defined functions:

- Scalar Function: As explained earlier, user-defined scalar functions return a single scalar value.
- Table-Valued Functions: User-defined table-valued functions return a table as output.
    - Inline: returns a table data type based on a single SELECT statement.
    - Multi-statement: returns a tabular result-set but, unlike inline, multiple **SELECT** statements can be used inside the function body.

☺ **Can I use DDL commands (Create, Alter, Drop, Truncate) inside a function?**

No, you can't use DDL commands inside a function, only DML commands (Insert, Update) are allowed and select statements.

☺ **Inline vs Multi-Statement Table-Valued Functions?**

Inline Table-Valued Functions: Return a table using SELECT statement.

Multi-Statement Table-Valued Functions: Return a table using multiple statements e.g. SELECT, IF, WHILE, Declare etc.

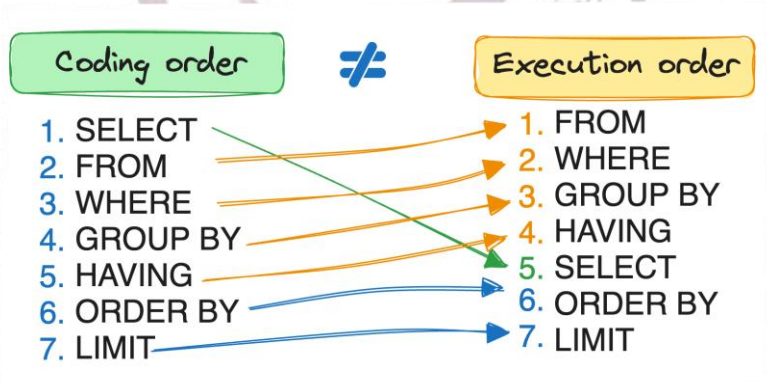
☺ **What is the main disadvantage of deleting data from an existing table using the DROP TABLE command?**

DROP TABLE command deletes complete data from the table along with removing the complete table structure too. In case our requirement entails just remove the data, then we would need to recreate the table to store data in it. In such cases, it is advised to use the TRUNCATE command.

☺ **What is the execution order of SQL query?**

The execution order of an SQL query is different from the way the query is written. Below is the typical logical execution order of a standard SQL query:

1. **FROM**: Specifies the table(s) from which to retrieve or manipulate data.
2. **JOIN**: Combines rows from two or more tables based on a related column.
3. **WHERE**: Filters rows before any grouping occurs based on specified conditions.
4. **GROUP BY**: Groups rows that have the same values into summary rows.
5. **HAVING**: Filters groups based on conditions (used with aggregate functions).
6. **SELECT**: Selects the columns to display, including applying any aggregate functions.
7. **ORDER BY:** Sorts the final result set by one or more columns.
8. **LIMIT/OFFSET:** Limits the number of rows returned.



☺ What is UNION, INTERSECT, MINUS?

UNION: Combines the result of two or more SELECT statements.
INTERSECT: Returns the common rows between two SELECT statements.
MINUS: Returns the rows that are in the first SELECT statement but not in the second SELECT statement.
Certain conditions need to be met before executing either of the above statements in SQL -

- Each SELECT statement within the clause must have the same number of columns
- The columns must also have similar data types
- The columns in each SELECT statement should necessarily have the same order

☺ What is the Difference UNION vs UNION ALL

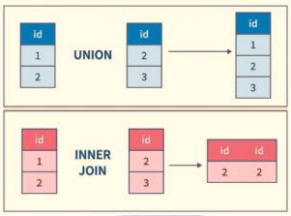
UNION: Combines the result of two or more SELECT statements and removes duplicates.
UNION ALL: Combines the result of two or more SELECT statements and keeps duplicates.

☺ SELECT INTO vs INSERT INTO?

SELECT INTO	INSERT INTO
Creates a new table and inserts the result of the SELECT statement into it.	Inserts data into an existing table.
Does not require the table to exist.	Requires the table to exist.
Does not require the column names to match.	Requires the column names to match.

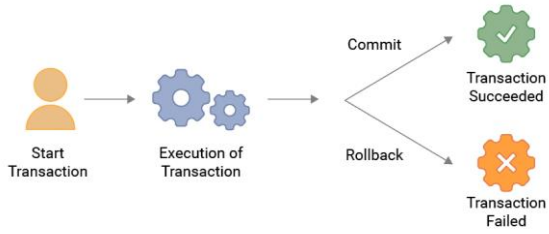
☺ What is the difference between the UNION and JOIN operators?

UNION combines the result sets of two or more SELECT statements vertically, while JOIN combines columns from two or more tables horizontally based on a join condition.



☺ Batch vs Script vs Transaction? What is a transaction?

- ✓ **Batch:** A batch is a group of SQL commands executed together as a single unit. It can include multiple SQL statements that are processed together, improving efficiency by reducing the overhead of multiple round trips to the database.
- ✓ **Script:** A script is a collection of SQL statements that can be executed in sequence. Scripts often include batches and can be used for tasks such as data manipulation and schema changes.
- ✓ **Transaction:** A transaction is a sequence of operations that are executed as a single unit of work, ensuring that either all operations succeed or none do (atomicity). Transactions provide a way to maintain data integrity.



☺ What is the transaction?

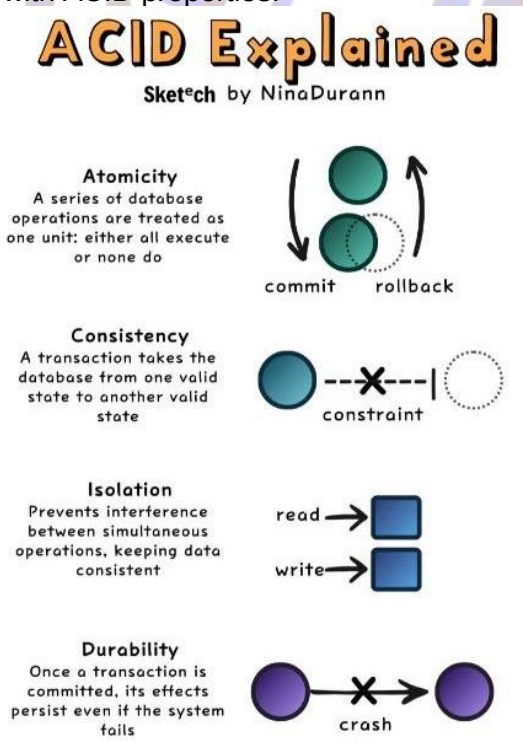
- A transaction in the context of databases and business is a sequence of operations performed as a single logical unit of work.
- It ensures that all tasks within the transaction are either complete successfully or none at all, maintaining data integrity.
- Transactions typically follow the ACID properties—Atomicity, Consistency, Isolation, and Durability—ensuring reliable processing of data.
- It ensures data consistency and integrity by either committing all changes or rolling them back if an error occurs.
- For example, a banking transaction might involve transferring funds from one account to another, where both the debit and credit operations must succeed to complete the transaction.

☺ What are ACID properties?

ACID stands for Atomicity, Consistency, Isolation, Durability. They are database transaction properties which are used for guaranteeing data validity in case of errors and failures.

- **Atomicity**: This property ensures that the transaction is completed in all-or-nothing way.
- **Consistency**: This ensures that updates made to the database is valid and follows rules and restrictions.
- **Isolation**: This property ensures integrity of transaction that are visible to all other transactions.
- **Durability**: This property ensures that the committed transactions are stored permanently in the database.

Note: PostgreSQL is compliant with ACID properties.



how to rollback a transaction?

The **ROLLBACK** statement is used to undo the changes made in a transaction

Differentiate between commit and checkpoint.

The commit action ensures that the data consistency of the transaction is maintained and it ends the current transaction in the section. Commit adds a new record in the log that describes the COMMIT to the memory. Whereas, a checkpoint is used for writing all changes that were committed to disk up to SCN which would be kept in datafile headers and control files.

What is the purpose of the SAVEPOINT statement?

The SAVEPOINT statement is used to define a specific point within a transaction to which you can roll back. It allows you to undo changes up to a specific savepoint without rolling back the entire transaction.

What is transaction types?

1. **Implicit Transaction**: Automatically starts when a SQL statement is executed.
2. **Explicit Transaction**: Manually starts and ends using the BEGIN TRANSACTION and COMMIT or ROLLBACK statements.

What is a transaction log?

The transaction log is a file that contains all the changes made to the database. It is used to recover the database in case of a failure.

How do you check the rows affected as part of previous transactions?

SQL standards state that the following three phenomena should be prevented whilst concurrent transactions. SQL standards define 4 levels of transaction isolations to deal with these phenomena.

- **Dirty reads:** If a transaction reads data that is written due to concurrent uncommitted transaction, these reads are called dirty reads.
- **Phantom reads:** This occurs when two same queries when executed separately return different rows. For example, if transaction A retrieves some set of rows matching search criteria. Assume another transaction B retrieves new rows in addition to the rows obtained earlier for the same search criteria. The results are different.
- **Non-repeatable reads:** This occurs when a transaction tries to read the same row multiple times and gets different values each time due to concurrency. This happens when another transaction updates that data and our current transaction fetches that updated data, resulting in different values.

To tackle these, there are 4 standard isolation levels defined by SQL standards. They are as follows:

- **Read Uncommitted** – The lowest level of the isolations. Here, the transactions are not isolated and can read data that are not committed by other transactions resulting in dirty reads.
- **Read Committed –** This level ensures that the data read is committed at any instant of read time. Hence, dirty reads are avoided here. This level makes use of read/write lock on the current rows which prevents read/write/update/delete of that row when the current transaction is being operated on.
- **Repeatable Read** – The most restrictive level of isolation. This holds read and write locks for all rows it operates on. Due to this, non-repeatable reads are avoided as other transactions cannot read, write, update or delete the rows.
- **Serializable** – The highest of all isolation levels. This guarantees that the execution is serializable where execution of any concurrent operations are guaranteed to be appeared as executing serially.

The following table clearly explains which type of unwanted reads the levels avoid:

Isolation Levels	Dirty Reads	Phantom Reads	Non-repeatable Reads
Read Uncommitted	Might occur	Might occur	Might occur
Read Committed	Won't occur	Might occur	Might occur
Repeatable Read	Won't occur	Might occur	Won't occur
Serializable	Won't occur	Won't occur	Won't occur

☺ What are database security levels?

1. **Server Level**: Controls access to the server.
2. **Database Level**: Controls access to the database.
3. **Schema Level**: Controls access to the schema.
4. **Object Level**: Controls access to the objects in the database.
5. **Column Level**: Controls access to the columns in the table.

☺ How the data is actually stored in the database?

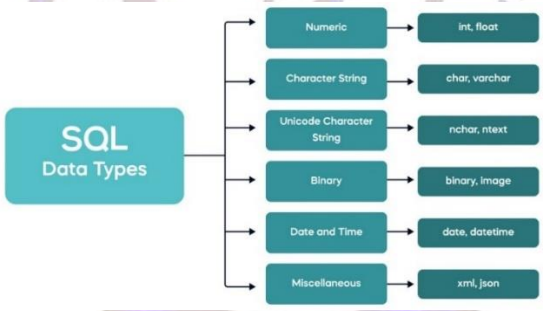
Data is stored in the database in the form of pages. A page is the smallest unit of data storage in a database. It is a fixed-size block of data that is read from or written to disk.

☺ When searching for a record in the database, how does the database engine search for it, and what is the best way to search for a record?

The database engine searches for a record in the database by reading the pages from the disk and looping through all the rows in the table to find the matching ones. The best way to search for a record is to use an index.

☺ What is SQL Data Types?

To allow the users to work with tables effectively, SQL provides us with various data types, each of which can be useful based on the type of data we handle.



☺ What is the difference between CHAR and VARCHAR data types?

CHAR is a fixed-length string data type, while VARCHAR is a variable-length string data type.

☺ Difference Between nvarchar, char, and nchar in SQL

Char: Fixed-length, non-Unicode, 1 byte per character. Ideal for storing fixed-length data, like postal codes.
nchar: Fixed-length, Unicode. 2 bytes per character. Best for fixed-length international character sets like Chinese or Arabic.

nvarchar: Variable-length, Unicode. 2 bytes per character, but optimized for varying lengths. Best for multilingual text fields with variable lengths, such as names or descriptions.

☺ What is variable types in SQL?

- **Local Variables**: These are declared within a specific scope, such as a procedure or batch, and can only be accessed within that scope. They are typically prefixed with an @ symbol in SQL Server. Local variables are useful for temporary storage of data during execution.

  ```
  DECLARE @variable_name data_type;
  SET @variable_name = value;
  ```

- **Global Variables**: These are accessible throughout the entire database session and are prefixed with @@. Global variables provide information about the server or session, such as @@version, which indicates the current version of the SQL Server

```
SELECT @@GLOBAL_VARIABLE_NAME;
SELECT @@version;
```

Feature	Local Variables	Global Variables
Definition	Declared inside a stored procedure or a function.	Declared outside a stored procedure or a function.
Scope	Limited to the stored procedure or function.	Available to all stored procedures and functions.
Usage	Used to store temporary data relevant to the procedure.	Used to store data that is shared between procedures and functions.
Example	`DECLARE @variable_name data_type;`	`DECLARE global_variable data_type;`

☺ if i have a User defined scalar function Multiply(a @int, @int b) and i called it like this sql SELECT Multiply(2, 3); , why is that wrong?

Because the user defines scalar function is not a built-in function, so you should write the schema name before the function even if it's a dbo, like this sql SELECT dbo.Multiply(2, 3);

☺ what are synonyms in SQL?

Synonyms are used to provide an alternative name for a table, view, sequence, or stored procedure. **Syntax**: CREATE SYNONYM synonym_name FOR object_name;

☺ what is index in SQL?

An index is a database structure that improves the speed of data retrieval operations on database tables. It allows faster searching, sorting, and filtering of data.

☺ What is the Types of Indexes in SQL?

Clustered Index: A clustered index determines the physical order of data in a table. Each table can have only one clustered index, and it is generally created on the primary key column(s).
Non-Clustered Index: A non-clustered index is a separate structure from the table that contains a sorted list of selected columns. It enhances the performance of searching and filtering operations.
Unique Index: Ensures that all values in the index are unique.
Column-store Index: Optimizes queries for large-scale data warehouses, storing data in a columnar format.
Filtered Index: Applies to a subset of data, useful for filtering rows.
Hash Index: Common in in-memory databases, it uses a hash function to distribute data across a fixed size array.

☺ What is the difference between Clustered and Non-Clustered Index?

Feature	Clustered Index	Non-Clustered Index
Definition	Sorts and stores data rows based on key values	A separate structure that holds pointers to data rows
Data Storage	Data rows stored in order of the index	Index stored separately from data
Number per Table	Only one per table	Multiple non-clustered indexes allowed per table
Performance	Faster for range queries due to sorted data	Useful for quick lookups, may require more I/O
Storage Size	No extra storage required for data	Requires additional storage for the index
Key Usage	Actual data is the index key	Separate key and pointer
Writes	Slower for write operations	Faster for write operations
Read	Faster for read operations	Slower for read operations
Usage	Used in columns frequently appearing in WHERE clauses	Used in columns frequently appearing in JOIN clauses
Example	Primary Key	Secondary Key

☺ should the clustered index always be the primary key?

No, the clustered index does not have to be the primary key. It can be any column that is frequently used in WHERE clauses, but it is recommended to use the primary key as the clustered index.

☺ How do clustered and non-clustered indexes impact query performance?

- Clustered Index: Defines data storage order, improves retrieval for sorted/range queries but can slow down writes.
- Non-Clustered Index: Improves read performance for specific columns, but at the cost of additional storage and slower write operations.

☺ What is the difference between a unique constraint and a unique index?

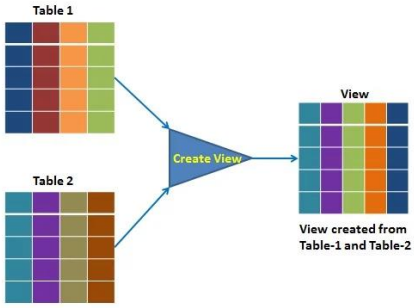
The unique constraint and unique index in SQL Server both ensure that the values in a column or set of columns are unique, but they serve slightly different purposes and have distinct use cases:

1. Unique Constraint:
    - Purpose: Enforces the rule that no duplicate values can be inserted into the column(s). It is a logical constraint that is part of the database schema.
    - Use Case: Primarily used to ensure data integrity. A foreign key constraint can reference a unique constraint.
    - Behavior: It enforces uniqueness at the schema level and integrates with SQL constraints like primary keys and foreign keys.
2. Unique Index:
    - Purpose: Creates a physical index on the column(s) to ensure uniqueness and improve query performance.
    - Use Case: Often used when you want to enforce uniqueness while also optimizing performance for certain queries that rely on indexed data.

- Behavior: It can improve performance by speeding up data retrieval for indexed queries, but its primary function is similar to that of a unique constraint.

☺ What is a view?

A view is a virtual table that is based on the result of a SELECT statement. It is used to simplify complex queries and to provide an additional layer of security.



☺ can we write DML or DDl commands inside a view body?

No, you can't write DML or DDL commands inside a view body, only SELECT statements are allowed.

☺ What is the benefit of using a view?

1. **Simplicity**: Views simplify complex queries by providing a virtual table.
2. **Security**: Views provide an additional layer of security by restricting access to the underlying tables, showing only the necessary columns.
3. **Reusability**: Views can be reused in multiple queries

☺ does the view store data?

No, a view does not store data. It is a virtual table that is based on the result of a SELECT statement.

☺ Do we use variable in views?

No, variables cannot be used directly in SQL views. A view is a stored query, and it doesn't allow the declaration of variables within its definition. If you need to use parameters or dynamic behavior similar to what variables offer, you should consider using a stored procedure or an inline table-valued function (iTVF) instead.

☺ does view increase performance?

No, a view does not increase performance. It is used to simplify complex queries. It's considered a saved subquery that we call when needed (but we can insert into it).

☺ What are the limitations of views?

- ✓ **No Parameters:** Views cannot accept parameters, making them less flexible compared to stored procedures or functions.
- ✓ **Performance**: Views can cause performance degradation, especially when they are complex or involve multiple tables, as they don't store data physically but dynamically fetch it

- ✓ **Dependency on Tables**: If a table used in the view is dropped or altered (e.g., column renamed), the view becomes invalid and throws errors
- ✓ **No ORDER BY Without TOP or FOR XML:** You cannot use an ORDER BY clause in a view unless it is combined with TOP or FOR XML
- ✓ **No Data Modification:** In some cases, updating data through a view (especially one with multiple tables or joins) is not allowed

☺ **can we insert into a view?**

Yes, you can insert into a view if the view is based on a single table and does not contain any aggregate functions. `INSERT INTO view_name VALUES (value1, value2);`

☺ **What is the difference between a view and a function? What is the difference between a table returned from the function and another returned from a view?**

View: A view is a virtual table based on the result of a predefined SQL query. It presents data from one or more tables in a specific format and can simplify complex queries or restrict access to certain data.

- Parameters: Cannot accept parameters. Always returns the same result set based on the underlying query.
- Return Type: Always returns a result set (table-like structure) when queried.
- Usage: Primarily used to simplify complex queries and present data in a specific format. Can be treated like a table in SELECT statements.
- Performance: Generally optimized for read operations, but performance can vary based on the complexity of the underlying query.

Function: A function is a stored program that can accept parameters and return a single value or a result set. Functions encapsulate reusable logic and calculations.

- Parameters: Can accept parameters, allowing for dynamic input and customized output based on those parameters.
- Return Type: Can return a single value (scalar function) or a result set (table-valued function).
- Usage: Used for calculations, data transformations, or encapsulating business logic. Functions can be called in SELECT, WHERE, or other clauses.
- Performance: May have performance implications, especially if used in contexts that require row-by-row processing.

☺ **What is the difference between a view and a materialized view?**

View: A view is a virtual table in SQL, essentially a saved SQL query. It doesn't store data itself but presents the result of the query every time it's accessed. The data is fetched dynamically from the underlying tables at runtime.

Materialized View: A materialized view, on the other hand, stores the result of the query physically on disk, meaning it contains precomputed data. It improves performance, especially for complex queries, as the data is cached and doesn't need to be recomputed for each access. However, materialized views require periodic refreshes to stay up-to-date with the underlying data.

In summary, views are dynamic and provide real-time data, while materialized views are static, storing the result of a query to optimize performance.

☺ Query execution steps in database.

Any query in the database goes through the following steps:

1. **Parsing**: The SQL query is parsed and checked for syntax errors.
2. **Optimization**: The query is optimized using the query optimizer.
3. **Query tree**: The query tree is generated; the engine manipulates the query tree to generate an execution plan.
4. **Execution**: The query is executed, and the result is returned.

So what if we have a block of code that is executed multiple times, we don't need to parse and optimize it every time, so how we can avoid that? by using stored **pprocedures**.

☺ What is a stored procedure?

A stored procedure is a precompiled set of SQL statements that performs a specific task. It can be called and executed multiple times with different parameters by calling the procedure name.

☺ what is the benefit of using a stored procedure?

1. **Performance**: Stored procedures are **precompiled and cached** hence we don't need to parse and optimize them every time they are executed, they only parsed and optimized once at the first execution.
2. **Security**: Stored procedures provide an additional layer of security by restricting access to the underlying tables.
3. **Reusability**: Stored procedures can be reused in multiple queries.
4. **Maintenance**: Stored procedures are easier to maintain as they are stored in the database.
5. **Transaction Management**: Stored procedures can be used to manage transactions.

☺ can we write DDL commands inside a stored procedure?

Yes, unlike functions you can write DDL commands inside a stored procedure.

☺ What is the stored procedure types? What is a trigger?

1. **User-Defined Stored Procedures**: Custom procedures created to perform specific tasks or logic tailored to user needs.
2. **System Stored Procedures**: Built-in procedures provided by SQL Server for administrative and maintenance tasks. Provided by the database management system, start with (sp_) e.g. sp_bindrule.
3. **Temporary Stored Procedures**: Procedures that exist only during a session, useful for temporary tasks.
4. **Extended Stored Procedures**: These allow interaction with external programs and can execute system-level commands.
5. **SQL Triggers:** Triggers are special types of stored procedures that automatically

execute in response to specific events on a table, such as INSERT, UPDATE, or DELETE actions. They help maintain data integrity and enforce business rules without requiring explicit calls

☺ what are the types of triggers?

1. **Data Manipulation Language (DML) Triggers (Table Level)**: Automatically execute in response to data modification events such as INSERT, UPDATE, or DELETE. They can be further divided into:
    - **AFTER Triggers**: Execute after the data modification occurs.
    - **INSTEAD OF Triggers**: Execute in place of the data modification event, allowing for custom logic.
2. **Data Definition Language (DDL) Triggers (Database Level)**: Respond to changes in the structure of the database, such as CREATE, ALTER, or DROP commands. These are useful for auditing and enforcing rules.
3. **Logon Triggers (Server level)**: Execute in response to logon events, allowing for actions such as logging and controlling access based on user credentials.

These triggers enhance data integrity and automate processes within the database.

☺ what is the benefit of using a trigger?

Using triggers in SQL offers several significant benefits:

1. **Automated Actions:** Triggers can automatically execute predefined actions in response to specific events, such as data modifications, enhancing efficiency
2. **Data Integrity:** They help enforce complex business rules and constraints, ensuring data consistency and integrity across the database
3. **Security**: Triggers can provide an additional layer of security by allowing limited access to sensitive data, only permitting certain actions under specified conditions
4. **Reduced Code Duplication:** By encapsulating logic within triggers, you can minimize code redundancy in your application, leading to cleaner and more maintainable code
5. **Auditing**: Triggers can be used to log changes to the database.

☺ What is Data Integrity?

Data Integrity is the assurance of accuracy and consistency of data over its entire life-cycle and is a critical aspect of the design, implementation, and usage of any system which stores, processes, or retrieves data. It also defines integrity constraints to enforce business rules on the data when it is entered into an application or a database.

☺ What is referential integrity?

Referential integrity refers to the consistency that must be maintained between primary and foreign keys, i.e. every foreign key value must have a corresponding primary key value.

☺ What is data integrity like domain integrity, entity integrity, referential integrity?

Data integrity ensures the accuracy and consistency of data in a database. It encompasses several types:

1. **Domain Integrity**: This type enforces valid entries for individual columns based on predefined rules or constraints. It ensures that data types, formats, and ranges are adhered to for each field in a database table.
2. **Entity Integrity**: These principal mandates that each table must have a primary key, which uniquely identifies each row. No two rows can have the same primary key value, and it cannot be null.
3. **Referential Integrity**: This ensures that relationships between tables remain consistent. Specifically, it dictates that a foreign key must either match an existing primary key in another table or be null, maintaining the accuracy of the links between related tables.

These integrity constraints are crucial for maintaining the reliability of data within databases.

☺ Table level triggers?

1. **Instead of Trigger**: Fired instead of the triggering action.
2. **After Trigger**: Fired after the triggering action.
3. **For Trigger**: Fired for each row affected by the triggering action.

☺ What is the difference between a stored procedure and a function?

Feature	Stored Procedure	Function
Return Type	Can return multiple values or none	Must return a single value
Purpose	Performs actions (e.g., modify data)	Primarily for computations and data retrieval
Execution	Can be executed via EXEC statement	Can be used in SQL expressions
Transaction Control	Can manage transactions (COMMIT/ROLLBACK)	Cannot manage transactions
Side Effects	Can have side effects (e.g., modifying tables)	No side effects, purely functional
Input Parameters	Supports input and output parameters	Supports only input parameters
Parsing and Optimization	Once	Every time it's called
Parameters	Yes	Yes
DDL Commands	Yes	No
Transaction Management (TCL)	Yes	No
Return	May return zero or more values	Must return a value
Calling	Using EXEC	Using SELECT
Performance	Faster	Slower
Error Handling	Yes, using TRY...CATCH	No
Transaction Restrictions	Can contain transactions	Transactions are not allowed

☺ What is the difference between triggers and query with alert

Aspect	Triggers	Alerts
Definition	**A set of SQL instructions that execute automatically in response to events (e.g., INSERT, UPDATE, DELETE). Operates synchronously with the event.**	**A notification mechanism that runs queries at intervals or in real-time based on conditions. Does not modify data directly.**

Purpose	**Enforces business rules, maintains data integrity, or logs changes automatically during database modifications.**	**Primarily used for monitoring, notifying users when specific thresholds or events are met, without altering the data.**
Execution	**Executes synchronously, potentially affecting the outcome of a data modification operation.**	**Executes asynchronously, typically notifying users without affecting database operations.**

☺ What is the difference between a stored procedure and a trigger?

Feature	Stored Procedure	Trigger
Invocation	Explicitly called by a user or application	Automatically executed in response to an event (e.g., INSERT, UPDATE, DELETE) [3]
Parameters	Can accept parameters	Cannot accept parameters [5]
Execution	Executes on demand	Executes automatically without direct invocation [4]
Return Type	Can return values (zero or more)	Does not return values [2]
Transaction Management	Supports transaction management	Cannot manage transactions [6]
Purpose	Performs operations based on user commands	Enforces rules and maintains data integrity automatically [1]

☺ Difference Group by Rollup and Group by Cube.

Feature	GROUP BY ROLLUP	GROUP BY CUBE
Definition	Generates subtotal rows along a hierarchy of columns	Generates subtotals for all combinations of columns
Output	Includes hierarchical summaries (e.g., by year, then by month)	Includes all possible combinations of values
Use Case	Ideal for reporting totals along a single hierarchy	Useful for multidimensional analysis
Example	`SELECT department, SUM(sales) FROM sales GROUP BY ROLLUP(department)`	`SELECT department, region, SUM(sales) FROM sales GROUP BY CUBE(department, region)`

☺ Difference Table Valued and Multi Statement Function.

Feature	Table-Valued Functions (TVF)	Multi-Statement Table-Valued Functions (MSTVF)
Definition	A user-defined function that returns a table.	A TVF that allows multiple SQL statements to build the result set.
Structure	Typically consists of a single SELECT statement.	Can contain multiple SQL statements, including procedural logic.
Performance	Generally more efficient due to a single query.	May be less efficient because it uses intermediate table variables.
Use Case	Best for simple queries returning rows.	Ideal for complex logic requiring multiple operations.
Example Syntax	`CREATE FUNCTION func_name() RETURNS TABLE AS RETURN (SELECT ...)`	`CREATE FUNCTION func_name() RETURNS @table_var TABLE (...) AS BEGIN ... END` [2] [3]

☺ Difference between rollup cube grouping sets pivot table

Feature	ROLLUP	CUBE	GROUPING SETS	PIVOT Table
Definition	Generates subtotals and grand totals in a hierarchical way.	Produces all possible combinations of aggregations.	Defines specific groups of aggregations.	Transforms unique values from one column into multiple columns.
Purpose	Used for summarizing data hierarchically.	Useful for comprehensive analysis across multiple dimensions.	Provides flexibility in defining aggregation needs.	Simplifies data presentation for analysis.
Execution	Executes synchronously, affecting data modification.	Executes synchronously, providing detailed aggregates.	Executes synchronously for specified combinations.	Executes separately, restructuring the result set.
Example	`SELECT year, month, SUM(sales) FROM sales GROUP BY ROLLUP(year, month);`	`SELECT region, product, SUM(sales) FROM sales GROUP BY CUBE(region, product);`	`SELECT year, product, SUM(sales) FROM sales GROUP BY GROUPING SETS((year), (product));`	`SELECT * FROM (SELECT product, region, sales FROM sales) PIVOT (SUM(sales) FOR region IN ('East', 'West')) AS pivot_table;`

☺ Difference Table Variable and Temporary Table

Feature	Table Variable	Temporary Table
Definition	A variable that holds a table structure in memory.	A physical table created in the `tempdb` database.
Scope	Limited to the batch, stored procedure, or function.	Exists until the connection is closed or explicitly dropped.
Storage	Stored in memory (in SQL Server 2000 and later).	Stored on disk with associated statistics.
Indexes	Can have primary key and unique constraints, but no explicit indexing.	Can have indexes, which can enhance performance.
Performance	Generally faster for small datasets due to less overhead.	Better for larger datasets and complex queries due to indexing capabilities.
Usage	Ideal for smaller datasets and quick operations.	Suitable for larger datasets requiring operations like joins and complex queries [1][3][5].

☺ Difference triggers and indexes

Aspect	Triggers	Indexes
Definition	Procedural code that automatically executes in response to events (e.g., INSERT, UPDATE, DELETE) to enforce business rules, validation, or auditing.	Database objects that improve data retrieval speed by creating a structure (usually B-tree) for faster searches.
Usage	Responds automatically to events without direct invocation and can enforce complex business logic.	Improves performance of queries, especially SELECT statements, by allowing quick row location.
Use Cases	Useful for logging changes to records and enforcing constraints that cannot be easily managed by integrity constraints (e.g., cascading updates).	Employed in large tables to enhance performance for frequently queried columns and fields involved in JOIN operations and WHERE clauses.

☺ What is Cursor? How to use a Cursor?

A database cursor is a control structure that allows for the traversal of records in a database. Cursors, in addition, facilitates processing a er traversal, such as retrieval, addition, and deletion of database records. They can be viewed as a pointer to one row in a set of rows.
Working with SQL Cursor:

1) DECLARE a cursor a er any variable declaration. The cursor declaration must always be associated with a SELECT Statement.
2) Open cursor to initialize the result set. The OPEN statement must be called before fetching rows from the result set.
3) **FETCH** statement to retrieve and move to the next row in the result set.
4) Call the **CLOSE** statement to deactivate the cursor.
5) Finally use the DEALLOCATE statement to delete the cursor definition and release the associated resources.

```
DECLARE @name VARCHAR(50)   /* Declare All Required Variables */
DECLARE db_cursor CURSOR FOR   /* Declare Cursor Name*/
SELECT name
FROM myDB.students
WHERE parent_name IN ('Sara', 'Ansh')
OPEN db_cursor   /* Open cursor and Fetch data into @name */
FETCH next
FROM db_cursor
INTO @name
CLOSE db_cursor   /* Close the cursor and deallocate the resources */
DEALLOCATE db_cursor
```

☺ What is OLTP? What are the differences between OLTP and OLAP?

OLTP (Online Transaction Processing): A class of software applications designed to support transaction-oriented programs.

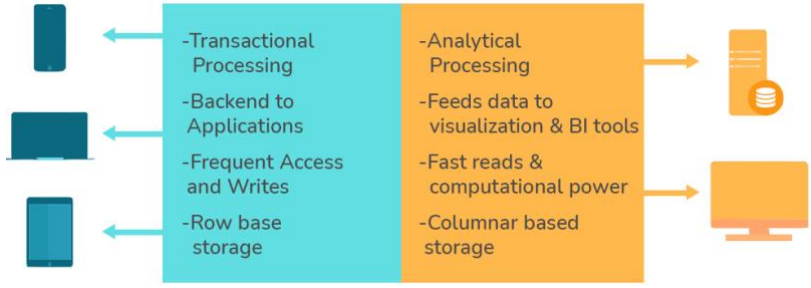
- **Key Attributes**:
  - Maintains **concurrency** to handle multiple transactions simultaneously.
  - Often follows a **decentralized architecture** to prevent single points of failure.
- **User Interaction**: Designed for a large audience conducting **short transactions**.
- **Query Characteristics**: Queries are generally **simple**, require **fast response times**, and return relatively few records.
- **Performance Measure**: Effectiveness is often measured by the **number of transactions per second**.

OLAP (Online Analytical Processing): A class of software programs characterized by a low frequency of online transactions.

- **Query Characteristics**: Involves **complex queries** with multiple aggregations.
- **Usage**: Widely used for **data mining** and maintaining **aggregated historical data**, typically

in **multi-dimensional schemas**.

- **Performance Measure**: Effectiveness relies heavily on **response time**.



☺ What is the difference between a database and a schema?

A **database** is a container that holds multiple objects such as tables, views, indexes, and procedures. It represents a logical grouping of related data.
A **schema** is a container within a database that holds objects and defines their ownership. It provides a way to organize and manage database objects.
Essentially, the database stores the data, while the schema defines the structure of that data

☺ What is Collation? What are the different types of Collation Sensitivity?

Collation refers to a set of rules that determine how data is sorted and compared. Rules defining the correct character sequence are used to sort the character data. It incorporates options for specifying case sensitivity, accent marks, kana character types, and character width. Below are the different types of collation sensitivity:

- Case sensitivity: A and a are treated differently.
- Accent sensitivity: a and á are treated differently.
- Kana sensitivity: Japanese kana characters Hiragana and Katakana are treated differently.
- Width sensitivity: Same character represented in single-byte (half-width) and double-byte (full-width) are treated differently.

☺ What is a Recursive Stored Procedure?

A stored procedure that calls itself until a boundary condition is reached, is called a recursive stored procedure. This recursive function helps the programmers to deploy the same set of code several times as and when required. Some SQL programming languages limit the recursion depth to prevent an infinite loop of procedure calls from causing a stack overflow, which slows down the system and may lead to system crashes.

☺ How to create empty tables with the same structure as another table?

```
SELECT * INTO Students_copy
FROM Students WHERE 1 = 2;
```

☺ What is Pattern Matching in SQL?

SQL pattern matching provides for pattern search in data if you have no clue as to what that word should be. This kind of SQL query uses wildcards to match a string pattern, rather than writing the exact word. The LIKE operator is used in conjunction with SQL Wildcards to fetch the required information.

- **Using the % wildcard to perform a simple search**

The % wildcard matches zero or more characters of any type and can be used to define wildcards both before and a er the pattern. Search a student in your database with first name beginning with the letter K:

```
SELECT *
FROM students
WHERE first_name LIKE 'K%'
```

- **Omitting the patterns using the NOT keyword**

Use the NOT keyword to select records that don't match the pattern. This query returns all students whose first name does not begin with K.

```
SELECT *
FROM students
WHERE first_name NOT LIKE 'K%'
```

- **Matching a pattern anywhere using the % wildcard twice**

Search for a student in the database where he/she has a K in his/her first name.

```
SELECT *
FROM students
WHERE first_name LIKE '%Q%'
```

- **Using the _ wildcard to match pattern at a specific position**

The _ wildcard matches exactly one character of any type. It can be used in conjunction with % wildcard. This query fetches all students with letter K at the third position in their first name.

```
SELECT *
FROM students
WHERE first_name LIKE '__K%'
```

- **Matching patterns for a specific length**

The _ wildcard plays an important role as a limitation when it matches exactly one character. It limits the length and position of the matched results. For example –

```
SELECT *   /* Matches first names with three or more letters */
FROM students
WHERE first_name LIKE '___%'

SELECT *   /* Matches first names with exactly four characters */
FROM students
WHERE first_name LIKE '____'
```

☺ **Explain the difference between a database & a data warehouse.**

Database

- **Purpose**: Designed to store and manage **real-time transactional data** for operational tasks.
- **Functionality**: Optimized for frequent **reading, writing, updating, and deleting** of data.
- **Use Cases**: Supports everyday applications like **customer orders** and **inventory tracking**.
- **Data Focus**: Primarily handles **current data** for operational purposes.

Data Warehouse

- **Purpose**: Designed for storing and analyzing **large volumes of historical data** from various sources.
- **Functionality**: Supports **complex queries** for business intelligence and reporting.

- **Optimization**: Optimized for **reading and querying** large datasets.
- **Data Focus**: Focuses on **long-term data analysis** rather than real-time transactions.

☺ What are your methods for ensuring data accuracy & integrity?

To ensure data accuracy and integrity, a combination of technical and procedural methods is used:

1. Data Validation: Implement rules to check for correct data types, formats, and ranges at the point of data entry, ensuring only valid data is accepted.
2. Access Control: Restrict data access to authorized personnel only to prevent unauthorized modifications.
3. Audit Trails: Keep detailed logs of data changes and who made them, enabling traceability and accountability.
4. Data Backups: Regularly back up data to prevent loss or corruption from system failures.
5. Encryption: Use encryption for data in transit and at rest to protect against unauthorized access or tampering.
6. Data Quality Tools: Use tools to detect and correct errors automatically, ensuring data remains reliable over time

☺ What is an execution plan? When would you use it? How would you view the execution plan?

An execution plan is basically a road map that graphically or textually shows the data retrieval methods chosen by the SQL Server query optimizer for a stored procedure or ad- hoc query and is a very useful tool for a developer to understand the performance characteristics of a query or stored procedure since the plan is the one that SQL Server will place in its cache and use to execute the stored procedure or query. From within Query Analyzer is an option called "Show Execution Plan" (located on the Query drop-down menu). If this option is turned on it will display query execution plan in separate window when query is ran again.

☺ What is a Scheduled Job or a Scheduled Task?

In SQL, a Scheduled Job or Scheduled Task is an automated process set to run at specific intervals or times without manual intervention. These jobs are typically created to perform repetitive tasks such as:

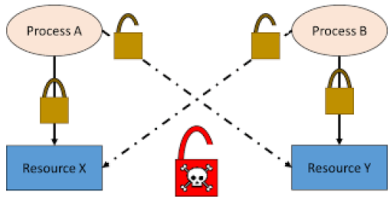
- Running SQL queries
- Database backups
- Data import/export
- Database maintenance (e.g., reindexing or clearing logs)

In SQL Server, this can be managed via the SQL Server Agent, which allows you to schedule jobs through SQL Server Management Studio (SSMS), Transact-SQL, or SQL Server Management Objects (SMO). You can define the job, its steps, and its schedule.

☺ SQL Server: How do you handle slow-running queries in a production environment?

To handle slow-running queries in a SQL Server production environment, several steps can help identify and optimize query performance:

1. Analyze the Execution Plan: Review the query execution plan in SQL Server Management Studio (SSMS) to identify performance bottlenecks, such as table scans or inefficient joins.
2. Index Optimization: Ensure that appropriate indexes are in place on the columns used in JOIN, WHERE, and ORDER BY clauses. Missing or outdated indexes can slow down queries.
3. Use the Query Store: SQL Server's Query Store can track query performance over time, allowing you to pinpoint which queries have regressed in speed.
4. Avoid SELECT *: Retrieve only necessary columns instead of using SELECT *. This reduces the data load and improves query performance.
5. Optimize Joins and Subqueries: Simplify joins or break complex subqueries into smaller queries for better performance. Consider rewriting queries where necessary.
6. Monitor Server Performance: Check for server-level resource issues (CPU, memory, disk I/O) that could affect query speed. Optimizing server performance can significantly improve slow queries.



☺ What is a deadlock? How do you detect and resolve deadlocks in SQL Server?

A deadlock occurs when two or more transactions are waiting for each other to release resources, resulting in a circular dependency. As a result, none of the transactions can proceed, and the system may become unresponsive. **Deadlocks in SQL Server** occur when two or more processes are waiting for each other to release locks, causing them to be stuck. Here's how to detect and resolve them:

Detection:

1. Deadlock Monitor: SQL Server's lock monitor automatically detects deadlocks by periodically checking running tasks for conflicts.
2. Trace Flags: Use Trace Flag 1222 or Trace Flag 1204 to capture detailed deadlock information, including a deadlock graph, which is logged in the SQL Server error log.
3. Extended Events: Use SQL Server Extended Events to capture and analyze deadlock occurrences.
4. SQL Server Profiler: This can also be used to trace deadlocks by setting up deadlock monitoring.

Resolution:

1. Terminate one of the processes: SQL Server automatically chooses one of the processes (the "deadlock victim") to terminate and rolls back its transaction.
2. Index Optimization: Create or modify indexes to reduce the likelihood of deadlock-prone queries locking overlapping resources.
3. Lock Hints: Use lock hints like WITH (NOLOCK) on SELECT queries to reduce lock contention, though it should be used with caution as it can lead to dirty reads.

4. Transaction Management: Keep transactions short and consistent in locking order to reduce deadlock opportunities.

☺ What is the difference between a temporary table and a table variable?

Temporary Table

- Scope: Available for the entire session or connection; can be used across multiple batches until explicitly dropped or the session ends.
- Performance: Generally slower for small datasets but can be indexed for improved performance with larger datasets.
- Transactions: Fully supports transactions, allowing changes to be committed or rolled back.
- Indexing: Allows the creation of multiple indexes.

Table Variable

- Scope: Limited to the specific batch or stored procedure where it is declared.
- Performance: Often faster for small datasets but does not support non-clustered indexes.
- Transactions: Does not fully participate in transactions.
- Indexing: Limited to primary key or unique constraints only.

☺ What is the purpose of the CONSTRAINT keyword in the ALTER TABLE statement?

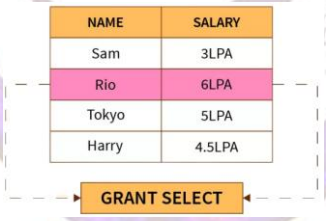
The CONSTRAINT keyword in the ALTER TABLE statement is used to add, modify, or drop constraints on columns within an existing table.

☺ What is the purpose of ALL-keyword in SQL?

The **CASCADE DELETE** constraint is used to automatically delete related rows in child tables when a row in the parent table is deleted.

☺ What is the purpose of the GRANT statement?

The **GRANT** statement is used in SQL to assign specific privileges to users or roles, allowing them to perform certain actions on database objects like tables, views, procedures, or sequences. Privileges that can be granted include operations such as **SELECT**, **INSERT**, **UPDATE**, **DELETE**, and more. By using the GRANT statement, administrators can control which users have access to specific database resources and define the extent of their permissions. For example, a user might be granted permission to view data in a table (SELECT), but not to alter or delete it



☺ What is the purpose of the CASE statement?

The CASE statement is used to perform conditional logic in SQL queries. It allows you to return different values based on specified conditions.

☺ What is the purpose of the COALESCE function?

The COALESCE function returns the first non-null expression from a list of expressions. It is often used to handle null values effectively.

☻ What is a correlated update?

A correlated update is an update statement that refers to a column from the same table in a subquery. It updates values based on the result of the subquery for each row. A **correlated update** is an SQL UPDATE statement that includes a correlated subquery in its WHERE clause. This means the subquery references columns from the outer query, typically the same table being updated. In a correlated update, each row from the outer query is processed individually, with the subquery being evaluated for each row, allowing updates based on conditions that involve other rows or tables**. For example,** a correlated update can be used to update a column in one table based on the values from another table where there's a relationship between the two tables.

☻ How do you manage and optimize large databases in SQL Server?

Managing and optimizing large databases in SQL Server involves several key techniques to ensure efficient performance and scalability:

1. Sharding and Partitioning:
    - Partition large tables into smaller, manageable segments to improve performance and make queries faster. Partitioning allows data to be accessed more efficiently.
2. Indexing:
    - Create and reorganize indexes to optimize read and query performance. Properly indexed columns can dramatically speed up data retrieval, especially for large datasets.
3. Query Optimization:
    - Avoid using SELECT *. Retrieve only the required columns to reduce the amount of data being processed and returned.
    - Optimize your JOIN operations and use window functions to limit the data returned in queries.
4. Backup Strategies: Use incremental or differential backups to manage backups more efficiently without affecting database performance.
5. Monitoring and Profiling: Monitor database performance and identify slow queries using SQL Server Profiler or Execution Plans to focus on the bottlenecks and optimize them

☻ How do you design and implement efficient indexing strategies in large-scale SQL databases?

Designing and implementing efficient indexing strategies for large-scale SQL databases requires a careful balance between query performance improvements and the impact on write operations. Here are key strategies:

1. Understand Query Patterns: Analyze queries to identify the most frequent and expensive operations. Focus on columns used in WHERE, JOIN, GROUP BY, and ORDER BY clauses, as these will benefit most from indexing.

2. Use the Right Index Types:

- Clustered Indexes: Use clustered indexes on columns that determine the physical order of the data, like primary keys or columns frequently used in range queries (e.g., date fields).

- Non-Clustered Indexes: Use non-clustered indexes on columns frequently involved in search conditions but do not determine data order. These indexes contain pointers to the actual data, allowing faster retrieval of results.

3. Index Selective Columns: Create indexes on columns with high cardinality, meaning columns with many distinct values (e.g., user IDs or email addresses). Avoid indexing columns with low cardinality, such as Boolean or status fields.

4. Composite Indexes: Use composite indexes (multi-column indexes) where queries frequently filter or sort on multiple columns. Ensure the columns are ordered based on the frequency of use in queries (e.g., if a query filters on columns A and B, index (A, B), not (B, A)).

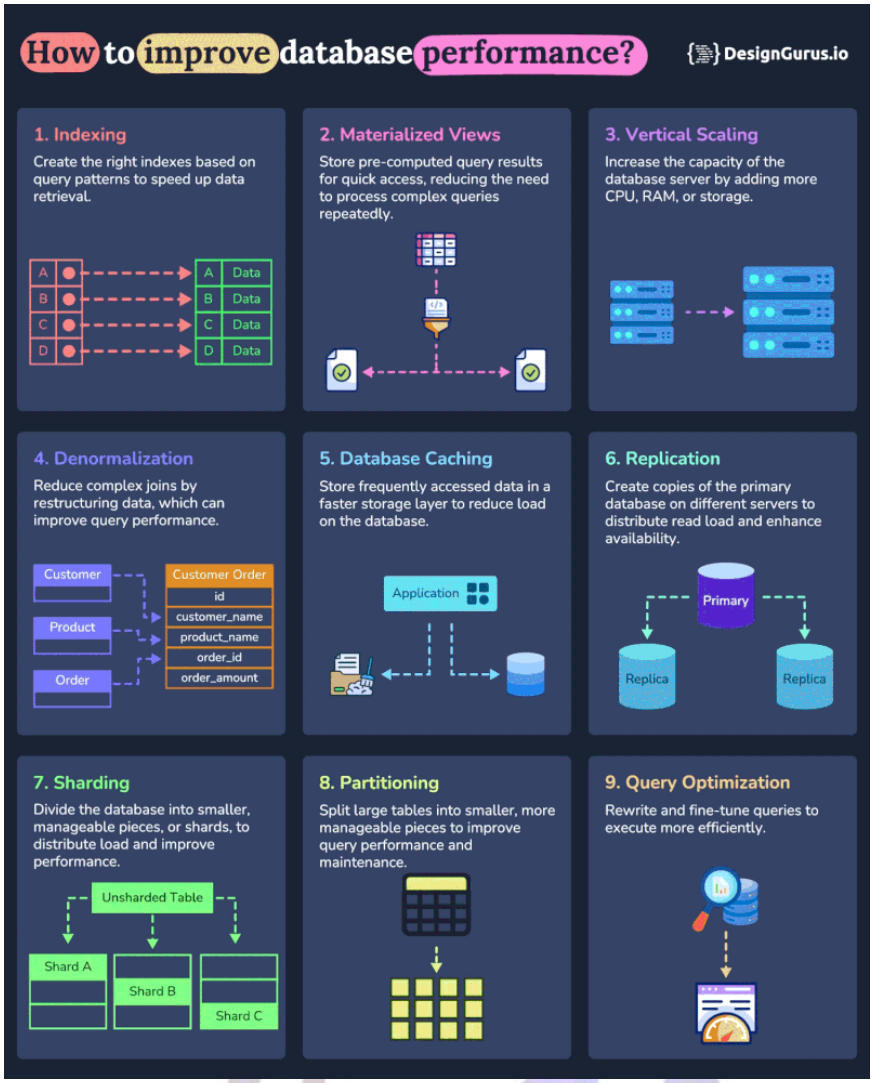
5. Monitor and Maintain Indexes: Regularly reorganize and rebuild indexes to remove fragmentation, especially on large tables. SQL Server offers built-in functions to automate index maintenance.

6. Avoid Over-Indexing: Be mindful of the overhead. Each index slows down inserts, updates, and deletes because the index needs to be updated as well. Limit the number of indexes to what's necessary for query performance.

7. Use Covering Indexes: Create covering indexes that include all columns used in a query (SELECT, WHERE, JOIN) to avoid additional lookups to the table data, reducing query execution time.

☺ How to improve database performance?

1. Indexing: Create the right indexes based on query patterns to speed up data retrieval.
2. Materialized Views: Store pre-computed query results for quick access, reducing the need to process complex queries repeatedly.
3. Vertical Scaling: Increase the capacity of the database server by adding more CPU, RAM, or storage.
4. Denormalization: Reduce complex joins by restructuring data, which can improve query performance.
5. Database Caching: Store frequently accessed data in a faster storage layer to reduce load on the database.
6. Replication: Create copies of the primary database on different servers to distribute read load and enhance availability.
7. Sharding: Divide the database into smaller, manageable pieces, or shards, to distribute load and improve performance.
8. Partitioning: Split large tables into smaller, more manageable pieces to improve query performance and maintenance.
9. Query Optimization: Rewrite and fine-tune queries to execute more efficiently.
10. Use of Appropriate Data Types: Select the most efficient data types for each column to save space and speed up processing.
11. Limiting Indexes: Avoid excessive indexing, which can slow down write operations; use indexes judiciously.
12. Archiving Old Data: Move infrequently accessed data to an archive to keep the active database smaller and faster.



References:

1. ITI notes DR Rami
2. https://www.interviewbit.com/
3. linkedin.com/in/arslanahmad
4. https://learnsql.com/tags/cheat-sheet/

SQL Basics Cheat Sheet

LearnSQL.com

SQL

SQL, or *Structured Query Language*, is a language to talk to databases. It allows you to select specific data and to build complex reports. Today, SQL is a universal language of data. It is used in practically all technologies that process data.

SAMPLE DATA

COUNTRY

id	name	population	area
1	France	66600000	640680
2	Germany	80700000	357000
...	...	...	...

CITY

id	name	country_id	population	rating
1	Paris	1	2243000	5
2	Berlin	2	3460000	3
...	...	...	...	...

QUERYING SINGLE TABLE

Fetch all columns from the `country` table:

```
SELECT *
FROM country;
```

Fetch `id` and `name` columns from the `city` table:

```
SELECT id, name
FROM city;
```

Fetch city names sorted by the `rating` column in the default ASCending order:

```
SELECT name
FROM city
ORDER BY rating [ASC];
```

Fetch city names sorted by the `rating` column in the DESCending order:

```
SELECT name
FROM city
ORDER BY rating DESC;
```

ALIASES

COLUMNS

```
SELECT name AS city_name
FROM city;
```

TABLES

```
SELECT co.name, ci.name
FROM city AS ci
JOIN country AS co
  ON ci.country_id = co.id;
```

FILTERING THE OUTPUT

COMPARISON OPERATORS

Fetch names of cities that have a rating above 3:

```
SELECT name
FROM city
WHERE rating > 3;
```

Fetch names of cities that are neither Berlin nor Madrid:

```
SELECT name
FROM city
WHERE name != 'Berlin'
  AND name != 'Madrid';
```

TEXT OPERATORS

Fetch names of cities that start with a 'P' or end with an 's':

```
SELECT name
FROM city
WHERE name LIKE 'P%'
   OR name LIKE '%s';
```

Fetch names of cities that start with any letter followed by 'ublin' (like Dublin in Ireland or Lublin in Poland):

```
SELECT name
FROM city
WHERE name LIKE '_ublin';
```

OTHER OPERATORS

Fetch names of cities that have a population between 500K and 5M:

```
SELECT name
FROM city
WHERE population BETWEEN 500000 AND 5000000;
```

Fetch names of cities that don't miss a rating value:

```
SELECT name
FROM city
WHERE rating IS NOT NULL;
```

Fetch names of cities that are in countries with IDs 1, 4, 7, or 8:

```
SELECT name
FROM city
WHERE country_id IN (1, 4, 7, 8);
```

QUERYING MULTIPLE TABLES

INNER JOIN

`JOIN` (or explicitly **`INNER JOIN`**) returns rows that have matching values in both tables.

```
SELECT city.name, country.name
FROM city
[INNER] JOIN country
  ON city.country_id = country.id;
```

CITY			COUNTRY	
id	name	country_id	id	name
1	Paris	1	1	France
2	Berlin	2	2	Germany
3	Warsaw	4	3	Iceland

LEFT JOIN

`LEFT JOIN` returns all rows from the left table with corresponding rows from the right table. If there's no matching row, **`NULL`**s are returned as values from the second table.

```
SELECT city.name, country.name
FROM city
LEFT JOIN country
  ON city.country_id = country.id;
```

CITY			COUNTRY	
id	name	country_id	id	name
1	Paris	1	1	France
2	Berlin	2	2	Germany
3	Warsaw	4	NULL	NULL

RIGHT JOIN

`RIGHT JOIN` returns all rows from the right table with corresponding rows from the left table. If there's no matching row, **`NULL`**s are returned as values from the left table.

```
SELECT city.name, country.name
FROM city
RIGHT JOIN country
  ON city.country_id = country.id;
```

CITY			COUNTRY	
id	name	country_id	id	name
1	Paris	1	1	France
2	Berlin	2	2	Germany
NULL	NULL	NULL	3	Iceland

FULL JOIN

`FULL JOIN` (or explicitly **`FULL OUTER JOIN`**) returns all rows from both tables – if there's no matching row in the second table, **`NULL`**s are returned.

```
SELECT city.name, country.name
FROM city
FULL [OUTER] JOIN country
  ON city.country_id = country.id;
```

CITY			COUNTRY	
id	name	country_id	id	name
1	Paris	1	1	France
2	Berlin	2	2	Germany
3	Warsaw	4	NULL	NULL
NULL	NULL	NULL	3	Iceland

CROSS JOIN

`CROSS JOIN` returns all possible combinations of rows from both tables. There are two syntaxes available.

```
SELECT city.name, country.name
FROM city
CROSS JOIN country;

SELECT city.name, country.name
FROM city, country;
```

CITY			COUNTRY	
id	name	country_id	id	name
1	Paris	1	1	France
1	Paris	1	2	Germany
2	Berlin	2	1	France
2	Berlin	2	2	Germany

NATURAL JOIN

`NATURAL JOIN` will join tables by all columns with the same name.

```
SELECT city.name, country.name
FROM city
NATURAL JOIN country;
```

CITY			COUNTRY	
country_id	id	name	name	id
6	6	San Marino	San Marino	6
7	7	Vatican City	Vatican City	7
5	9	Greece	Greece	9
10	11	Monaco	Monaco	10

`NATURAL JOIN` used these columns to match rows: **`city.id, city.name, country.id, country.name`**
`NATURAL JOIN` is very rarely used in practice.

SQL Basics Cheat Sheet

LearnSQL.com

AGGREGATION AND GROUPING

GROUP BY **groups** together rows that have the same values in specified columns. It computes summaries (aggregates) for each unique combination of values.

CITY

id	name	country_id
1	Paris	1
101	Marseille	1
102	Lyon	1
2	Berlin	2
103	Hamburg	2
104	Munich	2
3	Warsaw	4
105	Cracow	4

→

CITY

country_id	count
1	3
2	3
4	2

AGGREGATE FUNCTIONS

- **avg(**expr**)** – average value for rows within the group
- **count(**expr**)** – count of values for rows within the group
- **max(**expr**)** – maximum value within the group
- **min(**expr**)** – minimum value within the group
- **sum(**expr**)** – sum of values within the group

EXAMPLE QUERIES

Find out the number of cities:

```
SELECT COUNT(*)
FROM city;
```

Find out the number of cities with non-null ratings:

```
SELECT COUNT(rating)
FROM city;
```

Find out the number of distinctive country values:

```
SELECT COUNT(DISTINCT country_id)
FROM city;
```

Find out the smallest and the greatest country populations:

```
SELECT MIN(population), MAX(population)
FROM country;
```

Find out the total population of cities in respective countries:

```
SELECT country_id, SUM(population)
FROM city
GROUP BY country_id;
```

Find out the average rating for cities in respective countries if the average is above 3.0:

```
SELECT country_id, AVG(rating)
FROM city
GROUP BY country_id
HAVING AVG(rating) > 3.0;
```

SUBQUERIES

A subquery is a query that is nested inside another query, or inside another subquery. There are different types of subqueries.

SINGLE VALUE

The simplest subquery returns exactly one column and exactly one row. It can be used with comparison operators =, <, <=, >, or >=.

This query finds cities with the same rating as Paris:

```
SELECT name FROM city
WHERE rating = (
    SELECT rating
    FROM city
    WHERE name = 'Paris'
);
```

MULTIPLE VALUES

A subquery can also return multiple columns or multiple rows. Such subqueries can be used with operators IN, EXISTS, ALL, or ANY.

This query finds cities in countries that have a population above 20M:

```
SELECT name
FROM city
WHERE country_id IN (
    SELECT country_id
    FROM country
    WHERE population > 20000000
);
```

CORRELATED

A correlated subquery refers to the tables introduced in the outer query. A correlated subquery depends on the outer query. It cannot be run independently from the outer query.

This query finds cities with a population greater than the average population in the country:

```
SELECT *
FROM city main_city
WHERE population > (
    SELECT AVG(population)
    FROM city average_city
    WHERE average_city.country_id = main_city.country_id
);
```

This query finds countries that have at least one city:

```
SELECT name
FROM country
WHERE EXISTS (
    SELECT *
    FROM city
    WHERE country_id = country.id
);
```

SET OPERATIONS

Set operations are used to combine the results of two or more queries into a single result. The combined queries must return the same number of columns and compatible data types. The names of the corresponding columns can be different.

CYCLING

id	name	country
1	YK	DE
2	ZG	DE
3	WT	PL
...	...	...

SKATING

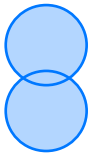
id	name	country
1	YK	DE
2	DF	DE
3	AK	PL
...	...	...

UNION

UNION combines the results of two result sets and removes duplicates. UNION ALL doesn't remove duplicate rows.

This query displays German cyclists together with German skaters:

```
SELECT name
FROM cycling
WHERE country = 'DE'
UNION / UNION ALL
SELECT name
FROM skating
WHERE country = 'DE';
```

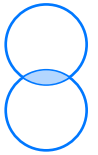


INTERSECT

INTERSECT returns only rows that appear in both result sets.

This query displays German cyclists who are also German skaters at the same time:

```
SELECT name
FROM cycling
WHERE country = 'DE'
INTERSECT
SELECT name
FROM skating
WHERE country = 'DE';
```

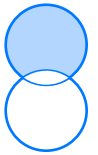


EXCEPT

EXCEPT returns only the rows that appear in the first result set but do not appear in the second result set.

This query displays German cyclists unless they are also German skaters at the same time:

```
SELECT name
FROM cycling
WHERE country = 'DE'
EXCEPT / MINUS
SELECT name
FROM skating
WHERE country = 'DE';
```

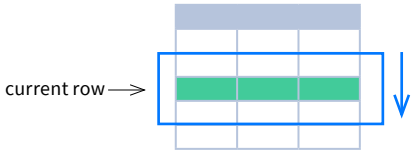


SQL Window Functions Cheat Sheet

LearnSQL.com

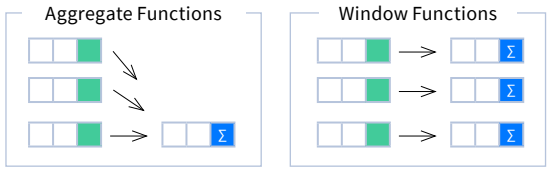
WINDOW FUNCTIONS

compute their result based on a sliding window frame, a set of rows that are somehow related to the current row.



AGGREGATE FUNCTIONS VS. WINDOW FUNCTIONS

unlike aggregate functions, window functions do not collapse rows.



SYNTAX

```
SELECT city, month,
    sum(sold) OVER (
        PARTITION BY city
        ORDER BY month
        RANGE UNBOUNDED PRECEDING) total
FROM sales;
```

```
SELECT <column_1>, <column_2>,
    <window_function>() OVER (
        PARTITION BY <...>
        ORDER BY <...>
        <window_frame>) <window_column_alias>
FROM <table_name>;
```

Named Window Definition

```
SELECT country, city,
    rank() OVER country_sold_avg
FROM sales
WHERE month BETWEEN 1 AND 6
GROUP BY country, city
HAVING sum(sold) > 10000
WINDOW country_sold_avg AS (
    PARTITION BY country
    ORDER BY avg(sold) DESC)
ORDER BY country, city;
```

```
SELECT <column_1>, <column_2>,
    <window_function>() OVER <window_name>
FROM <table_name>
WHERE <...>
GROUP BY <...>
HAVING <...>
WINDOW <window_name> AS (
    PARTITION BY <...>
    ORDER BY <...>
    <window_frame>)
ORDER BY <...>;
```

PARTITION BY, ORDER BY, and window frame definition are all optional.

LOGICAL ORDER OF OPERATIONS IN SQL

1. FROM, JOIN
2. WHERE
3. GROUP BY
4. aggregate functions
5. HAVING
6. **window functions**
7. SELECT
8. DISTINCT
9. UNION/INTERSECT/EXCEPT
10. ORDER BY
11. OFFSET
12. LIMIT/FETCH/TOP

You can use window functions in SELECT and ORDER BY. However, you can't put window functions anywhere in the FROM, WHERE, GROUP BY, or HAVING clauses.

PARTITION BY

divides rows into multiple groups, called **partitions**, to which the window function is applied.

month	city	sold
1	Rome	200
2	Paris	500
1	London	100
1	Paris	300
2	Rome	300
2	London	400
3	Rome	400

PARTITION BY city

month	city	sold	sum
1	Paris	300	800
2	Paris	500	800
1	Rome	200	900
2	Rome	300	900
3	Rome	400	900
1	London	100	500
2	London	400	500

Default Partition: with no PARTITION BY clause, the entire result set is the partition.

ORDER BY

specifies the order of rows in each partition to which the window function is applied.

sold	city	month
200	Rome	1
500	Paris	2
100	London	1
300	Paris	1
300	Rome	2
400	London	2
400	Rome	3

PARTITION BY city ORDER BY month

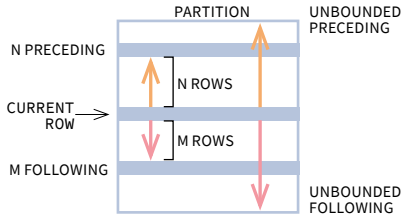
sold	city	month
300	Paris	1
500	Paris	2
200	Rome	1
300	Rome	2
400	Rome	3
100	London	1
400	London	2

Default ORDER BY: with no ORDER BY clause, the order of rows within each partition is arbitrary.

WINDOW FRAME

is a set of rows that are somehow related to the current row. The window frame is evaluated separately within each partition.

ROWS | RANGE | GROUPS BETWEEN lower_bound AND upper_bound



The bounds can be any of the five options:

- UNBOUNDED PRECEDING
- n PRECEDING
- CURRENT ROW
- n FOLLOWING
- UNBOUNDED FOLLOWING

The lower_bound must be BEFORE the upper_bound

ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING

	city	sold	month
	Paris	300	1
	Rome	200	1
	Paris	500	2
	Rome	100	4
current row →	Paris	200	4
	Paris	300	5
	Rome	200	5
	London	200	5
	London	100	6
	Rome	300	6

1 row before the current row and 1 row after the current row

RANGE BETWEEN 1 PRECEDING AND 1 FOLLOWING

	city	sold	month
	Paris	300	1
	Rome	200	1
	Paris	500	2
	Rome	100	4
current row →	Paris	200	4
	Paris	300	5
	Rome	200	5
	London	200	5
	London	100	6
	Rome	300	6

values in the range between 3 and 5
ORDER BY must contain a single expression

GROUPS BETWEEN 1 PRECEDING AND 1 FOLLOWING

	city	sold	month
	Paris	300	1
	Rome	200	1
	Paris	500	2
	Rome	100	4
current row →	Paris	200	4
	Paris	300	5
	Rome	200	5
	London	200	5
	London	100	6
	Rome	300	6

1 group before the current row and 1 group after the current row regardless of the value

As of 2020, GROUPS is only supported in PostgreSQL 11 and up.

ABBREVIATIONS

Abbreviation	Meaning
UNBOUNDED PRECEDING	BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
n PRECEDING	BETWEEN n PRECEDING AND CURRENT ROW
CURRENT ROW	BETWEEN CURRENT ROW AND CURRENT ROW
n FOLLOWING	BETWEEN AND CURRENT ROW AND n FOLLOWING
UNBOUNDED FOLLOWING	BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING

DEFAULT WINDOW FRAME

If ORDER BY is specified, then the frame is RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW.

Without ORDER BY, the frame specification is ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING.

SQL Window Functions Cheat Sheet

LearnSQL.com

LIST OF WINDOW FUNCTIONS

Aggregate Functions
- `avg()`
- `count()`
- `max()`
- `min()`
- `sum()`

Ranking Functions
- `row_number()`
- `rank()`
- `dense_rank()`

Distribution Functions
- `percent_rank()`
- `cume_dist()`

Analytic Functions
- `lead()`
- `lag()`
- `ntile()`
- `first_value()`
- `last_value()`
- `nth_value()`

AGGREGATE FUNCTIONS

- **`avg`***(expr)* – average value for rows within the window frame
- **`count`***(expr)* – count of values for rows within the window frame
- **`max`***(expr)* – maximum value within the window frame
- **`min`***(expr)* – minimum value within the window frame
- **`sum`***(expr)* – sum of values within the window frame

ORDER BY and `Window Frame`: Aggregate functions do not require an ORDER BY. They accept window frame definition (`ROWS`, `RANGE`, `GROUPS`).

RANKING FUNCTIONS

- **`row_number()`** – unique number for each row within partition, with different numbers for tied values
- **`rank()`** – ranking within partition, with gaps and same ranking for tied values
- **`dense_rank()`** – ranking within partition, with no gaps and same ranking for tied values

city	price	row_number	rank	dense_rank
		over(order by price)		
Paris	7	1	1	1
Rome	7	2	1	1
London	8.5	3	3	2
Berlin	8.5	4	3	2
Moscow	9	5	5	3
Madrid	10	6	6	4
Oslo	10	7	6	4

ORDER BY and Window Frame: `rank()` and `dense_rank()` require ORDER BY, but `row_number()` does not require ORDER BY. Ranking functions do not accept window frame definition (`ROWS`, `RANGE`, `GROUPS`).

DISTRIBUTION FUNCTIONS

- **`percent_rank()`** – the percentile ranking number of a row—a value in [0, 1] interval: (rank - 1) / (total number of rows - 1)
- **`cume_dist()`** – the cumulative distribution of a value within a group of values, i.e., the number of rows with values less than or equal to the current row's value divided by the total number of rows; a value in (0, 1] interval

`percent_rank() OVER(ORDER BY sold)`

city	sold	percent_rank
Paris	100	0
Berlin	150	0.25
Rome	200	0.5
Moscow	200	0.5
London	300	1

← without this row 50% of values are less than this row's value

`cume_dist() OVER(ORDER BY sold)`

city	sold	cume_dist
Paris	100	0.2
Berlin	150	0.4
Rome	200	0.8
Moscow	200	0.8
London	300	1

← 80% of values are less than or equal to this one

ORDER BY and Window Frame: Distribution functions require ORDER BY. They do not accept window frame definition (`ROWS`, `RANGE`, `GROUPS`).

ANALYTIC FUNCTIONS

- **`lead`***(expr, offset, default)* – the value for the row *offset* rows after the current; *offset* and *default* are optional; default values: *offset* = 1, *default* = NULL
- **`lag`***(expr, offset, default)* – the value for the row *offset* rows before the current; *offset* and *default* are optional; default values: *offset* = 1, *default* = NULL

`lead(sold) OVER(ORDER BY month)`

month	sold	
1	500	300
2	300	400
3	400	100
4	100	500
5	500	NULL

order by month

`lag(sold) OVER(ORDER BY month)`

month	sold	
1	500	NULL
2	300	500
3	400	300
4	100	400
5	500	100

order by month

`lead(sold, 2, 0) OVER(ORDER BY month)`

month	sold	
1	500	400
2	300	100
3	400	500
4	100	0
5	500	0

order by month; offset=2

`lag(sold, 2, 0) OVER(ORDER BY month)`

month	sold	
1	500	0
2	300	0
3	400	500
4	100	300
5	500	400

order by month; offset=2

- **`first_value`***(expr)* – the value for the first row within the window frame
- **`last_value`***(expr)* – the value for the last row within the window frame

`first_value(sold) OVER (PARTITION BY city ORDER BY month)`

city	month	sold	first_value
Paris	1	500	500
Paris	2	300	500
Paris	3	400	500
Rome	2	200	200
Rome	3	300	200
Rome	4	500	200

`last_value(sold) OVER (PARTITION BY city ORDER BY month RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)`

city	month	sold	last_value
Paris	1	500	400
Paris	2	300	400
Paris	3	400	400
Rome	2	200	500
Rome	3	300	500
Rome	4	500	500

Note: You usually want to use RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING with `last_value()`. With the default window frame for ORDER BY, RANGE UNBOUNDED PRECEDING, `last_value()` returns the value for the current row.

- **`ntile`***(n)* – divide rows within a partition as equally as possible into *n* groups, and assign each row its group number.

`ntile(3)`

city	sold	
Rome	100	1
Paris	100	1
London	200	1
Moscow	200	2
Berlin	200	2
Madrid	300	2
Oslo	300	3
Dublin	300	3

ORDER BY and Window Frame: `ntile()`, `lead()`, and `lag()` require an ORDER BY. They do not accept window frame definition (`ROWS`, `RANGE`, `GROUPS`).

- **`nth_value`***(expr, n)* – the value for the *n*-th row within the window frame; *n* must be an integer

```
nth_value(sold, 2) OVER (PARTITION BY city
  ORDER BY month RANGE BETWEEN UNBOUNDED
    PRECEDING AND UNBOUNDED FOLLOWING)
```

city	month	sold	nth_value
Paris	1	500	300
Paris	2	300	300
Paris	3	400	300
Rome	2	200	300
Rome	3	300	300
Rome	4	500	300
Rome	5	300	300
London	1	100	NULL

ORDER BY and Window Frame: `first_value()`, `last_value()`, and `nth_value()` do not require an ORDER BY. They accept window frame definition (`ROWS`, `RANGE`, `GROUPS`).

SQL JOINs Cheat Sheet

LearnSQL.com

JOINING TABLES

JOIN combines data from two tables.

TOY

toy_id	toy_name	cat_id
1	ball	3
2	spring	NULL
3	mouse	1
4	mouse	4
5	ball	1

CAT

cat_id	cat_name
1	Kitty
2	Hugo
3	Sam
4	Misty

JOIN typically combines rows with equal values for the specified columns. **Usually**, one table contains a **primary key**, which is a column or columns that uniquely identify rows in the table (the `cat_id` column in the `cat` table).
The other table has a column or columns that **refer to the primary key columns** in the first table (the `cat_id` column in the `toy` table). Such columns are **foreign keys**. The JOIN condition is the equality between the primary key columns in one table and columns referring to them in the other table.

JOIN

JOIN returns all rows that match the ON condition. JOIN is also called INNER JOIN.

```
SELECT *
FROM toy
JOIN cat
  ON toy.cat_id = cat.cat_id;
```

toy_id	toy_name	cat_id	cat_id	cat_name
5	ball	1	1	Kitty
3	mouse	1	1	Kitty
1	ball	3	3	Sam
4	mouse	4	4	Misty

There is also another, older syntax, but it **isn't recommended**.
List joined tables in the FROM clause, and place the conditions in the WHERE clause.

```
SELECT *
FROM toy, cat
WHERE toy.cat_id = cat.cat_id;
```

JOIN CONDITIONS

The JOIN condition doesn't have to be an equality – it can be any condition you want. JOIN doesn't interpret the JOIN condition, it only checks if the rows satisfy the given condition.

To refer to a column in the JOIN query, you have to use the full column name: first the table name, then a dot (.) and the column name:

```
  ON cat.cat_id = toy.cat_id
```

You can omit the table name and use just the column name if the name of the column is unique within all columns in the joined tables.

NATURAL JOIN

If the tables have columns with **the same name**, you can use NATURAL JOIN instead of JOIN.

```
SELECT *
FROM toy
NATURAL JOIN cat;
```

cat_id	toy_id	toy_name	cat_name
1	5	ball	Kitty
1	3	mouse	Kitty
3	1	ball	Sam
4	4	mouse	Misty

The common column appears only once in the result table.
Note: NATURAL JOIN is rarely used in real life.

LEFT JOIN

LEFT JOIN returns all rows from the **left table** with matching rows from the right table. Rows without a match are filled with NULLs. LEFT JOIN is also called LEFT OUTER JOIN.

```
SELECT *
FROM toy
LEFT JOIN cat
  ON toy.cat_id = cat.cat_id;
```

toy_id	toy_name	cat_id	cat_id	cat_name
5	ball	1	1	Kitty
3	mouse	1	1	Kitty
1	ball	3	3	Sam
4	mouse	4	4	Misty
2	spring	NULL	NULL	NULL

whole left table

RIGHT JOIN

RIGHT JOIN returns all rows from the **right table** with matching rows from the left table. Rows without a match are filled with NULLs. RIGHT JOIN is also called RIGHT OUTER JOIN.

```
SELECT *
FROM toy
RIGHT JOIN cat
  ON toy.cat_id = cat.cat_id;
```

toy_id	toy_name	cat_id	cat_id	cat_name
5	ball	1	1	Kitty
3	mouse	1	1	Kitty
NULL	NULL	NULL	2	Hugo
1	ball	3	3	Sam
4	mouse	4	4	Misty

whole right table

FULL JOIN

FULL JOIN returns all rows from the **left table** and all rows from the **right table**. It fills the non-matching rows with NULLs. FULL JOIN is also called FULL OUTER JOIN.

```
SELECT *
FROM toy
FULL JOIN cat
  ON toy.cat_id = cat.cat_id;
```

toy_id	toy_name	cat_id	cat_id	cat_name
5	ball	1	1	Kitty
3	mouse	1	1	Kitty
NULL	NULL	NULL	2	Hugo
1	ball	3	3	Sam
4	mouse	4	4	Misty
2	spring	NULL	NULL	NULL

whole left table | whole right table

CROSS JOIN

CROSS JOIN returns **all possible combinations** of rows from the left and right tables.

```
SELECT *
FROM toy
CROSS JOIN cat;
```

Other syntax:

```
SELECT *
FROM toy, cat;
```

toy_id	toy_name	cat_id	cat_id	cat_name
1	ball	3	1	Kitty
2	spring	NULL	1	Kitty
3	mouse	1	1	Kitty
4	mouse	4	1	Kitty
5	ball	1	1	Kitty
1	ball	3	2	Hugo
2	spring	NULL	2	Hugo
3	mouse	1	2	Hugo
4	mouse	4	2	Hugo
5	ball	1	2	Hugo
1	ball	3	3	Sam
...	...	...	...	...

SQL JOINs Cheat Sheet

LearnSQL.com

COLUMN AND TABLE ALIASES

Aliases give a temporary name to a **table** or a **column** in a table.

CAT AS c

cat_id	cat_name	mom_id	owner_id
1	Kitty	5	1
2	Hugo	1	2
3	Sam	2	2
4	Misty	1	NULL

OWNER AS o

id	name
1	John Smith
2	Danielle Davis

A **column alias** renames a column in the result. A **table alias** renames a table within the query. If you define a table alias, you must use it instead of the table name everywhere in the query. The AS keyword is optional in defining aliases.

```
SELECT
  o.name AS owner_name,
  c.cat_name
FROM cat   AS c
JOIN owner AS o
  ON c.owner_id = o.id;
```

cat_name	owner_name
Kitty	John Smith
Sam	Danielle Davis
Hugo	Danielle Davis

SELF JOIN

You can join a table to itself, for example, to show a parent-child relationship.

CAT AS child

cat_id	cat_name	owner_id	mom_id
1	Kitty	1	5
2	Hugo	2	1
3	Sam	2	2
4	Misty	NULL	1

CAT AS mom

cat_id	cat_name	owner_id	mom_id
1	Kitty	1	5
2	Hugo	2	1
3	Sam	2	2
4	Misty	NULL	1

Each occurrence of the table must be given a **different alias**. Each column reference must be preceded with an **appropriate table alias**.

```
SELECT
  child.cat_name AS child_name,
  mom.cat_name   AS mom_name
FROM cat AS child
JOIN cat AS mom
  ON child.mom_id = mom.cat_id;
```

child_name	mom_name
Hugo	Kitty
Sam	Hugo
Misty	Kitty

NON-EQUI SELF JOIN

You can use a **non-equality** in the ON condition, for example, to show **all different pairs** of rows.

TOY AS a

toy_id	toy_name	cat_id
3	mouse	1
5	ball	1
1	ball	3
4	mouse	4
2	spring	NULL

TOY AS b

cat_id	toy_id	toy_name
1	3	mouse
1	5	ball
3	1	ball
4	4	mouse
NULL	2	spring

```
SELECT
  a.toy_name AS toy_a,
  b.toy_name AS toy_b
FROM toy a
JOIN toy b
  ON a.cat_id < b.cat_id;
```

cat_a_id	toy_a	cat_b_id	toy_b
1	mouse	3	ball
1	ball	3	ball
1	mouse	4	mouse
1	ball	4	mouse
3	ball	4	mouse

MULTIPLE JOINS

You can join more than two tables together. First, two tables are joined, then the third table is joined to the result of the previous joining.

TOY AS t

toy_id	toy_name	cat_id
1	ball	3
2	spring	NULL
3	mouse	1
4	mouse	4
5	ball	1

CAT AS c

cat_id	cat_name	mom_id	owner_id
1	Kitty	5	1
2	Hugo	1	2
3	Sam	2	2
4	Misty	1	NULL

OWNER AS o

id	name
1	John Smith
2	Danielle Davis

JOIN & JOIN

```
SELECT
  t.toy_name,
  c.cat_name,
  o.name AS owner_name
FROM toy t
JOIN cat c
  ON t.cat_id = c.cat_id
JOIN owner o
  ON c.owner_id = o.id;
```

toy_name	cat_name	owner_name
ball	Kitty	John Smith
mouse	Kitty	John Smith
ball	Sam	Danielle Davis

JOIN & LEFT JOIN

```
SELECT
  t.toy_name,
  c.cat_name,
  o.name AS owner_name
FROM toy t
JOIN cat c
  ON t.cat_id = c.cat_id
LEFT JOIN owner o
  ON c.owner_id = o.id;
```

toy_name	cat_name	owner_name
ball	Kitty	John Smith
mouse	Kitty	John Smith
ball	Sam	Danielle Davis
mouse	Misty	NULL

LEFT JOIN & LEFT JOIN

```
SELECT
  t.toy_name,
  c.cat_name,
  o.name AS owner_name
FROM toy t
LEFT JOIN cat c
  ON t.cat_id = c.cat_id
LEFT JOIN owner o
  ON c.owner_id = o.id;
```

toy_name	cat_name	owner_name
ball	Kitty	John Smith
mouse	Kitty	John Smith
ball	Sam	Danielle Davis
mouse	Misty	NULL
spring	NULL	NULL

JOIN WITH MULTIPLE CONDITIONS

You can use multiple JOIN conditions using the **ON** keyword once and the **AND** keywords as many times as you need.

CAT AS c

cat_id	cat_name	mom_id	owner_id	age
1	Kitty	5	1	17
2	Hugo	1	2	10
3	Sam	2	2	5
4	Misty	1	NULL	11

OWNER AS o

id	name	age
1	John Smith	18
2	Danielle Davis	10

```
SELECT
  cat_name,
  o.name AS owner_name,
  c.age  AS cat_age,
  o.age  AS owner_age
FROM cat c
JOIN owner o
  ON c.owner_id = o.id
 AND c.age < o.age;
```

cat_name	owner_name	age	age
Kitty	John Smith	17	18
Sam	Danielle Davis	5	10

Standard SQL Functions Cheat Sheet

LearnSQL.com

TEXT FUNCTIONS

CONCATENATION

Use the `||` operator to concatenate two strings:

```
SELECT 'Hi ' || 'there!';
-- result: Hi there!
```

Remember that you can concatenate only character strings using `||`. Use this trick for numbers:

```
SELECT '' || 4 || 2;
-- result: 42
```

Some databases implement non-standard solutions for concatenating strings like `CONCAT()` or `CONCAT_WS()`. Check the documentation for your specific database.

LIKE OPERATOR – PATTERN MATCHING

Use the `_` character to replace any single character. Use the `%` character to replace any number of characters (including 0 characters).

Fetch all names that start with any letter followed by `'atherine'`:

```
SELECT name
FROM names
WHERE name LIKE '_atherine';
```

Fetch all names that end with `'a'`:

```
SELECT name
FROM names
WHERE name LIKE '%a';
```

USEFUL FUNCTIONS

Get the count of characters in a string:

```
SELECT LENGTH('LearnSQL.com');
-- result: 12
```

Convert all letters to lowercase:

```
SELECT LOWER('LEARNSQL.COM');
-- result: learnsql.com
```

Convert all letters to uppercase:

```
SELECT UPPER('LearnSQL.com');
-- result: LEARNSQL.COM
```

Convert all letters to lowercase and all first letters to uppercase (not implemented in MySQL and SQL Server):

```
SELECT INITCAP('edgar frank ted cODD');
-- result: Edgar Frank Ted Codd
```

Get just a part of a string:

```
SELECT SUBSTRING('LearnSQL.com', 9);
-- result: .com
SELECT SUBSTRING('LearnSQL.com', 0, 6);
-- result: Learn
```

Replace part of a string:

```
SELECT REPLACE('LearnSQL.com', 'SQL',
'Python');
-- result: LearnPython.com
```

NUMERIC FUNCTIONS

BASIC OPERATIONS

Use `+`, `-`, `*`, `/` to do some basic math. To get the number of seconds in a week:

```
SELECT 60 * 60 * 24 * 7; -- result: 604800
```

CASTING

From time to time, you need to change the type of a number. The `CAST()` function is there to help you out. It lets you change the type of value to almost anything (`integer`, `numeric`, `double precision`, `varchar`, and many more).
Get the number as an integer (without rounding):

```
SELECT CAST(1234.567 AS integer);
-- result: 1234
```

Change a column type to `double precision`

```
SELECT CAST(column AS double precision);
```

USEFUL FUNCTIONS

Get the remainder of a division:

```
SELECT MOD(13, 2);
-- result: 1
```

Round a number to its nearest integer:

```
SELECT ROUND(1234.56789);
-- result: 1235
```

Round a number to three decimal places:

```
SELECT ROUND(1234.56789, 3);
-- result: 1234.568
```

PostgreSQL requires the first argument to be of the type `numeric` – cast the number when needed.

To round the number **up**:

```
SELECT CEIL(13.1); -- result: 14
SELECT CEIL(-13.9); -- result: -13
```

The `CEIL(x)` function returns the **smallest** integer **not less** than x. In SQL Server, the function is called `CEILING()`.

To round the number **down**:

```
SELECT FLOOR(13.8); -- result: 13
SELECT FLOOR(-13.2); -- result: -14
```

The `FLOOR(x)` function returns the **greatest** integer **not greater** than x.

To round towards 0 irrespective of the sign of a number:

```
SELECT TRUNC(13.5); -- result: 13
SELECT TRUNC(-13.5); -- result: -13
```

`TRUNC(x)` works the same way as `CAST(x AS integer)`. In MySQL, the function is called `TRUNCATE()`.

To get the absolute value of a number:

```
SELECT ABS(-12); -- result: 12
```

To get the square root of a number:

```
SELECT SQRT(9); -- result: 3
```

NULLs

To retrieve all rows with a missing value in the `price` column:

```
WHERE price IS NULL
```

To retrieve all rows with the `weight` column populated:

```
WHERE weight IS NOT NULL
```

Why shouldn't you use `price = NULL` or `weight != NULL`? Because databases don't know if those expressions are true or false – they are evaluated as NULLs.
Moreover, if you use a function or concatenation on a column that is NULL in some rows, then it will get propagated. Take a look:

domain	LENGTH(domain)
LearnSQL.com	12
LearnPython.com	15
NULL	*NULL*
vertabelo.com	13

USEFUL FUNCTIONS

COALESCE(x, y, ...)

To replace NULL in a query with something meaningful:

```
SELECT
  domain,
  COALESCE(domain, 'domain missing')
FROM contacts;
```

domain	coalesce
LearnSQL.com	LearnSQL.com
NULL	domain missing

The `COALESCE()` function takes any number of arguments and returns the value of the first argument that isn't NULL.

NULLIF(x, y)

To save yourself from *division by 0* errors:

```
SELECT
  last_month,
  this_month,
  this_month * 100.0
    / NULLIF(last_month, 0)
    AS better_by_percent
FROM video_views;
```

last_month	this_month	better_by_percent
723786	1085679	150.0
0	178123	*NULL*

The `NULLIF(x, y)` function will return NULL if x is the same as y, else it will return the x value.

CASE WHEN

The basic version of `CASE WHEN` checks if the values are equal (e.g., if `fee` is equal to 50, then `'normal'` is returned). If there isn't a matching value in the `CASE WHEN`, then the `ELSE` value will be returned (e.g., if `fee` is equal to 49, then `'not available'` will show up.

```
SELECT
  CASE fee
    WHEN 50 THEN 'normal'
    WHEN 10 THEN 'reduced'
    WHEN  0 THEN 'free'
    ELSE 'not available'
  END AS tariff
FROM ticket_types;
```

The most popular type is the **searched CASE WHEN** – it lets you pass conditions (as you'd write them in the `WHERE` clause), evaluates them in order, then returns the value for the first condition met.

```
SELECT
  CASE
    WHEN score >= 90 THEN 'A'
    WHEN score >  60 THEN 'B'
    ELSE 'F'
  END AS grade
FROM test_results;
```

Here, all students who scored at least 90 will get an A, those with the score above 60 (and below 90) will get a B, and the rest will receive an F.

TROUBLESHOOTING

Integer division

When you don't see the decimal places you expect, it means that you are dividing between two integers. Cast one to `decimal`:

```
CAST(123 AS decimal) / 2
```

Division by 0

To avoid this error, make sure that the denominator is not equal to 0. You can use the `NULLIF()` function to replace 0 with a NULL, which will result in a NULL for the whole expression:

```
count / NULLIF(count_all, 0)
```

Inexact calculations

If you do calculations using `real` (floating point) numbers, you'll end up with some inaccuracies. This is because this type is meant for scientific calculations such as calculating the velocity.
Whenever you need accuracy (such as dealing with monetary values), use the `decimal` / `numeric` type (or `money` if available).

Errors when rounding with a specified precision

Most databases won't complain, but do check the documentation if they do. For example, if you want to specify the rounding precision in PostgreSQL, the value must be of the `numeric` type.

Standard SQL Functions Cheat Sheet

LearnSQL.com

AGGREGATION AND GROUPING

- **COUNT(**expr**)** – the count of values for the rows within the group
- **SUM(**expr**)** – the sum of values within the group
- **AVG(**expr**)** – the average value for the rows within the group
- **MIN(**expr**)** – the minimum value within the group
- **MAX(**expr**)** – the maximum value within the group

To get the number of rows in the table:

```
SELECT COUNT(*)
FROM city;
```

To get the number of non-NULL values in a column:

```
SELECT COUNT(rating)
FROM city;
```

To get the count of unique values in a column:

```
SELECT COUNT(DISTINCT country_id)
FROM city;
```

GROUP BY

CITY	
name	**country_id**
Paris	1
Marseille	1
Lyon	1
Berlin	2
Hamburg	2
Munich	2
Warsaw	4
Cracow	4

→

CITY	
country_id	**count**
1	3
2	3
4	2

The example above – the count of cities in each country:

```
SELECT name, COUNT(country_id)
FROM city
GROUP BY name;
```

The average rating for the city:

```
SELECT city_id, AVG(rating)
FROM ratings
GROUP BY city_id;
```

Common mistake: COUNT(*) and LEFT JOIN

When you join the tables like this: `client LEFT JOIN project`, and you want to get the number of projects for every client you know, `COUNT(*)` will return **1** for each client even if you've never worked for them. This is because, they're still present in the list but with the `NULL` in the fields related to the project after the `JOIN`. To get the correct count (**0** for the clients you've never worked for), count the values in a column of the other table, e.g., `COUNT(project_name)`. Check out this exercise to see an example.

DATE AND TIME

There are 3 main time-related types: `date`, `time`, and `timestamp`. Time is expressed using a 24-hour clock, and it can be as vague as just hour and minutes (e.g., `15:30` – 3:30 p.m.) or as precise as microseconds and time zone (as shown below):

```
2021-12-31 14:39:53.662522-05
└ date ┘└      time      ┘
└          timestamp          ┘
YYYY-mm-dd HH:MM:SS.ssssss±TZ
```

`14:39:53.662522-05` is almost 2:40 p.m. CDT (e.g., in Chicago; in UTC it'd be 7:40 p.m.). The letters in the above example represent:

In the date part:

- YYYY – the 4-digit year.
- mm – the zero-padded month (01—January through 12—December).
- dd – the zero-padded day.

In the time part:

- HH – the zero-padded hour in a 24-hour clock.
- MM – the minutes.
- SS – the seconds. *Omissible.*
- ssssss – the smaller parts of a second – they can be expressed using 1 to 6 digits. *Omissible.*
- ±TZ – the timezone. It must start with either + or −, and use two digits relative to UTC. *Omissible.*

What time is it?

To answer that question in SQL, you can use:

- CURRENT_TIME – to find what time it is.
- CURRENT_DATE – to get today's date. (`GETDATE()` in SQL Server.)
- CURRENT_TIMESTAMP – to get the timestamp with the two above.

Creating values

To create a `date`, `time`, or `timestamp`, simply write the value as a string and cast it to the proper type.

```
SELECT CAST('2021-12-31' AS date);
SELECT CAST('15:31' AS time);
SELECT CAST('2021-12-31 23:59:29+02' AS
timestamp);
SELECT CAST('15:31.124769' AS time);
```

Be careful with the last example – it will be interpreted as 15 minutes 31 seconds and 124769 microseconds! It is always a good idea to write `00` explicitly for hours: `'00:15:31.124769'`.

You might skip casting in simple conditions – the database will know what you mean.

```
SELECT airline, flight_number, departure_time
FROM airport_schedule
WHERE departure_time < '12:00';
```

INTERVALs

Note: In SQL Server, intervals aren't implemented – use the `DATEADD()` and `DATEDIFF()` functions.

To get the simplest interval, subtract one time value from another:

```
SELECT CAST('2021-12-31 23:59:59' AS
timestamp) - CAST('2021-06-01 12:00:00' AS
timestamp);
-- result: 213 days 11:59:59
```

To define an interval: `INTERVAL '1' DAY`
This syntax consists of three elements: the `INTERVAL` keyword, a quoted value, and a time part keyword (in singular form.) You can use the following time parts: `YEAR`, `MONTH`, `WEEK`, `DAY`, `HOUR`, `MINUTE`, and `SECOND`. In MySQL, omit the quotes. You can join many different `INTERVAL`s using the + or − operator:

```
INTERVAL '1' YEAR + INTERVAL '3' MONTH
```

In some databases, there's an easier way to get the above value. And it accepts plural forms! `INTERVAL '1 year 3 months'`
There are two more syntaxes in the Standard SQL:

Syntax	What it does
`INTERVAL 'x-y' YEAR TO MONTH`	`INTERVAL 'x year y month'`
`INTERVAL 'x-y' DAY TO SECOND`	`INTERVAL 'x day y second'`

In MySQL, write `year_month` instead of `YEAR TO MONTH` and `day_second` instead of `DAY TO SECOND`.

To get the last day of a month, add one month and subtract one day:

```
SELECT CAST('2021-02-01' AS date)
       + INTERVAL '1' MONTH
       - INTERVAL '1' DAY;
```

To get all events for next three months from today:

```
SELECT event_date, event_name
FROM calendar
WHERE event_date BETWEEN CURRENT_DATE AND
CURRENT_DATE + INTERVAL '3' MONTH;
```

To get part of the date:

```
SELECT EXTRACT(YEAR FROM birthday)
FROM artists;
```

One of possible returned values: 1946. In SQL Server, use the `DATEPART(part, date)` function.

TIME ZONES

In the SQL Standard, the `date` type can't have an associated time zone, but the `time` and `timestamp` types can. In the real world, time zones have little meaning without the date, as the offset can vary through the year because of **daylight saving time**. So, it's best to work with the `timestamp` values.

When working with the type `timestamp with time zone` (abbr. `timestamptz`), you can type in the value in your local time zone, and it'll get converted to the UTC time zone as it is inserted into the table. Later when you select from the table it gets converted back to your local time zone. This is immune to time zone changes.

AT TIME ZONE

To operate between different time zones, use the `AT TIME ZONE` keyword.

If you use this format: `{timestamp without time zone} AT TIME ZONE {time zone}`, then the database will read the time stamp in the specified time zone and convert it to the time zone local to the display. It returns the time in the format `timestamp with time zone`.

If you use this format: `{timestamp with time zone} AT TIME ZONE {time zone}`, then the database will convert the time in one time zone to the target time zone specified by `AT TIME ZONE`. It returns the time in the format `timestamp without time zone`, in the target time zone.

You can define the time zone with popular shortcuts like `UTC`, `MST`, or `GMT`, or by `continent/city` such as: `America/New_York`, `Europe/London`, and `Asia/Tokyo`.

Examples

We set the local time zone to `'America/New_York'`.

```
SELECT TIMESTAMP '2021-07-16 21:00:00' AT
TIME ZONE 'America/Los_Angeles';
-- result: 2021-07-17 00:00:00-04
```

Here, the database takes a timestamp without a time zone and it's told it's in Los Angeles time, which is then converted to the local time – New York for displaying. This answers the question **"At what time should I turn on the TV if the show starts at 9 PM in Los Angeles?"**

```
SELECT TIMESTAMP WITH TIME ZONE '2021-06-20
19:30:00' AT TIME ZONE 'Australia/Sydney';
-- result: 2021-06-21 09:30:00
```

Here, the database gets a timestamp specified in the local time zone and converts it to the time in Sydney (note that it didn't return a time zone.) This answers the question **"What time is it in Sydney if it's 7:30 PM here?"**

SQL for Data Analysis Cheat Sheet

LearnSQL.com

SQL

SQL, or *Structured Query Language*, is a language for talking to databases. It lets you select specific data and build complex reports. Today, SQL is a universal language of data, used in practically all technologies that process data.

SELECT

Fetch the `id` and `name` columns from the `product` table:

```
SELECT id, name
FROM product;
```

Concatenate the name and the description to fetch the full description of the products:

```
SELECT name || ' - ' || description
FROM product;
```

Fetch names of products with prices above 15:

```
SELECT name
FROM product
WHERE price > 15;
```

Fetch names of products with prices between 50 and 150:

```
SELECT name
FROM product
WHERE price BETWEEN 50 AND 150;
```

Fetch names of products that are not watches:

```
SELECT name
FROM product
WHERE name != 'watch';
```

Fetch names of products that start with a `'P'` or end with an `'s'`:

```
SELECT name
FROM product
WHERE name LIKE 'P%' OR name LIKE '%s';
```

Fetch names of products that start with any letter followed by `'rain'` (like `'train'` or `'grain'`):

```
SELECT name
FROM product
WHERE name LIKE '_rain';
```

Fetch names of products with non-null prices:

```
SELECT name
FROM product
WHERE price IS NOT NULL;
```

GROUP BY

PRODUCT	
name	**category**
Knife	Kitchen
Pot	Kitchen
Mixer	Kitchen
Jeans	Clothing
Sneakers	Clothing
Leggings	Clothing
Smart TV	Electronics
Laptop	Electronics

→

category	count
Kitchen	3
Clothing	3
Electronics	2

AGGREGATE FUNCTIONS

Count the number of products:

```
SELECT COUNT(*)
FROM product;
```

Count the number of products with non-null prices:

```
SELECT COUNT(price)
FROM product;
```

Count the number of unique `category` values:

```
SELECT COUNT(DISTINCT category)
FROM product;
```

Get the lowest and the highest product price:

```
SELECT MIN(price), MAX(price)
FROM product;
```

Find the total price of products for each category:

```
SELECT category, SUM(price)
FROM product
GROUP BY category;
```

Find the average price of products for each category whose average is above 3.0:

```
SELECT category, AVG(price)
FROM product
GROUP BY category
HAVING AVG(price) > 3.0;
```

ORDER BY

Fetch product names sorted by the `price` column in the default ASCending order:

```
SELECT name
FROM product
ORDER BY price [ASC];
```

Fetch product names sorted by the `price` column in DESCending order:

```
SELECT name
FROM product
ORDER BY price DESC;
```

COMPUTATIONS

Use `+`, `-`, `*`, `/` to do basic math. To get the number of seconds in a week:

```
SELECT 60 * 60 * 24 * 7;
-- result: 604800
```

ROUNDING NUMBERS

Round a number to its nearest integer:

```
SELECT ROUND(1234.56789);
-- result: 1235
```

Round a number to two decimal places:

```
SELECT ROUND(AVG(price), 2)
FROM product
WHERE category_id = 21;
-- result: 124.56
```

TROUBLESHOOTING

INTEGER DIVISION

In PostgreSQL and SQL Server, the `/` operator performs integer division for integer arguments. If you do not see the number of decimal places you expect, it is because you are dividing between two integers. Cast one to decimal:

```
123 / 2 -- result: 61
CAST(123 AS decimal) / 2 -- result: 61.5
```

DIVISION BY 0

To avoid this error, make sure the denominator is not `0`. You may use the `NULLIF()` function to replace `0` with a `NULL`, which results in a `NULL` for the entire expression:

```
count / NULLIF(count_all, 0)
```

JOIN

JOIN is used to fetch data from multiple tables. To get the names of products purchased in each order, use:

```
SELECT
 orders.order_date,
 product.name AS product,
 amount
FROM orders
JOIN product
  ON product.id = orders.product_id;
```

Learn more about JOINs in our interactive SQL JOINs course.

INSERT

To insert data into a table, use the `INSERT` command:

```
INSERT INTO category
VALUES
(1, 'Home and Kitchen'),
(2, 'Clothing and Apparel');
```

You may specify the columns to which the data is added. The remaining columns are filled with predefined default values or `NULL`s.

```
INSERT INTO category (name)
VALUES ('Electronics');
```

UPDATE

To update the data in a table, use the `UPDATE` command:

```
UPDATE category
SET
 is_active = true,
 name = 'Office'
WHERE name = 'Ofice';
```

DELETE

To delete data from a table, use the `DELETE` command:

```
DELETE FROM category
WHERE name IS NULL;
```

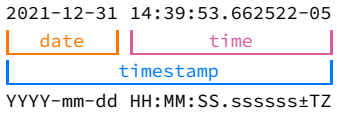
Check out our interactive course How to INSERT, UPDATE, and DELETE Data in SQL.

SQL for Data Analysis Cheat Sheet

LearnSQL.com

DATE AND TIME

There are 3 main time-related types: `date`, `time`, and `timestamp`. Time is expressed using a 24-hour clock, and it can be as vague as just hour and minutes (e.g., `15:30` – 3:30 p.m.) or as precise as microseconds and time zone (as shown below):



`14:39:53.662522-05` is almost 2:40 p.m. CDT (e.g., in Chicago; in UTC it'd be 7:40 p.m.). The letters in the above example represent:

In the date part:

- YYYY – the 4-digit year.
- mm – the zero-padded month (01—January through 12—December).
- dd – the zero-padded day.

In the time part:

- HH – the zero-padded hour in a 24-hour clock.
- MM – the minutes.
- SS – the seconds. *Omissible*.
- ssssss – the smaller parts of a second – they can be expressed using 1 to 6 digits. *Omissible*.
- ±TZ – the timezone. It must start with either + or −, and use two digits relative to UTC. *Omissible*.

CURRENT DATE AND TIME

Find out what time it is:

```
SELECT CURRENT_TIME;
```

Get today's date:

```
SELECT CURRENT_DATE;
```

In SQL Server:

```
SELECT GETDATE();
```

Get the timestamp with the current date and time:

```
SELECT CURRENT_TIMESTAMP;
```

CREATING DATE AND TIME VALUES

To create a date, time, or timestamp, write the value as a string and cast it to the proper type.

```
SELECT CAST('2021-12-31' AS date);
SELECT CAST('15:31' AS time);
SELECT CAST('2021-12-31 23:59:29+02'
       AS timestamp);
SELECT CAST('15:31.124769' AS time);
```

Be careful with the last example – it is interpreted as 15 minutes 31 seconds and 124769 microseconds! It is always a good idea to write 00 for hours explicitly: `'00:15:31.124769'`.

SORTING CHRONOLOGICALLY

Using `ORDER BY` on date and time columns sorts rows chronologically from the oldest to the most recent:

```
SELECT order_date, product, quantity
FROM sales
ORDER BY order_date;
```

order_date	product	quantity
2023-07-22	Laptop	2
2023-07-23	Mouse	3
2023-07-24	Sneakers	10
2023-07-24	Jeans	3
2023-07-25	Mixer	2

Use the DESCending order to sort from the most recent to the oldest:

```
SELECT order_date, product, quantity
FROM sales
ORDER BY order_date DESC;
```

COMPARING DATE AND TIME VALUES

You may use the comparison operators <, <=, >, >=, and = to compare date and time values. Earlier dates are less than later ones. For example, `2023-07-05` is "less" than `2023-08-05`.

Find sales made in July 2023:

```
SELECT order_date, product_name, quantity
FROM sales
WHERE order_date >= '2023-07-01'
  AND order_date <  '2023-08-01';
```

Find customers who registered in July 2023:

```
SELECT registration_timestamp, email
FROM customer
WHERE registration_timestamp >= '2023-07-01'
  AND registration_timestamp <  '2023-08-01';
```

Note: Pay attention to the end date in the query. The upper bound `'2023-08-01'` is not included in the range. The timestamp `'2023-08-01'` is actually the timestamp `'2023-08-01 00:00:00.0'`. The comparison operator < is used to ensure the selection is made for all timestamps less than `'2023-08-01 00:00:00.0'`, that is, all timestamps in July 2023, even those close to the midnight of August 1, 2023.

INTERVALS

An interval measures the difference between two points in time. For example, the interval between `2023-07-04` and `2023-07-06` is 2 days.

To define an interval in SQL, use this syntax:

```
INTERVAL '1' DAY
```

The syntax consists of three elements: the `INTERVAL` keyword, a quoted value, and a time part keyword. You may use the following time parts: `YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE`, and `SECOND`.

Adding intervals to date and time values

You may use + or − to add or subtract an interval to date or timestamp values.

Subtract one year from `2023-07-05`:

```
SELECT CAST('2023-07-05' AS TIMESTAMP)
       - INTERVAL '1' year;
-- result: 2022-07-05 00:00:00
```

Find customers who placed the first order within a month from the registration date:

```
SELECT id
FROM customers
WHERE first_order_date >
  registration_date + INTERVAL '1' month;
```

Filtering events to those in the last 7 days

To find the deliveries scheduled for the last 7 days, use:

```
SELECT delivery_date, address
FROM sales
WHERE delivery_date <= CURRENT_DATE
  AND delivery_date >= CURRENT_DATE
      - INTERVAL '7' DAY;
```

Note: In SQL Server, intervals are not implemented – use the `DATEADD()` and `DATEDIFF()` functions.

Filtering events to those in the last 7 days in SQL Server

To find the sales made within the last 7 days, use:

```
SELECT delivery_date, address
FROM sales
WHERE delivery_date <= GETDATE()
  AND delivery_date >=
    DATEADD(DAY, -7, GETDATE());
```

EXTRACTING PARTS OF DATES

The standard SQL syntax to get a part of a date is

```
SELECT EXTRACT(YEAR FROM order_date)
FROM sales;
```

You may extract the following fields:
`YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE`, and `SECOND`.

The standard syntax does not work In SQL Server.
Use the `DATEPART(part, date)` function instead.

```
SELECT DATEPART(YEAR, order_date)
FROM sales;
```

GROUPING BY YEAR AND MONTH

Find the count of sales by month:

```
SELECT
 EXTRACT(YEAR FROM order_date) AS year,
 EXTRACT(MONTH FROM order_date) AS month,
 COUNT(*) AS count
FROM sales
GROUP BY
 year,
 month
ORDER BY
 year
 month;
```

year	month	count
2022	8	51
2022	9	58
2022	10	62
2022	11	76
2022	12	85
2023	1	71
2023	2	69

Note that you must group by both the year and the month. `EXTRACT(MONTH FROM order_date)` only extracts the month number (1, 2, ..., 12). To distinguish between months from different years, you must also group by year.

More about working with date and time values in our interactive Standard SQL Functions course.

SQL for Data Analysis Cheat Sheet

LearnSQL.com

CASE WHEN

`CASE WHEN` lets you pass conditions (as in the `WHERE` clause), evaluates them in order, then returns the value for the first condition met.

```
SELECT
 name,
 CASE
   WHEN price > 150 THEN 'Premium'
   WHEN price > 100 THEN 'Mid-range'
   ELSE 'Standard'
 END AS price_category
FROM product;
```

Here, all products with prices above 150 get the *Premium* label, those with prices above 100 (and below 150) get the *Mid-range* label, and the rest receives the *Standard* label.

CASE WHEN and GROUP BY

You may combine `CASE WHEN` and `GROUP BY` to compute object statistics in the categories you define.

```
SELECT
 CASE
   WHEN price > 150 THEN 'Premium'
   WHEN price > 100 THEN 'Mid-range'
   ELSE 'Standard'
 END AS price_category,
 COUNT(*) AS products
FROM product
GROUP BY price_category;
```

Count the number of large orders for each customer using `CASE WHEN` and `SUM()`:

```
SELECT
  customer_id,
  SUM(
    CASE WHEN quantity > 10
    THEN 1 ELSE 0 END
  ) AS large_orders
FROM sales
GROUP BY customer_id;
```

... or using `CASE WHEN` and `COUNT()`:

```
SELECT
  customer_id,
  COUNT(
    CASE WHEN quantity > 10
    THEN order_id END
  ) AS large_orders
FROM sales
GROUP BY customer_id;
```

GROUP BY EXTENSIONS

GROUPING SETS

`GROUPING SETS` lets you specify multiple sets of columns to group by in one query.

```
SELECT region, product, COUNT(order_id)
FROM sales
GROUP BY
  GROUPING SETS ((region, product), ());
```

region	product	count	
USA	Laptop	10	GROUP BY (region, product)
USA	Mouse	5	
UK	Laptop	6	
NULL	NULL	21	GROUP BY () – all rows

CUBE

CUBE generates groupings for all possible subsets of the GROUP BY columns.

```
SELECT region, product, COUNT(order_id)
FROM sales
GROUP BY CUBE (region, product);
```

region	product	count	
USA	Laptop	10	GROUP BY region, product
USA	Mouse	5	
UK	Laptop	6	
USA	NULL	15	GROUP BY region
UK	NULL	6	
NULL	Laptop	16	GROUP BY product
NULL	Mouse	5	
NULL	NULL	21	GROUP BY () – all rows

ROLLUP

ROLLUP adds new levels of grouping for subtotals and grand totals.

```
SELECT region, product, COUNT(order_id)
FROM sales
GROUP BY ROLLUP (region, product);
```

region	product	count	
USA	Laptop	10	GROUP BY region, product
USA	Mouse	5	
UK	Laptop	6	
USA	NULL	15	GROUP BY region
UK	NULL	6	
NULL	NULL	21	GROUP BY () – all rows

COALESCE

COALESCE replaces the first NULL argument with a given value. It is often used to display labels with `GROUP BY` extensions.

```
SELECT region,
 COALESCE(product, 'All'),
 COUNT(order_id)
FROM sales
GROUP BY ROLLUP (region, product);
```

region	product	count
USA	Laptop	10
USA	Mouse	5
USA	All	15
UK	Laptop	6
UK	All	6
All	All	21

COMMON TABLE EXPRESSIONS

A common table expression (CTE) is a named temporary result set that can be referenced within a larger query. They are especially useful for complex aggregations and for breaking down large queries into more manageable parts.

```
WITH total_product_sales AS (
 SELECT product, SUM(profit) AS total_profit
 FROM sales
 GROUP BY product
)

SELECT AVG(total_profit)
FROM total_product_sales;
```

Check out our hands-on courses on Common Table Expressions and GROUP BY Extensions.

WINDOW FUNCTIONS

Window functions compute their results based on a sliding window frame, a set of rows related to the current row. Unlike aggregate functions, window functions do not collapse rows.

COMPUTING THE PERCENT OF TOTAL WITHIN A GROUP

```
SELECT product, brand, profit,
  (100.0 * profit /
    SUM(profit) OVER(PARTITION BY brand)
  ) AS perc
FROM sales;
```

product	brand	profit	perc
Knife	Culina	1000	25
Pot	Culina	3000	75
Doll	Toyze	2000	40
Car	Toyze	3000	60

RANKING

Rank products by price:

```
SELECT RANK() OVER(ORDER BY price), name
FROM product;
```

RANKING FUNCTIONS

RANK – gives the same rank for tied values, leaves gaps.
DENSE_RANK – gives the same rank for tied values without gaps.
ROW_NUMBER – gives consecutive numbers without gaps.

name	rank	dense_rank	row_number
Jeans	1	1	1
Leggings	2	2	2
Leggings	2	2	3
Sneakers	4	3	4
Sneakers	4	3	5
Sneakers	4	3	6
T-Shirt	7	4	7

RUNNING TOTAL

A running total is the cumulative sum of a given value and all preceding values in a column.

```
SELECT date, amount,
  SUM(amount) OVER(ORDER BY date)
    AS running_total
FROM sales;
```

MOVING AVERAGE

A moving average (*a.k.a.* rolling average, running average) is a technique for analyzing trends in time series data. It is the average of the current value and a specified number of preceding values.

```
SELECT date, price,
  AVG(price) OVER(
    ORDER BY date
    ROWS BETWEEN 2 PRECEDING
      AND CURRENT ROW
  ) AS moving_averge
FROM stock_prices;
```

DIFFERENCE BETWEEN TWO ROWS (DELTA)

```
SELECT year, revenue,
  LAG(revenue) OVER(ORDER BY year)
    AS revenue_prev_year,
  revenue -
    LAG(revenue) OVER(ORDER BY year)
    AS yoy_difference
FROM yearly_metrics;
```

Learn about SQL window functions in our interactive Window Functions course.

SQL Server Cheat Sheet

LearnSQL.com

SQL Server is a popular relational database management system developed by Microsoft. It is widely used for storing, managing, and processing data in various environments.

Transact-SQL (T-SQL) is an extension of the SQL language, designed specifically for SQL Server. It allows for advanced database operations such as defining stored procedures, triggers, and indexes.

SQL Server Management Studio (SSMS) is the official graphical tool for managing SQL Server databases. It offers a comprehensive interface for administrators and developers to design databases, write queries, and optimize database performance, among other tasks.

Download Microsoft SQL Server here:
https://www.microsoft.com/en-us/sql-server/sql-server-downloads

CREATING AND DISPLAYING DATABASES

To create a database:
```
CREATE DATABASE Zoo;
```

To list all databases on a server:
```
SELECT *
FROM sys.databases;
```

To use a specified database:
```
USE Zoo;
```

To delete a specified database:
```
DROP DATABASE Zoo;
```

To create a schema:
```
CREATE SCHEMA AnimalSchema;
```

DISPLAYING TABLES

To list all tables in a database:
```
SELECT *
FROM sys.tables;
```

To get information about a specified table:
```
exec sp_help 'Animal'
```

CREATING TABLES

To create a table:
```
CREATE TABLE Habitat (
  Id INT,
  Name VARCHAR(64)
);
```

Use IDENTITY to increment the ID automatically with each new record.
```
CREATE TABLE Habitat (
  Id INT PRIMARY KEY IDENTITY,
  Name VARCHAR(64)
);
```

To create a table with a foreign key:
```
CREATE TABLE Animal (
  Id INT PRIMARY KEY IDENTITY,
  Name VARCHAR(64),
  Species VARCHAR(64),
  Age INT,
  HabitatId INT,
  FOREIGN KEY (HabitatId)
    REFERENCES Habitat(Id)
);
```

MODIFYING TABLES

Use the `ALTER TABLE` or the `EXEC` statement to modify a table structure.

To change a table name:
```
EXEC sp_rename 'AnimalSchema.Animal', 'Pet'
```

To add a column to a table:
```
ALTER TABLE Animal
ADD COLUMN Name VARCHAR(64);
```

To change a column name:
```
EXEC sp_rename 'AnimalSchema.Animal.Id',
'Identifier', 'COLUMN';
```

To change a column data type:
```
ALTER TABLE Animal
ALTER COLUMN Name VARCHAR(128);
```

To delete a column:
```
ALTER TABLE Animal
DROP COLUMN Name;
```

To delete a table:
```
DROP TABLE Animal;
```

QUERYING DATA

To select data from a table, use the `SELECT` command.

An example of a single-table query:
```
SELECT Species, AVG(Age) AS AverageAge
FROM Animal
WHERE Id != 3
GROUP BY Species
HAVING AVG(Age) > 3
ORDER BY AVG(Age) DESC;
```

An example of a multiple-table query:
```
SELECT City.Name, Country.Name
FROM City
[INNER | LEFT | RIGHT | FULL] JOIN Country
  ON City.CountryId = Country.Id;
```

AGGREGATION AND GROUPING

- **`AVG(expr)`** – average value of `expr` for the group.
- **`COUNT(expr)`** – count of `expr` values within the group.
- **`MAX(expr)`** – maximum value of `expr` values within the group.
- **`MIN(expr)`** – minimum value of `expr` values within the group.
- **`SUM(expr)`** – sum of `expr` values within the group.

To count the rows in the table:
```
SELECT COUNT(*)
FROM Animal;
```

To count the non-NULL values in a column:
```
SELECT COUNT(Name)
FROM Animal;
```

To count unique values in a column:
```
SELECT COUNT(DISTINCT Name)
FROM Animal;
```

GROUP BY

To count the animals by species:
```
SELECT Species, COUNT(Id)
FROM Animal
GROUP BY Species;
```

To get the average, minimum, and maximum ages by habitat:
```
SELECT HabitatId, AVG(Age),
       MIN(Age), MAX(Age)
FROM Animal
GROUP BY HabitatId;
```

INSERTING DATA

To insert data into a table, use the `INSERT` command:
```
INSERT INTO Habitat VALUES
(1, 'River'),
(2, 'Forest');
```

You may specify the columns in which the data is added. The remaining columns are filled with default values or `NULL`s.
```
INSERT INTO Habitat (Name) VALUES
('Savanna');
```

UPDATING DATA

To update the data in a table, use the `UPDATE` command:
```
UPDATE Animal
SET
  Species = 'Duck',
  Name = 'Quack'
WHERE Id = 2;
```

DELETING DATA

To delete data from a table, use the `DELETE` command:
```
DELETE FROM Animal
WHERE Id = 1;
```

This deletes all rows satisfying the `WHERE` condition.
To delete all data from a table, use the `TRUNCATE TABLE` statement:
```
TRUNCATE TABLE Animal;
```

SQL SERVER CONVENTIONS

In SQL Server, use square brackets to handle table or column names that contain spaces, special characters, or reserved keywords. For example:
```
SELECT
 [First Name],
 [Age]
FROM [Customers];
```

Often, you refer to a table by its full name that consists of the schema name and the table name (for example, `AnimalSchema.Habitat`, `sys.databases`). For simplicity, we use plain table names in this cheat sheet.

THE GO SEPARATOR

In SQL Server, GO is a batch separator used to execute multiple SQL statements together. It is typically used in SQL Server Management Studio and similar tools.

SQL Server Cheat Sheet

LearnSQL.com

TEXT FUNCTIONS

Character strings are enclosed in single quotes:

```
SELECT 'Michael';
```

Unicode strings are enclosed in single quotes and prefixed with capital N:

```
SELECT N'Michél';
```

CONCATENATION

Use the CONCAT() function to concatenate two strings:

```
SELECT CONCAT(N'Hi ', N'there!');
-- result: Hi there!
```

CONCAT() treats NULL as an empty string:

```
SELECT CONCAT(N'Learn ', NULL, N'SQL.com');
-- result: LearnSQL.com
```

SQL Server allows specifying a separating character (separator) using the CONCAT_WS() function. The separator is placed between the concatenated values:

```
SELECT CONCAT_WS(' ', 1, N'Olivier',
N'Norris'); -- result: 1 Olivier Norris
```

FILTERING THE OUTPUT

To fetch the city names that are not Berlin:

```
SELECT Name
FROM City
WHERE Name != N'Berlin';
```

TEXT OPERATORS

To fetch the city names that start with a 'P' or end with an 's':

```
SELECT Name
FROM City
WHERE Name LIKE N'P%' OR Name LIKE N'%s';
```

To fetch the city names that start with any letter followed by 'ublin' (like Dublin in Ireland or Lublin in Poland):

```
SELECT Name
FROM City
WHERE Name LIKE N'_ublin';
```

OTHER USEFUL TEXT FUNCTIONS

To get the count of characters in a string:

```
SELECT LEN(N'LearnSQL.com'); -- result: 12
```

To convert all letters to lowercase:

```
SELECT LOWER(N'LEARNSQL.COM');
-- result: learnsql.com
```

To convert all letters to uppercase:

```
SELECT UPPER(N'LearnSQL.com');
-- result: LEARNSQL.COM
```

To get just a part of a string:

```
SELECT SUBSTRING(N'LearnSQL.com', 1, 5);
-- result: Learn
```

To replace a part of a string:

```
SELECT REPLACE(N'LearnSQL.com', 'SQL',
'Python');
-- result: LearnPython.com
```

NUMERIC FUNCTIONS

Use +, −, *, / to do some basic math.
To get the number of seconds in a week:

```
SELECT 60 * 60 * 24 * 7; -- result: 604800
```

In SQL Server, the division operator / performs an integer division on integer arguments. For example:

```
SELECT 25 / 4; -- result 6
```

To avoid the integer division, make sure at least one of the arguments is not an integer:

```
SELECT CAST(25 AS DECIMAL) / 4;
-- result 6.25
SELECT 25.0 / 4;
-- result 6.25
```

To get the remainder of a division:

```
SELECT MOD(13, 2); -- result: 1
```

To round a number to three decimal places:

```
SELECT ROUND(1234.56789, 3);
-- result: 1234.568
```

To round a number up:

```
SELECT CEILING(13.1), CEILING(-13.9);
-- result: 14, -13
```

To round a number down:

```
SELECT FLOOR(13.8), FLOOR(-13.2);
-- result: 13, -14
```

USEFUL NULL FUNCTIONS

To fetch the names of the cities whose rating values are not missing:

```
SELECT Name
FROM City
WHERE Rating IS NOT NULL;
```

COALESCE(x, y, ...)

To replace NULL in a query with something meaningful:

```
SELECT Domain,
       COALESCE(Domain, 'domain missing')
FROM Contacts;
```

The COALESCE() function takes any number of arguments and returns the value of the first argument that is not NULL.

NULLIF(x, y)

To save yourself from *division-by-0* errors:

```
SELECT LastMonth, ThisMonth,
  ThisMonth * 100.0 / NULLIF(LastMonth, 0)
    AS BetterByPercent
FROM VideoViews;
```

The NULLIF(x, y) function returns NULL if x equals y, else it returns the value of x value.

DATE AND TIME

There are 6 main time-related types in MySQL:

DATE – stores the year, month, and day in the YYYY-MM-DD format.
The supported range is '0001-01-01' to '9999-12-31'.

TIME – stores the hours, minutes, seconds, and nanoseconds in the HH:MM:SS[.nnnnnnn] format.
The supported range is '00:00:00.0000000' to '23:59:59.9999999'.

SMALLDATETIME – stores the date and time in the YYYY-MM-DD HH:MM:SS format.
The supported range is '1900-01-01' to '2079-06-06'.

DATETIME – stores the date and time in the YYYY-MM-DD HH:MM:SS[.nnn] format.
The supported range is '1753-01-01' to '9999-12-31'.

DATETIME2 – stores the date and time in the YYYY-MM-DD HH:MM:SS[.nnnnnnn] format.
The supported range is '0001-01-01 00:00:00.0000000' to '9999-12-31 23:59:59.9999999'.

DATETIMEOFFSET – stores the date and time in the YYYY-MM-DD HH:MM:SS[.nnnnnnn][+|-]hh:mm format.
The supported range is '0001-01-01 00:00:00.0000000' to '9999-12-31 23:59:59.9999999' in UTC.

WHAT TIME IS IT?

To get the current datetime without the time-zone offset:

```
SELECT GETDATE(); -- result: 2023-07-27
07:21:13.937
```

To get the current datetime without the time-zone offset in DATETIME2 data type (higher fractional seconds precision):

```
SELECT SYSDATETIME(); -- result: 2023-07-27
07:21:13.9398213
```

To get the current datetime in UTC:

```
SELECT GETUTCDATE(); -- result: 2023-07-27
07:21:13.937
```

or in datetime2 data type (higher fractional seconds precision):

```
SELECT SYSUTCDATETIME(); -- result: 2023-07-
27 07:21:13.9398213
```

To get the current datetime with the time-zone offset:

```
SELECT SYSDATETIMEOFFSET();
-- result: 2023-07-27 07:21:13.9398213 +00:00
```

CREATING VALUES

To create a date, time, or datetime, write the value as a string and cast it to the proper type.

```
SELECT CAST('2021-12-31' AS date),
  CAST('2021-12-31 23:59:29' AS DATETIME2);
```

DATE ARITHMETICS

To add or subtract from a DATE, use the DATEADD() function:

```
DATEADD(day, -3, '2014-04-05');
-- result: '2014-04-02'
```

To find the difference between two dates, use the DATEDIFF() function:

```
SELECT DATEDIFF(year, '2019-05-15',
  '2017-05-15');
-- result: -2
SELECT DATEDIFF(month, '2019-06-15',
  '2023-12-15');
-- result: 54
```

The supported date parts are: year, quarter, month, dayofyear, day, week, hour, minute, second, millisecond, microsecond, nanosecond.

EXTRACTING PARTS OF DATES

To extract a part of a date, use the functions YEAR(), MONTH(), or DAY():

```
SELECT YEAR(CAST('2021-12-31' AS date));
-- result: 2021
SELECT MONTH(CAST('2021-12-31' AS date));
-- result: 12
SELECT DAY(CAST('2021-12-31' AS date));
-- result: 31
```

You may also use the DATEPART() function:

```
SELECT DATEPART(year, '2013-09-15');
-- result: 2013
```

Supported date parts are: year, quarter, month, dayofyear, day, week, weekday, hour, minute, second, millisecond, microsecond, nanosecond, tzoffset, iso_week.

CHANGING THE TIME ZONE

Use AT TIME ZONE to convert a date and time value into a target time zone. You may use meaningful time zone names such as 'Pacific Standard Time'. SQL Server uses the names stored in the Windows Registry.

To add the target time-zone offset to a datetime value without offset information:

```
SELECT start_time AT TIME ZONE 'UTC';
```

To convert values between different time zones:

```
SELECT '2023-07-20 12:30:00'
  AT TIME ZONE 'UTC'
  AT TIME ZONE 'Eastern Standard Time';
-- result: 2023-07-20 08:30:00
```

Specify the known time-zone offset first (here, UTC) and then the time zone to which you want to convert.