INTRODUCTION

Everything is data, large amounts of data can't be stored in just simple spreadsheets,
so we need databases to make it more efficient. SQL is used to talk (ask questions) to the databases.
Also more secure than a spreadsheet.

People and web applications can use SQL to communicate with the databases.
Database Management System (DBMS) is used to handle all of these SQL queries,
including filtering out tainted queries that could negatively affect the database.

Relational database is one or more tables that are related.
Key-Value databases are organized by pairs of keys and values.
Column-Based is grouped by columns.
Graph is about the connections between data points.
Document is stored as entire documents.
The last 4 are known as NOSQL databases, while relational is SQL.

Server -> Database -> Schema -> Table (All can branch into more than 1)
Primary Key: A unique identifier in each row
Value = Cell, stores one type of data in a row given a column

Data Types: Numeric, Text/String, Date and Time

Data Definition Language (DDL): CREATE, ALTER, DROP
Data Manipulation Language (DML): INSERT, UPDATE, DELETE
Data Query Language: SELECT

Query = Question, a SQL query is asking the database a question, the output is the answer.

Why learn SQL?
Needed to talk to data/databases.
High demand, many, many tech jobs list it as a requirement.
Industry Standard.

SELECT

Individual SQL 'keywords' (SELECT, FROM, etc.) are called 'Clauses'

```
-- Select all columns from Table
SELECT * FROM Table
```

-- Single line comment
/*
Multi
Line
Comment
*/

-- Select by individual columns
SELECT Col1, Col2 [, ColN] FROM Table

-- No comma after last column name!

-- Select given (WHERE) certain condition
SELECT * FROM Table WHERE Column Conditional Value

-- Conditional may be =, >=, >, <, !=, etc.
-- Value may be any data type, as long as column matches data type (Or else warning)

-- Sort data by column (ORDER BY)
SELECT * FROM Table ORDER BY Column [ASC|DESC] [, Column2 [ASC|DESC], ColumnN [ASC|DESC]]

-- ASC = Ascending, default
-- DESC = Descending (Highest first)
-- Can add more columns to sort the sort (If Column contains duplicates, then sorts these by Column2, etc.)
-- Column order is important, sorts sequentially

-- Aggregate data with (GROUP BY)
SELECT [Column,] AGGR(Column) FROM Table GROUP BY Column

-- AGGR = Aggregation function, COUNT, SUM, etc.
-- For example: SELECT country, SUM(score) FROM customers GROUP BY country;

--Filter data after aggregation with HAVING
SELECT [Column,] AGGR(Column) FROM Table GROUP BY Column HAVING AGGR(Column) CONDITIONAL Value

-- Example: SELECT country, SUM(score) FROM customers GROUP BY country HAVING SUM(score) > 800;
-- WHERE after FROM, HAVING after GROUP BY
-- Example: SELECT country, SUM(score) FROM customers WHERE score > 400 GROUP BY country HAVING SUM(score) > 800;

-- Filter after select by DISTINCT (Removes duplicates)
SELECT DISTINCT Column FROM Table

-- Example: SELECT DISTINCT country FROM customers;
-- Can slow down query, don't use when not necessary

-- Filter after SELECT by TOP, restrict number of rows returned
SELECT TOP N Column FROM Table

-- !!! MySQL uses LIMIT at end of query instead!
SELECT * FROM Table LIMIT N;
-- Example: SELECT * FROM customers LIMIT 3;

Coding order:

SELECT DISTINCT AGGR(Column1), [Column2, ColumnN]
FROM Table
WHERE Column CONDITIONAL Value
GROUP BY Column1
HAVING AGGR(Column1) CONDITIONAL Value
ORDER BY Column1 [ASC|DESC][, Column2 [ASC|DESC], ColumnN [ASC|DESC]]
LIMIT N

Execution order:

1. FROM
2. WHERE
3. GROUP BY
4. HAVING
5. SELECT DISTINCT
6. ORDER BY
7. LIMIT

-- Multiple queries
-- Separate with semicolon on MySQL, executes both in order

-- Static (Fixed) Values
SELECT Value AS SomeName

AS = Alias for column name

Data Definition Language (DDL): CREATE, ALTER, DROP

-- Create new table
```
CREATE TABLE Table (
	Column1 Data_Type Constraint,
	Column2 Data_Type Constraint,
	...
	ColumnN Data_Type Constraint
)
```

```
-- Example:
CREATE TABLE persons (
	id int not null,
	person_name varchar(50) not null,
	birth_date date,
	phone varchar(15) not null,
	CONSTRAINT pk_persons PRIMARY KEY (id)
)
```

-- Show definition of table
```
DESCRIBE persons;
```

-- Alter table
```
ALTER TABLE Table ADD Column Data_Type Constraint

-- Example:
ALTER TABLE persons ADD email varchar(50) not null;
```

-- Remove column:
```
ALTER TABLE Table DROP COLUMN Column

-- Example:
ALTER TABLE persons DROP COLUMN email;
-- !!! Drops all data inside column!
```

-- Remove table:
```
DROP TABLE Table

-- Example:
DROP TABLE persons;
-- !!! Drops all data inside table!
```

Data Manipulation Language (DML): INSERT, UPDATE, DELETE

[INSERT]

-- Add rows to table
```
INSERT INTO Table (Column1[, Column2, ..., ColumnN]) VALUES (Value1[, Value2, ..., ValueN])
-- ! # Of Values must match number of Columns!

--Example:
INSERT INTO customers (id, first_name, country, score) VALUES
(6, 'Anya', 'Ukraine', NULL),
(7, 'Masha', 'Ukraine', NULL);

-- ! Order of columns matches order of values!
```

-- Automatically inserts NULL into unused columns, if no contraint about NULL.
INSERT INTO Table (Column1, Column2, (Exclude other columns)) VALUES (Blah, Blah)

-- Move data from one table to another
INSERT INTO persons (id, person_name, birth_date, phone)
SELECT id, first_name, NULL, 'Unknown' FROM customers

[UPDATE]

UPDATE Table SET Column1 = Value1[, Column2 = Value2, ..., ColumnN = ValueN] [WHERE CONDITION]
-- !!! WHERE clause is optional, but should always be used to avoid updating all rows unintentionally!!

-- If checking for null in conditional, use 'IS NULL', not '= NULL'

[DELETE]

DELETE FROM Table WHERE CONDITION
-- !!! Use WHERE CONDITION or big no-no!!! :O

-- Delete all data from table
DELETE FROM Table;

-- Or, without checking or logging to be faster:
TRUNCATE TABLE Table;

--

Intermediate Level

--

WHERE operators

Comparison:

= Equals
<> != Does Not Equal
> Greater Than
>= Greater Then Or Equal To
< Less Then
<= Less Than Or Equal To

Expression1 Operator Expression2

Examples:
SELECT * FROM Table WHERE Column1 = Column2
SELECT * FROM Table WHERE Column1 >= Value
SELECT * FROM Table WHERE FUNC(Column1) = Value
SELECT * FROM Table WHERE Column1 + Value1 > Value2
SELECT * FROM Table WHERE (SELECT AVG(Column1) FROM Table2) < Value

---

Logical (Grouping together conditions):

AND (All conditions must be TRUE)
OR (Either condition1 or condition2 must be TRUE)
NOT ('Negates' following condition)

Examples:

SELECT * FROM Table WHERE Column1 = Value1 AND Column2 != Value2
SELECT * FROM Table WHERE Column1 = Value1 OR Column2 = Value2
SELECT * FROM Table WHERE NOT (Column1 = Value1 AND Column2 > Value2)

Range:

BETWEEN (Checks if a value is within a range, boundaries inclusive)

Example:

SELECT * FROM Table WHERE Column BETWEEN Lower_Bound AND Upper_Bound

(Can also be accomplished this way:)
SELECT * FROM Table WHERE Column >= Lower_Bound AND Column <= Upper_Bound
(May be better due to that the inclusive bounds are clearly shown)

Membership:

IN (Is member of list)
NOT IN (Is not member of list)

Examples:

SELECT * FROM Table WHERE Column IN ('Value1', 'Value2'[, ValueN])
SELECT * FROM Table WHERE Column NOT IN ('Value1', 'Value2'[, ValueN])

(Can be accomplished with multiple comparisons and ORs, but this way is more concise)
(If using multiple same comparisons of a column with just different values connect by ORs, then IN is likely a better choice)

Search:

LIKE (To search for a pattern in text)

% Wildcard character (Can also match nothing, such as 'A%' matching 'A')
_ Exactly 1 of any character

SELECT * FROM Table WHERE Column LIKE 'A%'
-- Matches any starting with 'A'

SELECT * FROM Table WHERE Column LIKE '%a'
-- Matches any ending with 'a'

-- '%c%' Contains a 'c' anywhere

Intermediate Level

The following examples all use the database 'MyDatabase' from the collection 'sql-ultimate-course-main'

JOIN puts together by columns, side to side
Need common key column to combine

Recombine Data "Big Picture"

Data Enrichment "Getting extra Data"
Master Table <- JOIN Reference Table

Also, Check For Existence "filtering", filter data based on join,
such as filtering by only customers who ordered something

Not specifying a JOIN TYPE default is INNER
! Should always specify to make clearer!

Joins

LEFT TABLE ( ), RIGHT TABLE ( ), BOTH (( ))
(_) Matching, ( ) Unmatching

Basic Types:

INNER JOIN ((_))
Only matching rows from both Tables
SELECT * FROM TableA INNER JOIN TableB ON TableA.Key_Column = TableB.Key_Column

Example:

SELECT * FROM customers INNER JOIN orders ON customer_id = id;

-- Only customers that have placed an order (Exists in orders table)

id	first_name	country	score	order_id	customer_id	order_date	sales
1	Maria	Germany	350	1001	1	2021-01-11	35
2	John	USA	900	1002	2	2021-04-05	15
3	Georg	UK	750	1003	3	2021-06-18	20
6	Anya	Ukraine	350	1004	6	2021-08-31	10

-- Get only the needed columns, avoid redundancy:

SELECT id, first_name, country, score, order_id, order_date, sales
FROM customers INNER JOIN orders ON customer_id = id;

id	first_name	country	score	order_id	order_date	sales
1	Maria	Germany	350	1001	2021-01-11	35
2	John	USA	900	1002	2021-04-05	15
3	Georg	UK	750	1003	2021-06-18	20
6	Anya	Ukraine	350	1004	2021-08-31	10

Column Ambiguity: Sometimes columns from 2 or more tables will have the same name.
Use the table Name. Column to avoid the error:

SELECT TableA.id, TableB.id FROM TableA INNER JOIN TableB ON TableA.id = TableB.id

Good practice to use table name before any column for many different columns in Query

Also, can use alias to save typing:

SELECT TA.id, TB.id FROM TableA AS TA INNER JOIN TableB AS TB ON TA.id = TB.id

Should definitely make names easy to use, but conveys information on what table they are an alias for.

FULL JOIN (_(_)_)
Returns all rows from both tables

Order of tables doesn't matter

SELECT * FROM TableA AS TA FULL JOIN TableB AS TB ON TA.Key_Column = T
B.Key_Column

====================
!!! **MySQL Problem** !!!

From Gemini:

"Yes, Full Outer Join works differently in MySQL because MySQL does not natively support
the FULL OUTER JOIN keyword as a direct operation like some other SQL database systems
(e.g., PostgreSQL, SQL Server, Oracle).
Instead of a direct FULL OUTER JOIN, you need to emulate its behavior using a combination
of LEFT JOIN, RIGHT JOIN, and the UNION operator."

Given Example:
"
SELECT columns
FROM table1
LEFT JOIN table2 ON table1.common_column = table2.common_column
UNION
SELECT columns
FROM table1
RIGHT JOIN table2 ON table1.common_column = table2.common_column
WHERE table1.common_column IS NULL;
"

"While the functionality of a full outer join is achievable in MySQL,
the syntax and method of execution are different compared to databases
that offer a direct FULL OUTER JOIN keyword."

Example:

SELECT * FROM customers LEFT JOIN orders ON id = customer_id
UNION
SELECT * FROM customers RIGHT JOIN orders ON id = customer_id WHERE id IS NULL;

id	first_name	country	score	order_id	customer_id	order_date	sales
1	Maria	Germany	350	1001	1	2021-01-11	35
2	John	USA	900	1002	2	2021-04-05	15
3	Georg	UK	750	1003	3	2021-06-18	20
4	Martin	Germany	500	NULL	NULL	NULL	NULL
6	Anya	Ukraine	350	1004	6	2021-08-31	10
7	Masha	Ukraine	300	NULL	NULL	NULL	NULL
NULL	NULL	NULL	NULL	1005	8	2021-09-15	25

LEFT JOIN (_(_))
Returns all rows from left and only matching rows from right.

SELECT TA.id, TB.id FROM TableA AS TA LEFT JOIN TableB AS TB ON TA.id = TB.id

Example:

SELECT id, first_name, country, score, order_id, order_date, sales
FROM customers LEFT JOIN orders ON customer_id = id;

id	first_name	country	score	order_id	order_date	sales
1	Maria	Germany	350	1001	2021-01-11	35
2	John	USA	900	1002	2021-04-05	15
3	Georg	UK	750	1003	2021-06-18	20
4	Martin	Germany	500	NULL	NULL	NULL
6	Anya	Ukraine	350	1004	2021-08-31	10
7	Masha	Ukraine	300	NULL	NULL	NULL

There are NULLS in columns that did not exist in the right table (orders),
but all the rest of the columns from the left table (customers) are there.

Example with all columns:

SELECT * FROM customers LEFT JOIN orders ON customer_id = id;

id	first_name	country	score	order_id	customer_id	order_date	sales
1	Maria	Germany	350	1001	1	2021-01-11	35
2	John	USA	900	1002	2	2021-04-05	15
3	Georg	UK	750	1003	3	2021-06-18	20
4	Martin	Germany	500	NULL	NULL	NULL	NULL
6	Anya	Ukraine	350	1004	6	2021-08-31	10
7	Masha	Ukraine	300	NULL	NULL	NULL	NULL

Puts left table columns first, no matter what, then sees if match in right table.
Fills rest of right table columns with NULL if no match.

RIGHT JOIN ((_)_)
Returns all rows from the right and only matching rows from left

SELECT TA.id, TB.id FROM TableA AS TA RIGHT JOIN TableB AS TB ON TA.id = TB.id

Example:

(Updated orders table first so this example would be clearer):
INSERT INTO orders (order_id, customer_id, order_date, sales) VALUES (1005, 8, '2021-09-15', 25);

SELECT id, first_name, country, score, order_id, order_date, sales
FROM customers RIGHT JOIN orders ON customer_id = id;

id	first_name	country	score	order_id	order_date	sales
1	Maria	Germany	350	1001	2021-01-11	35
2	John	USA	900	1002	2021-04-05	15
3	Georg	UK	750	1003	2021-06-18	20
6	Anya	Ukraine	350	1004	2021-08-31	10
NULL	NULL	NULL	NULL	1005	2021-09-15	25

Example with all columns:

SELECT * FROM customers RIGHT JOIN orders ON customer_id = id;

id	first_name	country	score	order_id	customer_id	order_date	sales
1	Maria	Germany	350	1001	1	2021-01-11	35
2	John	USA	900	1002	2	2021-04-05	15
3	Georg	UK	750	1003	3	2021-06-18	20
6	Anya	Ukraine	350	1004	6	2021-08-31	10
NULL	NULL	NULL	NULL	1005	8	2021-09-15	25

Puts right table columns first, no matter what, then sees if match in left table.
Fills rest of left table columns with NULL if no match.

Anti JOINS

!!! **MySQL Note** !!!

MySQL doesn't have an actual ANTI keyword for joins, but the same results can be gathered using the kind of syntax shown in these examples:

"
AI Overview
Yes, MySQL supports anti-joins. While there isn't a specific ANTI JOIN keyword like some other database systems, the functionality of an anti-join can be achieved through various SQL constructs that MySQL's optimizer recognizes and can optimize into an anti-join strategy.

Here are the common ways to implement an anti-join in MySQL:

LEFT JOIN with WHERE IS NULL: This is a widely used and effective method. You perform a LEFT JOIN from the "left" table to the "right" table and then filter the results in the WHERE clause to include only those rows from the left table where there is no match in the right table (i.e., the joined columns from the right table are NULL).

Code

```
SELECT t1.*
FROM table1 t1
LEFT JOIN table2 t2 ON t1.id = t2.id
WHERE t2.id IS NULL;
```

NOT EXISTS subquery: This approach checks for the non-existence of a correlated subquery.

Code

```
SELECT t1.*
FROM table1 t1
WHERE NOT EXISTS (SELECT 1 FROM table2 t2 WHERE t1.id = t2.id);
```

NOT IN subquery: While NOT IN can be less efficient with nullable columns, MySQL's optimizer often rewrites NOT IN subqueries as NOT EXISTS for better performance.

Code

```
SELECT t1.*
FROM table1 t1
WHERE t1.id NOT IN (SELECT t2.id FROM table2 t2);
```

MySQL's optimizer can transform these constructs into an internal anti-join operation, especially when dealing with NOT EXISTS and NOT IN subqueries, to improve query performance.
"

=================

LEFT ANTI JOIN (_( ))
Returns rows from left that has no match in right

SELECT * FROM TableA AS TA LEFT JOIN TableB AS TB ON TA.Key_Column = TB.Key_Column WHERE TB.Key_Column IS NULL

Example:

-- "Customers who haven't ordered anything"

SELECT * FROM customers LEFT JOIN orders ON customer_id = id WHERE customer_id IS NULL;

id	first_name	country	score	order_id	customer_id	order_date	sales
4	Martin	Germany	500	NULL	NULL	NULL	NULL
7	Masha	Ukraine	300	NULL	NULL	NULL	NULL

RIGHT ANTI JOIN (( )_)
Returns rows from right that has no match in left

SELECT * FROM TableA AS TA RIGHT JOIN TableB AS TB ON TA.Key_Column =
TB.Key_Column
WHERE TA.Key_Column IS NULL

Example:

-- "Orders that don't have a customer associated with it"

SELECT * FROM customers RIGHT JOIN orders ON customer_id = id WHERE id IS NULL;

id	first_name	country	score	order_id	customer_id	order_date	sales
NULL	NULL	NULL	NULL	1005	8	2021-09-15	25

Or With LEFT JOIN:

SELECT * FROM orders LEFT JOIN customers ON customer_id = id WHERE id IS NULL;

order_id	customer_id	order_date	sales	id	first_name	country	score
1005	8	2021-09-15	25	NULL	NULL	NULL	NULL

FULL ANTI JOIN (_( )_)
Only the rows that don't match in either table

SELECT * FROM TableA AS TA FULL JOIN TableB AS TB ON TA.Key_Column = TB.Key_Column
WHERE TA.Key_Column IS NULL OR TB.Key_Column IS NULL

!!! **MySQL** !!!

No FULL JOIN, use this instead:

SELECT * FROM TableA AS TA LEFT JOIN TableB AS TB ON TA.Key_Column = TB.Key_Column
WHERE TB.Key_Column IS NULL
UNION

SELECT * FROM TableA AS TA RIGHT JOIN TableB AS TB ON TA.Key_Column =
TB.Key_Column
WHERE TA.Key_Column IS NULL

Example:

-- "Customers without orders and orders without customers"

SELECT * FROM customers LEFT JOIN orders ON id = customer_id WHERE customer_id IS NULL
UNION
SELECT * FROM customers RIGHT JOIN orders ON id = customer_id WHERE id IS NULL;

id	first_name	country	score	order_id	customer_id	order_date	sales
4	Martin	Germany	500	NULL	NULL	NULL	NULL
7	Masha	Ukraine	300	NULL	NULL	NULL	NULL
NULL	NULL	NULL	NULL	1005	8	2021-09-15	25

-- "Only customers who have placed an order" WITHOUT INNER JOIN:

SELECT * FROM orders LEFT JOIN customers ON customer_id = id WHERE id IS NOT NULL;

order_id	customer_id	order_date	sales	id	first_name	country	score
1001	1	2021-01-11	35	1	Maria	Germany	350
1002	2	2021-04-05	15	2	John	USA	900
1003	3	2021-06-18	20	3	Georg	UK	750
1004	6	2021-08-31	10	6	Anya	Ukraine	350

CROSS JOIN
Combines every row from left with every row from right
All possible combinations (Cartesian Join)

Good for testing purposes.

SELECT * FROM TableA CROSS JOIN TableB

Example:

SELECT * FROM customers CROSS JOIN orders;

id	first_name	country	score	order_id	customer_id	order_date	sales
1	Maria	Germany	350	1005	8	2021-09-15	25
1	Maria	Germany	350	1004	6	2021-08-31	10
1	Maria	Germany	350	1003	3	2021-06-18	20
1	Maria	Germany	350	1002	2	2021-04-05	15
1	Maria	Germany	350	1001	1	2021-01-11	35
2	John	USA	900	1005	8	2021-09-15	25
2	John	USA	900	1004	6	2021-08-31	10
2	John	USA	900	1003	3	2021-06-18	20
2	John	USA	900	1002	2	2021-04-05	15
2	John	USA	900	1001	1	2021-01-11	35
3	Georg	UK	750	1005	8	2021-09-15	25
3	Georg	UK	750	1004	6	2021-08-31	10
3	Georg	UK	750	1003	3	2021-06-18	20
3	Georg	UK	750	1002	2	2021-04-05	15
3	Georg	UK	750	1001	1	2021-01-11	35
4	Martin	Germany	500	1005	8	2021-09-15	25
4	Martin	Germany	500	1004	6	2021-08-31	10
4	Martin	Germany	500	1003	3	2021-06-18	20
4	Martin	Germany	500	1002	2	2021-04-05	15
4	Martin	Germany	500	1001	1	2021-01-11	35
6	Anya	Ukraine	350	1005	8	2021-09-15	25
6	Anya	Ukraine	350	1004	6	2021-08-31	10
6	Anya	Ukraine	350	1003	3	2021-06-18	20
6	Anya	Ukraine	350	1002	2	2021-04-05	15
6	Anya	Ukraine	350	1001	1	2021-01-11	35
7	Masha	Ukraine	300	1005	8	2021-09-15	25
7	Masha	Ukraine	300	1004	6	2021-08-31	10
7	Masha	Ukraine	300	1003	3	2021-06-18	20

7	Masha	Ukraine	300	1002	2	2021-04-05	15
7	Masha	Ukraine	300	1001	1	2021-01-11	35

How to choose between JOIN types

Only matching? INNER JOIN

All Rows?
    One Side more important (Master table)? LEFT JOIN
    Both Important? FULL JOIN

Only unmatching data?
    One side more important (Master table)? LEFT ANTI JOIN
    Both important? FULL ANTI JOIN

RIGHT JOIN? Not really any need. :ON

Multi_Table JOINs

Master TableA LEFT JOIN TableB ON blah LEFT JOIN TableC ON blah etc.

WHERE clause to control final result (What to keep)

Get complete big picture.

Maybe only matching data important where there is no master table?
Use INNER JOINs instead (Overlapping between all tables)

!!! **CHALLENGE** !!!

-- Using SalesDB, retrieve a list of all orders, along with the related
-- customer, product, and employee details
-- For each order, display:
-- Order ID, Customer's name, product name, sales amount, product price, and saleperson's name

----

Query:

SELECT o.orderid, c.firstname AS cust_first_name, c.lastname AS cust_last_name,
p.product, o.sales, p.price, e.firstname AS empl_first_name, e.lastname AS empl_last_name
FROM orders AS o
LEFT JOIN customers AS c ON c.customerid = o.customerid
LEFT JOIN products AS p ON p.productid = o.productid
LEFT JOIN employees AS e ON e.employeeid = o.salespersonid;

orderid	cust_first_name	cust_last_name	product	sales	price	empl_first_name	empl_last_name
1	Kevin	Brown	Bottle	10	10	Mary	NULL
2	Mary	NULL	Tire	15	15	Mary	NULL
3	Jossef	Goldberg	Bottle	20	10	Carol	Baker
4	Jossef	Goldberg	Gloves	60	30	Mary	NULL
5	Kevin	Brown	Caps	25	25	Carol	Baker
6	Mary	NULL	Caps	50	25	Carol	Baker
7	Jossef	Goldberg	Tire	30	15	Frank	Lee
8	Mark	Schwarz	Bottle	90	10	Mary	NULL
9	Kevin	Brown	Bottle	20	10	Mary	NULL
10	Mary	NULL	Tire	60	15	Carol	Baker

'orders' is master table, contains foreign keys found as all the primary keys in other tables, should use as starting point.

SET puts together by rows, top to bottom
Need same number of columns to combine

Sets

(Examples use the 'salesdb' database)

SELECT blah SET_OPERATOR SELECT blah

Rules:

SET operator can be used in almost all clauses, except for ORDER BY,
can only use this once at end of query.

The number of columns in each query must be the same.

The data types of columns in each query must match.

The order of columns in each query must be the same.

The column names in the result set are determined by the
column names specified in the first query. (Controls aliases)

Even if all rules are met and SQL shows no errors,
the result may be incorrect. Incorrect column selection leads to
inaccurate results. You are responsible for mapping the correct information!

Basic Types:

UNION
Returns all distinct rows from both queries.
Removes duplicate rows from results.

(_( )_)

SELECT * FROM TableA UNION SELECT * FROM TableB

Example:

```
SELECT firstname, lastname FROM customers
UNION
SELECT firstname, lastname FROM employees
ORDER BY firstname;
```

firstname	lastname
Anna	Adams
Carol	Baker
Frank	Lee
Jossef	Goldberg
Kevin	Brown
Mark	Schwarz
Mary	NULL

Michael	Ray

UNION All
Returns all rows from both queries, including duplicates.

(_(_)_)

UNION with duplicates, not distinct (everything).

Faster, since there is no need to look for duplicates.
Sure there are no duplicates to begin with? UNION ALL is faster choice!

SELECT * FROM TableA UNION ALL SELECT * FROM TableB

Example:

SELECT firstname, lastname FROM customers
UNION ALL
SELECT firstname, lastname FROM employees;

firstname	lastname
Jossef	Goldberg
Kevin	Brown
Mary	NULL
Mark	Schwarz
Anna	Adams
Frank	Lee
Kevin	Brown
Mary	NULL
Michael	Ray
Carol	Baker

EXCEPT (MINUS)
Return all distinct rows from the first query that are not
found in the second query.

(_( ))

Everything in first query, EXCEPT (MINUS) the rows found in 2nd query.
If row in 2nd query matches ones in 1st query, removes them from result.
If row in 2nd query doesn't match, still don't include.
Results in whatever rows are remaining.

Order of queries is important!
Grabs from 1st query, uses 2nd query only as a check to filter out rows.

Example:

SELECT firstname, lastname FROM customers
EXCEPT
SELECT firstname, lastname FROM employees;

firstname	lastname
Jossef	Goldberg
Mark	Schwarz
Anna	Adams

INTERSECT
Return only the distinct rows that are common in both queries.

((_))

The duplicates that would be removed in UNION.

Example:

SELECT firstname, lastname FROM customers
INTERSECT
SELECT firstname, lastname FROM employees;

firstname	lastname
Kevin	Brown
Mary	NULL

UNION Use Cases

Combine similar information before analyzing the data.

Can get inconsistent results if multiple queries and have to change them regularly.

Instead, UNION them all into a new table, then apply queries to the new table.

Sometimes data is divided into multiple tables to optimize performance.
May need to combine them in order to work with all of them more easily.

Should never use '*' to list all columns, instead use the names explicitly.
What if some column name changes? Or a new column is added? Would be a problem to map data with UNION using '*'.

Add extra information by using an alias to make clearer which originating table the data belongs to:

```
SELECT 'TableA' AS SourceTable ...
UNION
SELECT 'TableB' AS SourceTable ...
```

Example:

```
SELECT 'CurrentOrders' AS SourceTable, orderid, productid, customerid
FROM orders
UNION
SELECT 'OldOrders' AS SourceTable, orderid, productid, customerid
FROM orders_archive;
```

SourceTable	orderid	productid	customerid
CurrentOrders	1	101	2
CurrentOrders	2	102	3
CurrentOrders	3	101	1
CurrentOrders	4	105	1
CurrentOrders	5	104	2
CurrentOrders	6	104	3
CurrentOrders	7	102	1
CurrentOrders	8	101	4
CurrentOrders	9	101	2
CurrentOrders	10	102	3
OldOrders	1	101	2
OldOrders	2	102	3
OldOrders	3	101	1
OldOrders	4	105	1
OldOrders	5	104	2

OldOrders	6	104	3
OldOrders	6	101	3
OldOrders	7	102	3

Delta Detection

Identifying the differences or changes (delta) between two batches of data.

Gets set of new orders, first time.
Gets set of second orders, now use EXCEPT to get the new data by comparison
to then put the unique data into the 'Data Warehouse'.

Data Completeness Check

EXCEPT operator can be used to compare tables to detect discrepancies between databases.

Migrate Table from DatabaseA to DatabaseB.
Use EXCEPT between Table in DatabaseA and DatabaseB, if they match exactly,
then the result will be empty.
Use same DatabaseB -> DatabaseA.
If both results in empty, Tables match exactly.

Summary

Combine the results of multiple queries into a single result set.

Types: UNION, UNION ALL, EXCEPT, INTERSECT.

Rules:
    Same # of columns, Data Types, order of columns.
    1st query controls column names.

Use Cases:
    Combine information (UNION / UNION ALL)
    Delta Detection (EXCEPT)
    Data Completeness Check (EXCEPT)

SQL Row-Level Functions

Data manipulation
Analyze data
Clean data
Data transformation

Functions can be used for these tasks.

Input -> Function -> Output

Single Row Functions, IO 1 Value

Manipulate and prepare for Multi-Row

String
Numeric
Date & Time
NULL

Multi-Row, Input many values, output 1 values:

Aggregate
Window

Functions can be nested:

LOWER(LEFT(StringColumn, 2))

String Functions

Manipulation:

CONCAT
Concatenate 2+ strings

Example:

SELECT CONCAT(firstname, ' ', lastname) AS Name FROM employees LIMIT 1;

Name
Frank Lee

UPPER
Convert all characters in string to uppercase

Example:

SELECT UPPER(CONCAT(firstname, ' ', lastname)) AS Name FROM employees LIMIT 1;

Name
FRANK LEE

LOWER
Convert all characters in string to lowercase

Example:

SELECT LOWER(CONCAT(firstname, ' ', lastname)) AS Name FROM employees LIMIT 1;

Name
frank lee

Combined Example:

SELECT CONCAT(UPPER(firstname), ' ', LOWER(lastname)) AS Name FROM employees LIMIT 1;

Name
FRANK lee

TRIM
Remove leading and trailing spaces in string

Example:

SELECT CONCAT('   ', firstname, ' ', lastname, '   ') AS Name FROM employees LIMIT 1;

Name
Frank Lee

SELECT TRIM(CONCAT('   ', firstname, ' ', lastname, '   ')) AS Name FROM employees LIMIT 1;

Name
Frank Lee

In order to find first names that have white space in them:
SELECT first_name FROM customers WHERE first_name != TRIM(first_name)

If flag > 0, has white space
LENGTH(first_name) - LENGTH(TRIM(first_name)) AS flag

REPLACE
Replaces specific character with a new character

Example:

Remove char:
SELECT birthdate, REPLACE(birthdate, '-', '') FROM employees LIMIT 1;

birthdate	REPLACE(birthdate, '-', '')
1988-12-05	19881205

Replace char:
SELECT birthdate, REPLACE(birthdate, '-', '/') FROM employees LIMIT 1;

birthdate	REPLACE(birthdate, '-', '/')
1988-12-05	1988/12/05

Replace substring:
SELECT 'test.txt' AS old_filename, REPLACE('test.txt', '.txt', '.sql') AS new_filename;

old_filename	new_filename
test.txt	test.sql

Calculation:

LENGTH
Returns number of characters in value

Example:
SELECT firstname, LENGTH(firstname) AS char_count FROM employees LIMIT 1;

firstname	char_count

Frank	5

String Extraction:

LEFT

Example:
SELECT firstname, LEFT(firstname, 2) AS left2 FROM employees LIMIT 1;

firstname	left2
Frank	Fr

RIGHT

Example:
SELECT firstname, RIGHT(firstname, 2) AS left2 FROM employees LIMIT 1;

firstname	left2
Frank	nk

SUBSTRING
SUBSTRING(StringColumn, StartPos, Length)

Example:

No length? Everything from StartPos
SELECT firstname, SUBSTRING(firstname, 3) AS sub FROM employees LIMIT 1;

firstname	sub
Frank	ank

SELECT firstname, SUBSTRING(firstname, 3, 2) AS sub FROM employees LIMIT 1;

firstname	sub

Frank	an

Remove all first chars:
SELECT firstname, SUBSTRING(firstname, 2) AS sub FROM employees LIMIT 1;

firstname	sub
Frank	rank

Numeric Functions

ROUND

SELECT 3.516, ROUND(3.516, 2);

3.516	ROUND(3.516, 2)
3.516	3.52

SELECT 3.516, ROUND(3.516, 1);

3.516	ROUND(3.516, 1)
3.516	3.5

SELECT 3.516, ROUND(3.516, 0);

3.516	ROUND(3.516, 0)
3.516	4

ABS

SELECT -22, ABS(-22);

-22	ABS(-22)
-22	22

SQL Row-Level Functions Part 2

Date & Time Functions

Date:
yyyy-mm-dd

Time:
hh:mm:ss

Timestamp (Datetime2 in SQL Server):
yyyy-mm-dd hh:mm:ss

(For Sample Purposes):

SELECT orderdate, shipdate, creationtime FROM orders LIMIT 5;

orderdate	shipdate	creationtime
2025-01-01	2025-01-05	2025-01-01 12:34:56
2025-01-05	2025-01-10	2025-01-05 23:22:04
2025-01-10	2025-01-25	2025-01-10 18:24:08
2025-01-20	2025-01-25	2025-01-20 05:50:33
2025-02-01	2025-02-05	2025-02-01 14:02:41

Sources for Dates:

Date Value Column
Hardcoded Date String
GETDATE() Function (For SQL Server, use NOW() or CURRENT_TIMESTAMP() for MySQL)

Examples:

SELECT orderdate AS DateColumn, '2025-08-20' AS HardCoded, NOW() AS DateFunction
FROM orders
LIMIT 3;

DateColumn	HardCoded	DateFunction
2025-01-01	2025-08-20	2025-09-03 12:46:14
2025-01-05	2025-08-20	2025-09-03 12:46:14
2025-01-10	2025-08-20	2025-09-03 12:46:14

Date & Time Functions

Extract parts of dates.
Change date format.
Do date calculations.
Validate date (Check if real) (Boolean)

Part Extraction

DAY

MONTH

YEAR

Example:

```
SELECT orderdate AS DateColumn, DAY(orderdate), MONTH(orderdate), YEAR(orderdate)
FROM orders
LIMIT 3;
```

DateColumn	DAY(orderdate)	MONTH(orderdate)	YEAR(orderdate)
2025-01-01	1	1	2025
2025-01-05	5	1	2025
2025-01-10	10	1	2025

DATEPART (Use EXTRACT(unit FROM Date) in MySQL)

Example (SQL Server):
DATEPART(part, date)

part: month, mm, week, etc.

Example:

SELECT EXTRACT(week FROM orderdate) AS Week, EXTRACT(quarter FROM orderdate) AS Quarter

FROM orders
LIMIT 3;

Week	Quarter
0	1
1	1
1	1

DATENAME (DAYNAME, MONTHNAME for MySQL)
If grouping sales by month, looks much nicer given names such as January, etc.

Example:

SELECT DAYNAME(orderdate) AS day, MONTHNAME(orderdate) AS month FROM orders LIMIT 3;

day	month
Wednesday	January
Sunday	January
Friday	January

DATETRUNC
Truncates the date to a specific part.

DATETRUNC(part, date)

Example:

!!!!!!!!!!!!!!!!!!!!!!!!!
Note about MySQL
!!!!!!!!!!!!!!!!!!!!!!!!!

AI Overview
MySQL does not have a direct equivalent to the DATE_TRUNC() function found in other SQL dialects like PostgreSQL or SQL Server. However, you can achieve similar functionality using a combination of MySQL's date and time functions, primarily DATE_FORMAT() and DATE().

Here's how to emulate DATE_TRUNC() for various time units in MySQL:

----

1. Truncate to Day (Remove Time Component):
Use the DATE() function to extract only the date part from a DATETIME or TIMESTAMP column.

Code

```
SELECT DATE(your_datetime_column) FROM your_table;
```

----

2. Truncate to Month:
Use DATE_FORMAT() to format the date to the first day of the month.

Code

```
SELECT DATE_FORMAT(your_datetime_column, '%Y-%m-01') FROM your_table;
```

Example:

```
SELECT DATE_FORMAT(orderdate, '%Y-%m-01') FROM orders LIMIT 3;
```

DATE_FORMAT(orderdate, '%Y-%m-01')
2025-01-01
2025-01-01
2025-01-01

----

3. Truncate to Year:
Use DATE_FORMAT() to format the date to the first day of the year.

Code

```
SELECT DATE_FORMAT(your_datetime_column, '%Y-01-01') FROM your_table;
```

Example:

```
SELECT DATE_FORMAT(orderdate, '%Y-01-01') FROM orders LIMIT 3;
```

DATE_FORMAT(orderdate, '%Y-01-01')
2025-01-01
2025-01-01
2025-01-01

4. Truncate to Hour/Minute/Second:
Use DATE_FORMAT() with the appropriate format specifiers to include only the desired precision. For example, to truncate to the hour:

Code

```
SELECT DATE_FORMAT(your_datetime_column, '%Y-%m-%d %H:00:00') FROM your_table;
```

Example:

```
SELECT DATE_FORMAT(orderdate, '%Y-%m-%d %H:00:00') FROM orders LIMIT 3;
```

DATE_FORMAT(orderdate, '%Y-%m-%d %H:00:00')
2025-01-01 00:00:00
2025-01-05 00:00:00
2025-01-10 00:00:00

In summary: While MySQL lacks a single DATE_TRUNC() function, the DATE() and DATE_FORMAT() functions provide the necessary tools to achieve the same date truncation effects by manipulating the date and time format.

Can count orders given creation time, but if no truncate, will likely only get 1 per count.
Truncate sets of dates, get better count within certain Dates.

Example:

```
SELECT COUNT(*) AS OrderCounts, DATE_FORMAT(orderdate, '%Y-01-01') AS TruncOrderDate
FROM orders
GROUP BY TruncOrderDate;
```

OrderCounts	TruncOrderDate

10	2025-01-01

EOMONTH (LAST_DAY() In MySQL)
Returns the last day of a month.

Example:

SELECT LAST_DAY(orderdate) FROM orders LIMIT 3;

LAST_DAY(orderdate)
2025-01-31
2025-01-31
2025-01-31

Note:
Can use TRUNCATE to get the first day of the month.

Use Cases:

Doing data aggregations and reporting:
Sales by year
Sales by month
etc.

Example:

SELECT YEAR(orderdate), COUNT(*) FROM orders GROUP BY YEAR(orderdate);

YEAR(orderdate)	COUNT(*)
2025	10

Filtering:

SELECT orderid, productid, customerid, salespersonid, orderdate
FROM orders
WHERE MONTH(orderdate) = 2
LIMIT 10;

orderid	productid	customerid	salespersonid	orderdate
5	104	2	5	2025-02-01
6	104	3	5	2025-02-05
7	102	1	1	2025-02-15
8	101	4	3	2025-02-18

!!! **Important Note** !!!

Filtering data using an integer is faster than using a string!

Date Extraction Function Outputs:

FUNCTION	Data Type
DAY MONTH YEAR DATEPART	INT
DATENAME	String
DATETRUNC	DATETIME
EOMONTH	DATE

Which part do I want to extract?

Day, Month?
    Numeric?
        DAY() MONTH()
    Full Name?
        DATENAME()

Year?
    YEAR()

Other Parts?
    DATEPART()

Format & Casting

FORMAT (DATE_FORMAT in MySQL)

Date Format specifiers:

Case sensitive!
ISO 8601: yyyy-MM-dd HH:mm:ss (SQL uses this)
USA Standard: MM-dd-yyyy
European Standard: dd-MM-yyyy

MM/dd/yyyy 01/01/25
MMM yyyy Jan 2025

In SQL Server FORMAT:
FORMAT(date_value, format [, culture])
Where culture can be 'ja-JP', 'fr-FR', etc. (Default 'en-US')

Example:

!!! **MySQL** !!!
%M is for named month
%m is for month as number

SELECT DATE_FORMAT(orderdate, '%m/%d/%y') FROM orders LIMIT 3;

DATE_FORMAT(orderdate, '%m/%d/%y')
01/01/25
01/05/25
01/10/25

%D is for the day, like '1st':

SELECT DATE_FORMAT(orderdate, '%M %D, %y') FROM orders LIMIT 3;

DATE_FORMAT(orderdate, '%M %D, %y')
January 1st, 25
January 5th, 25
January 10th, 25

%W is for the named weekday:

SELECT DATE_FORMAT(orderdate, '%W, %M %D, %y') FROM orders LIMIT 3;

DATE_FORMAT(orderdate, '%W, %M %D, %y')
Wednesday, January 1st, 25
Sunday, January 5th, 25
Friday, January 10th, 25

Format time as well:

SELECT DATE_FORMAT(creationtime, '%d-%b-%Y %h:%i %p') FROM orders LIMIT 3;

DATE_FORMAT(creationtime, '%d-%b-%Y %h:%i %p')
01-Jan-2025 12:34 PM
05-Jan-2025 11:22 PM
10-Jan-2025 06:24 PM

Abbreviated weekday and month names:

SELECT DATE_FORMAT(orderdate, '%a, %b %D, %Y') FROM orders LIMIT 3;

DATE_FORMAT(orderdate, '%a, %b %D, %Y')
Wed, Jan 1st, 2025
Sun, Jan 5th, 2025
Fri, Jan 10th, 2025

SELECT DATE_FORMAT(creationtime, 'Day %a %b %y %h:%i:%s %p') FROM orders LIMIT 3;

DATE_FORMAT(creationtime, 'Day %a %b %y %h:%i:%s %p')
Day Wed Jan 25 12:34:56 PM
Day Sun Jan 25 11:22:04 PM
Day Fri Jan 25 06:24:08 PM

Complete list of specifiers:

Specifier Description
%a Abbreviated weekday name (Sun..Sat)
%b Abbreviated month name (Jan..Dec)
%c Month, numeric (0..12)
%D Day of the month with English suffix (0th, 1st, 2nd, 3rd,)
%d Day of the month, numeric (00..31)
%e Day of the month, numeric (0..31)
%f Microseconds (000000..999999)
%H Hour (00..23)
%h Hour (01..12)
%I Hour (01..12)
%i Minutes, numeric (00..59)
%j Day of year (001..366)
%k Hour (0..23)
%l Hour (1..12)
%M Month name (January..December)
%m Month, numeric (00..12)
%p AM or PM
%r Time, 12-hour (hh:mm:ss followed by AM or PM)
%S Seconds (00..59)
%s Seconds (00..59)
%T Time, 24-hour (hh:mm:ss)
%U Week (00..53), where Sunday is the first day of the week; WEEK() mode 0
%u Week (00..53), where Monday is the first day of the week; WEEK() mode 1
%V Week (01..53), where Sunday is the first day of the week; WEEK() mode 2; used with %X
%v Week (01..53), where Monday is the first day of the week; WEEK() mode 3; used with %x
%W Weekday name (Sunday..Saturday)
%w Day of the week (0=Sunday..6=Saturday)
%X Year for the week where Sunday is the first day of the week, numeric, four digits; used with %V
%x Year for the week, where Monday is the first day of the week, numeric, four digits; used with
%v
%Y Year, numeric, four digits
%y Year, numeric (two digits)
%% A literal % character
%x x, for any "x" not listed above

Use Cases:

Format dates for aggregations:

SELECT DATE_FORMAT(creationtime, '%b') AS Month, COUNT(*) AS Number_Of_Orders
FROM orders
GROUP BY Month;

Month	Number_Of_Orders
Jan	4
Feb	4
Mar	2

Can end up getting different formats from different sources, so need to format incoming data before putting it into the main database.

CONVERT
Converts a value to a different data type.
Style
6 -> 01 Jan 25
112 -> 20250820

Note: CONVERT(data_type, value) in SQL Server,
CONVERT(value, data_type) in MySQL

Example:

SELECT CONVERT(creationtime, CHAR) FROM orders LIMIT 3;

CONVERT(creationtime, CHAR)
2025-01-01 12:34:56
2025-01-05 23:22:04

2025-01-10 18:24:08

Convert to utf8:

SELECT CONVERT(creationtime USING utf8mb4) FROM orders LIMIT 3;

CONVERT(creationtime USING utf8mb4)
2025-01-01 12:34:56
2025-01-05 23:22:04
2025-01-10 18:24:08

Convert string to INT to String:

SELECT CONVERT(555, CHAR);

CONVERT(555, CHAR)
555

!!! **NOTE** !!!

If MySQL puts out a warning on a query, immediately use 'show warnings;' to see the warning message.

CAST
Changes data type from one to another,
example String to Number, Date to String, String to Date

The MySQL CAST() function is used to explicitly convert a value from one data type to another. It is a standard SQL function that provides a way to enforce data type conversions within your queries.

Example:

SELECT CAST(creationtime AS CHAR) FROM orders LIMIT 3;

CAST(creationtime AS CHAR)
2025-01-01 12:34:56
2025-01-05 23:22:04
2025-01-10 18:24:08

MySQL supports casting to a variety of data types, including:

BINARY
CHAR
DATE
DATETIME
TIME
DECIMAL
SIGNED (for signed integers)
UNSIGNED (for unsigned integers)

Example after using CAST on made up date:

SELECT DATEDIFF(orderdate, CAST('1999-01-05' AS DATE)) AS Days_Since_First_Order
FROM orders LIMIT 3;

Days_Since_First_Order
9493
9497
9502

Still works without cast...:

SELECT DATEDIFF(orderdate, '1999-01-05') AS Days_Since_First_Order
FROM orders LIMIT 3;

Days_Since_First_Order
9493
9497
9502

Comparison:	CASTING	FORMATTING
CAST	Any type to any type	X No Formatting
CONVERT	Any type to any type	Formats only Date & Time
FORMAT	Any type to only String	Formats Date & Time, Numbers

Calculations

DATEADD (DATE_ADD in MySQL)

Example:

SELECT DATE_ADD('2005-01-01', INTERVAL 5 DAY);

DATE_ADD('2005-01-01', INTERVAL 5 DAY)
2005-01-06

SELECT DATE_ADD('2005-01-01', INTERVAL 9 MONTH);

DATE_ADD('2005-01-01', INTERVAL 9 MONTH)
2005-10-01

SELECT DATE_ADD('2005-01-01', INTERVAL 20 YEAR);

DATE_ADD('2005-01-01', INTERVAL 20 YEAR)
2025-01-01

With time, INTERVAL 3 HOUR, etc.

Use - to subtract values:

SELECT DATE_ADD('2005-01-05', INTERVAL -4 DAY);

DATE_ADD('2005-01-05', INTERVAL -4 DAY)
2005-01-01

With all 3:

```
SELECT DATE_ADD(DATE_ADD(DATE_ADD('2005-01-01', INTERVAL 19 YEAR),
INTERVAL 9 MONTH),
INTERVAL 24 DAY) AS NewDate;
```

NewDate
2024-10-25

DATEDIFF

Example:

```
SELECT DATEDIFF('1999-01-10', '1999-01-05');
```

DATEDIFF('1999-01-10', '1999-01-05')
5

Returns number of days between, negative if the 2nd argument is later than the 1st.
Ignores time components of DATE value.

Better Example:

```
SELECT DATEDIFF(shipdate, orderdate) AS ShipDays, orderdate, shipdate
FROM orders
LIMIT 3;
```

ShipDays	orderdate	shipdate
4	2025-01-01	2025-01-05
5	2025-01-05	2025-01-10
15	2025-01-10	2025-01-25

Note:
Use NOW() with DATEDIFF to find number of days since some other Date.

“For a more precise calculation of the difference in years, MySQL's TIMESTAMPDIFF() function is recommended. This function directly calculates the difference between two datetime expressions in a specified unit, including years.”

Example Code

SELECT TIMESTAMPDIFF(YEAR, birth_date, CURDATE()) AS age_in_years;

Example:

SELECT firstname, lastname, TIMESTAMPDIFF(YEAR, birthdate, NOW()) AS Years_Old FROM employees ORDER BY TIMESTAMPDIFF(YEAR, birthdate, NOW())

firstname	lastname	Years_Old
Kevin	Brown	52
Michael	Ray	48
Carol	Baker	43
Mary	NULL	39
Frank	Lee	36

Validation

ISDATE
Checks if a value is a valid date

!!!!!!!!!!!!!!!!!!!!!!! **MySQL** !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
MySQL does not have a direct equivalent to SQL Server's ISDATE() function, which checks if an expression is a valid date. However, you can achieve similar functionality in MySQL using the STR_TO_DATE() function or by attempting a CAST() to a date type.

1. Using STR_TO_DATE():

The STR_TO_DATE() function attempts to parse a string into a date based on a specified format. If the string does not conform to the format or represents an invalid date (e.g., February 30th), STR_TO_DATE() will return NULL. You can then check for this NULL value.

SELECT STR_TO_DATE('2025-02-30', '%Y-%m-%d') IS NOT NULL AS IsValidDate;
-- This will return 0 (false) because '2025-02-30' is an invalid date.

Example:

```sql
SELECT STR_TO_DATE('2025-02-30', '%Y-%m-%d') IS NOT NULL AS IsValidDate;
```

IsValidDate
0

```sql
SELECT STR_TO_DATE('2025-09-04', '%Y-%m-%d') IS NOT NULL AS IsValidDate;
-- This will return 1 (true) because '2025-09-04' is a valid date.
```

Example:

```sql
SELECT STR_TO_DATE('2025-09-04', '%Y-%m-%d') IS NOT NULL AS IsValidDate;
```

IsValidDate
1

Can CAST only if Valid Date from ISDATE:

Error:

```sql
SELECT
	CAST(OrderDate AS DATE) AS OrderDate,
FROM (
	SELECT '2025-08-20' AS OrderDate UNION
	SELECT '2025-08-21' UNION
	SELECT '2025-08-23' UNION
	SELECT '2025-08'
)

SELECT
	OrderDate,
	ISDATE(OrderDate),
	CASE WHEN ISDATE(OrderDate) = 1 THEN CAST(OrderDate AS DATE)
	END NewOrderDate
FROM (
	SELECT '2025-08-20' AS OrderDate UNION
	SELECT '2025-08-21' UNION
	SELECT '2025-08-23' UNION
	SELECT '2025-08'
)
```

NULL Functions

If a field has optional fields, can automatically put NULL into one that is unused.

Such as:

First Name [ ]
Middle Name (optional) [ ]
Last Name [ ]

Replace values:

ISNULL

Seems to be a boolean function in MySQL:

SELECT ISNULL(NULL);

ISNULL(NULL)
1

SELECT ISNULL('not null');

ISNULL('not null')
0

Instead:

SELECT CASE WHEN ISNULL(NULL)
THEN 'null found'
ELSE 'null not found'
END
AS 'NULL';

NULL
null found

SELECT CASE WHEN ISNULL('derp')
THEN 'null found'
ELSE 'null not found'
END
AS 'NULL';

NULL
null not found

That works, but MySQL has IFNULL instead:

SELECT IFNULL(NULL, 'test');

IFNULL(NULL, 'test')
test

SELECT IFNULL('derp', 'test');

IFNULL('derp', 'test')
derp

Use to find if column NULL, if so, can give different default value instead.

COALESCE
Returns the first non-null value from a list

Example:

SELECT COALESCE(NULL, NULL, 'derp', NULL, 'testing') AS SomeValue;

SomeValue
derp

Can put all columns, put a default value at very end to use if everything NULL.

ISNULL	COALESCE
Limited to 2 values	Unlimited
Faster	Slow

SQL Server -> ISNULL, Oracle -> NVL, MySQL -> IFNULL	Available in al DBMSs

Use Case:

***Handle the NULL before doing data aggregations.**

All AGGR functions ignore null, unless COUNT(*).

Example:

SELECT Score FROM customers;

Score
350
900
750
500
NULL

SELECT COUNT(score) AS ScoreCount FROM customers;

ScoreCount
4

SELECT COUNT(IFNULL(score, 1)) AS ScoreCount FROM customers;

ScoreCount
5

***Handle the NULL before doing mathematical operations:**

SELECT score, score + 1 AS UpdatedScore FROM customers;

score	UpdatedScore
350	351
900	901
750	751
500	501
NULL	NULL

SELECT score, IFNULL(score, 0) + 1 AS UpdatedScore FROM customers;

score	UpdatedScore
350	351
900	901
750	751
500	501
NULL	1

SELECT score, score * 2 AS UpdatedScore FROM customers;

score	UpdatedScore
350	700
900	1800
750	1500
500	1000

NULL	NULL

SELECT score, IFNULL(score, 1) * 1 AS UpdatedScore FROM customers;

score	UpdatedScore
350	350
900	900
750	750
500	500
NULL	1

Last name NULL:

SELECT CONCAT(firstname, ' ', lastname) AS FullName FROM customers;

FullName
Jossef Goldberg
Kevin Brown
NULL
Mark Schwarz
Anna Adams

SELECT CONCAT(firstname, ' ', COALESCE(lastname, ")) AS FullName FROM customers;

FullName
Jossef Goldberg
Kevin Brown
Mary
Mark Schwarz

Anna Adams

***Handle NULL before JOINing tables:**

Need 2 keys to JOIN 2 tables, but some of the second keys are NULL.
Can't compare NULL using = operator used in JOIN condition ON.

a.type = b.type

Instead:

IFNULL(a.type, '') = IFNULL(b.type, '')

***Handle NULL before sorting the data:**

NULL becomes first if sorting lowest to highest:

SELECT firstname, lastname FROM customers ORDER BY lastname;

firstname	lastname
Mary	NULL
Anna	Adams
Kevin	Brown
Jossef	Goldberg
Mark	Schwarz

Method #1, replace with very low ranking last name:

SELECT firstname, lastname FROM customers
ORDER BY IFNULL(lastname, 'ZZZZZZZZZZZZZZZ');

firstname	lastname
Anna	Adams
Kevin	Brown

Jossef	Goldberg
Mark	Schwarz
Mary	NULL

Not best method, use this instead:

```
SELECT firstname, lastname,
CASE WHEN ISNULL(lastname)
        THEN 1
        ELSE 0
END AS Flag
FROM customers
ORDER BY Flag, lastname;
```

firstname	lastname	Flag
Anna	Adams	0
Kevin	Brown	0
Jossef	Goldberg	0
Mark	Schwarz	0
Mary	NULL	1

NULLIF
Returns NULL if Column1/Value1 matches Column2/Value2
Else returns Column1/Value1

Example:

```
SELECT NULLIF('derp', 'derp');
```

NULLIF('derp', 'derp')
NULL

```
SELECT NULLIF('notderp', 'derp');
```

NULLIF('notderp', 'derp')

notderp

Use Case:

Compare Original_Price to Discount_Price,
if NULLIF returns NULL on this comparison, clearly there is an error somewhere.
(The 2 shouldn't match)

Prevent error of dividing by zero:

SELECT 5 / 0;

5 / 0
NULL

1 row in set, **1 warning** (0.00 sec)

MySQL gives warning:

show warnings;

Level	Code	Message
Warning	1365	Division by 0

Use this instead:

SELECT 5 / NULLIF(0, 0);

5 / NULLIF(0, 0)
NULL

Check for NULLs:

IS NULL

Return TRUE if value is NULL
Else return FALSE

IS NOT NULL

Return TRUE if value is not NULL
Else return FALSE

Example:

```
SELECT
	'IS NULL' AS FunctionName,
	NULL IS NULL AS NullValue,
	'derp' IS NULL AS DerpValue
UNION ALL
SELECT
	'IS NOT NULL' AS FunctionName,
	NULL IS NOT NULL AS NullValue,
	'derp' IS NOT NULL AS DerpValue;
```

FunctionName	NullValue	DerpValue
IS NULL	1	0
IS NOT NULL	0	1

Use Cases:

Searching for missing information, then can be followed up by filling in or fixing in some way.

```
SELECT * FROM customers WHERE score IS NULL;
```

customerid	firstname	lastname	country	score
5	Anna	Adams	USA	NULL

Find all customers that don't match names with employees:

```
SELECT * FROM customers c
LEFT JOIN employees e
ON
	c.firstname = e.firstname
	AND
```

c.lastname = e.lastname
WHERE e.firstname IS NULL AND e.lastname IS NULL;

customerid	firstname	lastname	country	score	employeeid	firstname	lastname	department	birthdate	gender	salary	managerid
1	Jossef	Goldberg	Germany	350	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
3	Mary	NULL	USA	750	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
4	Mark	Schwarz	Germany	500	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
5	Anna	Adams	USA	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Mary NULL should not be in there, does exist in the employees table, here:

SELECT * FROM customers c
LEFT JOIN employees e
ON
c.firstname = e.firstname
AND
IFNULL(c.lastname, '') = IFNULL(e.lastname, '')
WHERE e.firstname IS NULL

customerid	firstname	lastname	country	score	employeeid	firstname	lastname	department	birthdate	gender	salary	managerid
1	Jossef	Goldberg	Germany	350	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
4	Mark	Schwarz	Germany	500	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
5	Anna	Adams	USA	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Fixed!

NULL vs. Empty String vs. Blank Space

NULL is nothing, don't know about it.

Empty string is just a string with zero characters.

Blank space has one or more space characters.
>= 1 looks the same at a glance!

Can use LENGTH to find number of whitespace chars in string.

	Blank Space	NULL	Empty String
Representation	' '	NULL	''
Meaning	Known, Space value	Unknown	Known, Empty value
Data Type	String(1+)	Special Marker	String(0)

Storage	Each Occupies Memory	Very Minimal	Occupies Memory
Performance	Slow	Best	Fast
Comparison	= ' '	IS NULL	= ''

Data Policies:

Data input can often involve data with very different setups or corruption,
need to know how to work with all cases, how to filter, how to fix.
All that given the data policy for whatever organization you are working with.

Should always avoid spaces and just use empty strings or NULLs instead.

#1 Data Policy

Use NULLS and Empty strings.

#2 Data Policy

Only use NULLS and avoid using empty strings and blank spaces.

Policy #2 easier to understand and deal with.

#3 Data Policy

Use the default value 'unknown' and avoid using nulls, empty string, and blank spaces:

All 3 as an example:

```
WITH orders AS (
	SELECT 1 Id, 'A' Category
	UNION
	SELECT 2, NULL
	UNION
	SELECT 3, ''
	UNION
	SELECT 4, '     '
)
SELECT
	*,
	LENGTH(Category) CategoryLen,
	LENGTH(TRIM(Category)) Policy1,
```

```
	NULLIF(TRIM(Category), '') Policy2,
	COALESCE(NULLIF(TRIM(Category), '') , 'unknown') Policy3
FROM orders;
```

Id	Category	CategoryLen	Policy1	Policy2	Policy3
1	A	1	1	A	A
2	NULL	NULL	NULL	NULL	unknown
3		0	0	NULL	unknown
4		5	0	NULL	unknown

When working with data on own, use Policy #2,
But use Policy #3 for presentation.

Summary:

NULLs are special markers in SQL that mean 'missing value'
Using NULLs can optimize storage and performance.

NULL functions can replace values or check values if they are NULL.

Need to handle NULLs before data aggregation, math ops, joining tables, sorting data

CASE Statement

Evaluates a list of conditions and returns a value when the first condition is met.

Syntax:

```
CASE
	WHEN condition1 THEN result1
	WHEN condition2 THEN result2
	...
	ELSE default_result
END
```

Most important condition should go first, order matters.
ELSE is optional.

Example:

```
SELECT sales,
CASE
    WHEN sales > 50 THEN 'High'
    WHEN sales > 30 THEN 'Mid'
    WHEN sales > 10 THEN 'Low'
    ELSE 'Very Low'
END AS sales_score
FROM orders;
```

sales	sales_score
10	Very Low
15	Low
20	Low
60	High
25	Low
50	Mid
30	Low
90	High
20	Low
60	High

Use Cases:

Main purpose is Data Transformation

Derive new information

- Create new Columns based on existing data -

***Categorizing Data: Group the data into different categories based on certain conditions.**

```
WITH sales_score AS ( SELECT CASE
        WHEN sales > 50 THEN 'High'
        WHEN sales > 30 THEN 'Mid'
        WHEN sales > 10 THEN 'Low'
        ELSE 'Very Low'
END AS sales_score
FROM orders
)
SELECT sales_score, COUNT(sales_score) AS score_count
FROM sales_score
GROUP BY sales_score;
```

sales_score	score_count
Very Low	1
Low	5
High	3
Mid	1

```
WITH sales_score AS ( SELECT sales, CASE
        WHEN sales > 50 THEN 'High'
        WHEN sales > 30 THEN 'Mid'
        WHEN sales > 10 THEN 'Low'
        ELSE 'Very Low'
END AS sales_score
FROM orders
)
SELECT sales_score, COUNT(sales_score) AS score_count, SUM(sales) AS total_sales
FROM sales_score
GROUP BY sales_score;
```

sales_score	score_count	total_sales

Very Low	1	10
Low	5	110
High	3	210
Mid	1	50

***Mapping: Transform the values from one form to another.**

```
SELECT firstname, lastname, gender,
CASE
	WHEN gender = 'M' THEN 'Male'
	WHEN gender = 'F' THEN 'Female'
	ELSE 'Not Available'
END AS Full_Gender
FROM employees;
```

firstname	lastname	gender	Full_Gender
Frank	Lee	M	Male
Kevin	Brown	M	Male
Mary	NULL	F	Female
Michael	Ray	M	Male
Carol	Baker	F	Female

Often writing same thing, such as 'Country = ____'

Can use this instead:

```
CASE Country
	WHEN 'Germany' THEN 'DE'
	WHEN 'India' THEN 'IN'
	...
	ELSE 'n/a'
END
```

Example:

```
SELECT firstname, lastname, gender,
CASE gender
	WHEN 'M' THEN 'Male'
	WHEN 'F' THEN 'Female'
	ELSE 'Not Available'
END AS Full_Gender
FROM employees;
```

firstname	lastname	gender	Full_Gender
Frank	Lee	M	Male
Kevin	Brown	M	Male
Mary	NULL	F	Female
Michael	Ray	M	Male
Carol	Baker	F	Female

'Quick Form'.
Cannot use if logic gets complicated. Might want to use Full Form at all times, to avoid having to rewrite everything in the quick form if adding more logic.

***Handling NULLS**

Replace NULLs with a specific value.

(covered in previous section)

***Conditional aggregation**

Apply aggregate functions only on subsets of data that fulfill certain conditions.

```
WITH sales_score AS ( SELECT customerid, CASE
        WHEN sales > 30 THEN 1
        ELSE 0
END AS sales_score
FROM orders
)
SELECT sales_score.customerid, SUM(sales_score) AS sales_higher_than_30
FROM sales_score
GROUP BY sales_score.customerid;
```

customerid	sales_higher_than_30
1	1
2	0
3	2
4	1

Rules:

The data type of the results must be matching!
(THEN/ELSE output must have same data type)

CASE statement can be used anywhere in the query.

When mapping CASE WHEN, need to know all possible values to write statements,
don't forget to add default ELSE value.

Aggregate Functions

Accept multiple rows as input, output single value.

Example:

SELECT COUNT(*), SUM(sales), AVG(sales), MAX(sales), MIN(sales) FROM orders;

COUNT(*)	SUM(sales)	AVG(sales)	MAX(sales)	MIN(sales)
10	380	38.0000	90	10

GROUP BY breaks the list into parts based on column(s).

Example:

SELECT COUNT(*), SUM(score), AVG(score), MAX(score), MIN(score) FROM customers GROUP BY country;

COUNT(*)	SUM(score)	AVG(score)	MAX(score)	MIN(score)
2	850	425.0000	500	350
3	1650	825.0000	900	750

Window Functions

AKA Analytical Functions

Basics

Window Functions: Perform calculations (e.g. aggregation) on a specific subset of data, without losing the level of details of rows.

Columns 'product' has 'Caps' and 'Gloves'. 4 rows, GROUP BY product yields 2 rows with total sales for 'Caps' and 'Gloves'.

Window function gets total sales, but keeps same number of rows showing all the info.

Doesn't lose level of details, 'Row-Level Calculation'.

WINDOW functions are for more advanced data tasks.

WINDOW Functions:

Example:

```
SELECT orderid, orderdate, productid, sales,
SUM(sales) OVER(PARTITION BY productid) AS TotalSalesByProducts
FROM orders;
```

orderid	orderdate	productid	sales	TotalSalesByProducts
1	2025-01-01	101	10	140
3	2025-01-10	101	20	140
8	2025-02-18	101	90	140
9	2025-03-10	101	20	140
2	2025-01-05	102	15	105
7	2025-02-15	102	30	105
10	2025-03-15	102	60	105
5	2025-02-01	104	25	75
6	2025-02-05	104	50	75
4	2025-01-20	105	60	60

Syntax:

WINDOW_FUNCTION() OVER(Partition_Clause Order_Clause Frame_Clause)

Example:

AVG(Column1) OVER (PARTITION BY Column2 ORDER BY Column3 ROWS UNBOUNDED PRECEDING)

Perform calculations within a window.

The Window_Function is the calculation.

Column1 can be empty, column, can be number (in certain functions), some with multiple arguments, or whole conditional logic.

OVER specifies that it is a Window Function.

PARTITION BY Divides the result set into partitions (Windows).

Empty? Calculation is done on entire Dataset.
Column? Divides into Windows.
PARTITION BY always optional.

Examples:

```
SELECT MONTH(orderdate) AS Month, sales, orderstatus,
	SUM(sales) OVER(PARTITION BY MONTH(orderdate)) AS SalesPerMonth,
	SUM(sales) OVER () AS TotalSales,
	SUM(sales) OVER (PARTITION BY MONTH(orderdate), orderstatus) AS
SalesPerMonthAndStatus
FROM orders;
```

Month	sales	orderstatus	SalesPerMonth	TotalSales	SalesPerMonthAndStatus
1	10	Delivered	105	380	30
1	20	Delivered	105	380	30
1	15	Shipped	105	380	75
1	60	Shipped	105	380	75
2	25	Delivered	195	380	105
2	50	Delivered	195	380	105
2	30	Delivered	195	380	105
2	90	Shipped	195	380	90
3	20	Shipped	80	380	80
3	60	Shipped	80	380	80

Window Functions: Allows aggregation of data at different granularity within the same query.

ORDER BY sorts data within a Window. Optional in Aggregate, but required in Rank and Value functions.

```
SELECT sales,
       SUM(sales) OVER(PARTITION BY MONTH(orderdate) ORDER BY sales DESC) AS
SalesPerMonth
FROM orders;
```

sales	SalesPerMonth
60	60
20	80
15	95
10	105
90	90
50	140
30	170
25	195
60	60
20	80

RANK:

```
SELECT orderid, orderdate, sales,
       RANK() OVER (ORDER BY sales DESC) AS RankSales
FROM orders;
```

orderid	orderdate	sales	RankSales
8	2025-02-18	90	1

4	2025-01-20	60	2
10	2025-03-15	60	2
6	2025-02-05	50	4
7	2025-02-15	30	5
5	2025-02-01	25	6
3	2025-01-10	20	7
9	2025-03-10	20	7
2	2025-01-05	15	9
1	2025-01-01	10	10

Window Frame (Frame Clause):

Defines a subset of rows within each window that is relevant for the calculation.

```
AVG(sales) OVER (
    PARTITION BY Category
    ORDER BY orderdate
    ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING
)
```

Must have ORDER BY to use Frame Clause.

ROWS: Frame Types: **ROWS, RANGE**

CURRENT ROW: Frame Boundary (Lower Value): CURRENT ROW, N PRECEDING, UNBOUNDED PRECEDING

UNBOUNDED FOLLOWING: Frame Boundary (Higher Value): CURRENT ROW, N FOLLOWING, UNBOUNDED FOLLOWING

```
SELECT sales,
SUM(sales) OVER (ORDER BY MONTH(orderdate)) AS SalesTotal
FROM orders;
```

sales	SalesTotal
10	105
15	105
20	105
60	105
25	300
50	300

30	300
90	300
20	380
60	380

```
SELECT sales,
SUM(sales) OVER (
        ORDER BY MONTH(orderdate)
        ROWS BETWEEN CURRENT ROW AND 2 FOLLOWING
) AS SalesTotal
FROM orders;
```

sales	SalesTotal	(Explanation)
10	45	<- Sum of current row and following 2 rows: 10 + 15 + 20 = 45
15	95	<- Sum of current row and following 2 rows: 15 + 20 + 60 = 95
20	105	<- etc.
60	135	
25	105	
50	170	
30	140	
90	170	
20	80	<- Sum of current row and following 2 rows: 20 + 60 + 0(None) = 80
60	60	<- Sum of current row and following 2 rows: 60 + 0 + 0 = 60

Slides down the rows calculating as it goes.

UNBOUNDED FOLLOWING is all the rest (Less and less rows as it slides down)

1 PRECEDING AND CURRENT ROW:

```
SELECT sales,
SUM(sales) OVER (
        ORDER BY MONTH(orderdate)
        ROWS BETWEEN 1 PRECEDING AND CURRENT ROW
) AS SalesTotal
FROM orders;
```

sales	SalesTotal	(Explanation)
10	10	<- 0 + 10 = 10
15	25	<- 10 + 15 = 25
20	35	<- etc.
60	80	
25	85	
50	75	
30	80	
90	120	
20	110	
60	80	<- 20 + 60 = 80

UNBOUNDED PRECEDING AND CURRENT ROW

Current row and all before it, so more rows as it moves on.

1 PRECEDING AND 1 FOLLOWING

1 row before and after current row.

UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING

All rows regardless of current row.

Compact Frame:

For only PRECEDING, the CURRENT ROW can be skipped:

Normal Form: ROWS BETWEEN CURRENT ROW AND 2 PRECEDING
Short Form: ROWS 2 PRECEDING

Without specifying a Frame using ORDER BY, SQL uses default Frame:
ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

No Frame if not using ORDER BY.

4x Rules (Limitations):

Rule #1. Window functions can be used only in SELECT and ORDER BY clauses.

Example:

```
SELECT sales,
SUM(sales) OVER (ORDER BY MONTH(orderdate)) AS SalesTotal
FROM orders
ORDER BY SUM(sales) OVER (ORDER BY MONTH(orderdate)) DESC;
```

sales	SalesTotal
20	380
60	380
25	300
50	300
30	300
90	300
10	105
15	105
20	105
60	105

If needed to use same exact Window in both, just use the alias:

```
SELECT sales,
SUM(sales) OVER (ORDER BY MONTH(orderdate)) AS SalesTotal
FROM orders
ORDER BY SalesTotal DESC;
```

sales	SalesTotal
20	380
60	380
25	300
50	300
30	300
90	300
10	105
15	105
20	105
60	105

Window functions cannot be used to filter data.
(No WHERE Clause or GROUP BY)

Rule #2. Window functions cannot be nested.

Rule #3. SQL executes WINDOW function after the WHERE Clause.
(Filters first, then executes Window function)

Rule #4. Window Function can be used together with GROUP BY in the same query, ONLY if the same columns are used.

Example:

```
SELECT customerid,
SUM(sales) AS TotalSales, RANK() OVER (
        ORDER BY SUM(sales) DESC
) AS RankCustomers
FROM orders
GROUP BY customerid;
```

customerid	TotalSales	RankCustomers
3	125	1
1	110	2
4	90	3
2	55	4

First build by GROUP BY, then define and build Window Function.

Summary:

Performs calculations on a subset of data without losing details.

Window functions are more powerful and dynamic in comparison to the GROUP BY.

Data analysis can be more advanced.

Can use both at once if same columns are used.

Rule #1. Window functions can be used only in SELECT and ORDER BY clauses.

Rule #2. Window functions cannot be nested.

Rule #3. SQL executes WINDOW function after the WHERE Clause.
(Filters first, then executes Window function)

Rule #4. Window Function can be used together with GROUP BY in the same query, ONLY if the same columns are used.

Aggregate:

COUNT for any data type, Numeric only with SUM, AVG, MIN, and MAX.

For all Aggregate: Partition, Order, and Frame clauses are optional.

COUNT

Example:

SELECT MONTH(orderdate), sales, COUNT(*) OVER () AS total_orders FROM orders;

MONTH(orderdate)	sales	total_orders
1	10	10
1	15	10
1	20	10
1	60	10
2	25	10
2	50	10
2	30	10
2	90	10
3	20	10
3	60	10

```
SELECT
	MONTH(orderdate),
	sales,
	customerid,
	COUNT(*) OVER () AS total_orders,
	COUNT(*) OVER (PARTITION BY customerid) AS OrdersByCustomers
FROM orders;
```

MONTH(orderdate)	sales	customerid	total_orders	OrdersByCustomers
1	20	1	10	3

1	60	1	10	3
2	30	1	10	3
1	10	2	10	3
2	25	2	10	3
3	20	2	10	3
1	15	3	10	3
2	50	3	10	3
3	60	3	10	3
2	90	4	10	1

```
SELECT
	*,
	COUNT(*) OVER () AS TotalCustomers,
	COUNT(score) OVER () AS TotalScores
FROM customers;
```

customerid	firstname	lastname	country	score	TotalCustomers	TotalScores
1	Jossef	Goldberg	Germany	350	5	4
2	Kevin	Brown	USA	900	5	4
3	Mary	NULL	USA	750	5	4
4	Mark	Schwarz	Germany	500	5	4
5	Anna	Adams	USA	NULL	5	4

***Duplicates can lead to inaccuracies in data:**

Duplicate primary keys can be a problem.

To check:

```
SELECT
	orderid,
	COUNT(*) OVER (PARTITION BY orderid) AS CheckPrimaryKey
FROM orders;
```

orderid	CheckPrimaryKey
1	1
2	1

3	1
4	1
5	1
6	1
7	1
8	1
9	1
10	1

Here is where there is a problem:

```
SELECT
	orderid,
	COUNT(*) OVER (PARTITION BY orderid) AS CheckPrimaryKey
FROM orders_archive;
```

orderid	CheckPrimaryKey
1	1
2	1
3	1
4	2
4	2
5	1
6	3
6	3
6	3
7	1

Duplicates!

Find each duplicate exactly:

```
WITH CheckPrimaryKey AS (SELECT
	orderid,
	COUNT(*) OVER (PARTITION BY orderid) AS CheckPrimaryKey
FROM orders_archive)
SELECT * FROM CheckPrimaryKey
```

WHERE CheckPrimaryKey > 1;

orderid	CheckPrimaryKey
4	2
4	2
6	3
6	3
6	3

Use Cases:

#1. Overall analysis
#2. Category analysis
#3. Quality Checks: Identify NULLs
#4. Quality Checks: Identify Duplicates

SUM

Example:

```
SELECT
	MONTH(orderdate),
	sales,
	SUM(sales) OVER (PARTITION BY MONTH(orderdate)) AS total_sales
FROM orders;
```

MONTH(orderdate)	sales	total_sales
1	10	105
1	15	105
1	20	105
1	60	105
2	25	195
2	50	195
2	30	195
2	90	195
3	20	80
3	60	80

```
SELECT
	orderid,
	orderdate,
	sales,
	productid,
	SUM(sales) OVER (PARTITION BY productid) AS SalesByProduct
FROM orders;
```

orderid	orderdate	sales	productid	SalesByProduct
1	2025-01-01	10	101	140
3	2025-01-10	20	101	140
8	2025-02-18	90	101	140
9	2025-03-10	20	101	140
2	2025-01-05	15	102	105
7	2025-02-15	30	102	105
10	2025-03-15	60	102	105
5	2025-02-01	25	104	75
6	2025-02-05	50	104	75
4	2025-01-20	60	105	60

Comparison Use Cases

Compare the current value and aggregated value of window functions

Part-to-Whole analysis

Compare current sales to total sales

Example:

```
SELECT
	orderid,
	productid,
	sales,
	SUM(sales) OVER () AS TotalSales,
	CONCAT(
		CAST(
			CAST(sales AS float) / SUM(sales) OVER () * 100 AS DECIMAL(10, 3)
		), '%'
	) AS PercentageOfTotal
FROM orders;
```

orderid	productid	sales	TotalSales	PercentageOfTotal
1	101	10	380	2.632%
2	102	15	380	3.947%
3	101	20	380	5.263%
4	105	60	380	15.789%
5	104	25	380	6.579%
6	104	50	380	13.158%
7	102	30	380	7.895%
8	101	90	380	23.684%
9	101	20	380	5.263%
10	102	60	380	15.789%

Compare to Average analysis

Help to evaluate whether a value is > or < the Average

Example:

```
SELECT
    orderid,
    productid,
    sales,
    AVG(sales) OVER () AS AverageSales
FROM orders;
```

orderid	productid	sales	AverageSales
1	101	10	38.0000
2	102	15	38.0000
3	101	20	38.0000
4	105	60	38.0000
5	104	25	38.0000
6	104	50	38.0000
7	102	30	38.0000
8	101	90	38.0000
9	101	20	38.0000
10	102	60	38.0000

AVG

Example:

```
SELECT
	orderid,
	productid,
	sales,
	AVG(sales) OVER (PARTITION BY productid) AS AverageSales
FROM orders;
```

orderid	productid	sales	AverageSales
1	101	10	35.0000
3	101	20	35.0000
8	101	90	35.0000
9	101	20	35.0000
2	102	15	35.0000
7	102	30	35.0000
10	102	60	35.0000
5	104	25	37.5000
6	104	50	37.5000
4	105	60	60.0000

```sql
SELECT
	orderid,
	orderdate,
	productid,
	sales,
	AVG(sales) OVER () AS AverageSales,
	AVG(sales) OVER (PARTITION BY productid) AS AveragePerProduct
FROM orders;
```

orderid	orderdate	productid	sales	AverageSales	AveragePerProduct
1	2025-01-01	101	10	38.0000	35.0000
3	2025-01-10	101	20	38.0000	35.0000
8	2025-02-18	101	90	38.0000	35.0000
9	2025-03-10	101	20	38.0000	35.0000
2	2025-01-05	102	15	38.0000	35.0000
7	2025-02-15	102	30	38.0000	35.0000
10	2025-03-15	102	60	38.0000	35.0000
5	2025-02-01	104	25	38.0000	37.5000
6	2025-02-05	104	50	38.0000	37.5000
4	2025-01-20	105	60	38.0000	60.0000

```sql
SELECT
	orderid,
	orderdate,
	productid,
	sales,
	ROUND(AVG(sales) OVER ()) AS AverageSales,
	ROUND(AVG(sales) OVER (PARTITION BY productid)) AS AveragePerProduct
FROM orders;
```

orderid	orderdate	productid	sales	AverageSales	AveragePerProduct

1	2025-01-01	101	10	38	35
3	2025-01-10	101	20	38	35
8	2025-02-18	101	90	38	35
9	2025-03-10	101	20	38	35
2	2025-01-05	102	15	38	35
7	2025-02-15	102	30	38	35
10	2025-03-15	102	60	38	35
5	2025-02-01	104	25	38	38
6	2025-02-05	104	50	38	38
4	2025-01-20	105	60	38	60

Average Score without NULL:

```
SELECT
	customerid,
	lastname,
	score,
	COALESCE(score, 0) AS CustomerScore,
	ROUND(AVG(score) OVER ()) AS AverageScore,
	ROUND(AVG(COALESCE(score, 0)) OVER ()) AS AverageScoreNoNULL
FROM customers;
```

customerid	lastname	score	CustomerScore	AverageScore	AverageScoreNoNULL
1	Goldberg	350	350	625	500
2	Brown	900	900	625	500
3	NULL	750	750	625	500
4	Schwarz	500	500	625	500
5	Adams	NULL	0	625	500

Find all orders where sales are higher than the average sales across all orders:

```
WITH AverageSales AS (
	SELECT
		orderid,
		productid,
		sales,
		AVG(COALESCE(sales, 0)) OVER () AS AverageSales
	FROM orders
) SELECT *
FROM AverageSales
```

WHERE sales > AverageSales;

orderid	productid	sales	AverageSales
4	105	60	38.0000
6	104	50	38.0000
8	101	90	38.0000
10	102	60	38.0000

MIN & MAX

Compare to Extremes analysis

Compare current sales to the highest or lowest sales

Example:

```
SELECT
	orderid,
	productid,
	sales,
	MIN(sales) OVER () AS LowestSales,
	MAX(sales) OVER () AS HighestSales
FROM orders;
```

orderid	productid	sales	LowestSales	HighestSales
1	101	10	10	90
2	102	15	10	90
3	101	20	10	90
4	105	60	10	90
5	104	25	10	90
6	104	50	10	90
7	102	30	10	90
8	101	90	10	90
9	101	20	10	90
10	102	60	10	90

```
SELECT
	orderid,
	orderdate,
	productid,
	sales,
	MIN(COALESCE(sales, 0)) OVER () AS LowestSales,
	MAX(COALESCE(sales, 0)) OVER () AS HighestSales,
	MIN(COALESCE(sales, 0)) OVER (PARTITION BY productid) AS LowestSalesByProd,
	MAX(COALESCE(sales, 0)) OVER (PARTITION BY productid) AS HighestSalesByProd
FROM orders;
```

orderid	orderdate	productid	sales	LowestSales	HighestSales	LowestSalesByProd	HighestSalesByProd
1	2025-01-01	101	10	10	90	10	90
3	2025-01-10	101	20	10	90	10	90
8	2025-02-18	101	90	10	90	10	90
9	2025-03-10	101	20	10	90	10	90
2	2025-01-05	102	15	10	90	15	60
7	2025-02-15	102	30	10	90	15	60
10	2025-03-15	102	60	10	90	15	60
5	2025-02-01	104	25	10	90	25	50
6	2025-02-05	104	50	10	90	25	50
4	2025-01-20	105	60	10	90	60	60

Show the employees who have the highest salaries:

```
WITH HighestSalary AS (
	SELECT
		*,
		MAX(salary) OVER () AS HighestSalary
```

```
    FROM employees
) SELECT *
FROM HighestSalary
WHERE salary = HighestSalary;
```

employeeid	firstname	lastname	department	birthdate	gender	salary	managerid	HighestSalary
4	Michael	Ray	Sales	1977-02-10	M	90000	2	90000

Show deviation of each sales from the minimum and maximum sales amounts
(Lower is closest to value)

```
SELECT
    orderid,
    orderdate,
    productid,
    sales,
    sales - MIN(COALESCE(sales, 0)) OVER () AS DevFromLowest,
    MAX(COALESCE(sales, 0)) OVER () - sales AS DevFromHighest
FROM orders;
```

orderid	orderdate	productid	sales	DevFromLowest	DevFromHighest
1	2025-01-01	101	10	0	80
2	2025-01-05	102	15	5	75
3	2025-01-10	101	20	10	70
4	2025-01-20	105	60	50	30
5	2025-02-01	104	25	15	65
6	2025-02-05	104	50	40	40
7	2025-02-15	102	30	20	60
8	2025-02-18	101	90	80	0
9	2025-03-10	101	20	10	70
10	2025-03-15	102	60	50	30

Analytical Use Case: Running & Rolling Total

Running total:

Aggregate all values from the beginning up to the current point without dropping off older data.

Example:

```
SELECT
    sales,
    SUM(sales) OVER (
        ORDER BY MONTH(orderdate)
        ROWS BETWEEN
            UNBOUNDED PRECEDING
            AND
            CURRENT ROW
    ) AS SalesTotal
FROM orders;
```

sales	SalesTotal
10	10
15	25
20	45
60	105
25	130
50	180

30	210
90	300
20	320
60	380

Rolling total:

Aggregate all values within a fixed time window (Ex. 30 days)
As new data is added, the oldest data point will be dropped.

Moving Average:

```
SELECT
	orderid,
	productid,
	orderdate,
	sales,
	ROUND(AVG(sales) OVER (PARTITION BY productid)) AS AverageByProduct,
	ROUND(AVG(sales) OVER (PARTITION BY productid ORDER BY orderdate)) AS
MovingAvg
FROM orders;
```

orderid	productid	orderdate	sales	AverageByProduct	MovingAvg
1	101	2025-01-01	10	35	10
3	101	2025-01-10	20	35	15
8	101	2025-02-18	90	35	40
9	101	2025-03-10	20	35	35
2	102	2025-01-05	15	35	15
7	102	2025-02-15	30	35	23
10	102	2025-03-15	60	35	35
5	104	2025-02-01	25	38	25
6	104	2025-02-05	50	38	38
4	105	2025-01-20	60	60	60

Rolling Average:

```
SELECT
        orderid,
        productid,
        orderdate,
        sales,
        ROUND(AVG(sales) OVER (PARTITION BY productid)) AS AverageByProduct,
        ROUND(AVG(sales) OVER (PARTITION BY productid ORDER BY orderdate
        ROWS BETWEEN CURRENT ROW AND 1 FOLLOWING
        )) AS RollingAverage
FROM orders;
```

orderid	productid	orderdate	sales	AverageByProduct	RollingAverage
1	101	2025-01-01	10	35	15
3	101	2025-01-10	20	35	55
8	101	2025-02-18	90	35	55
9	101	2025-03-10	20	35	20
2	102	2025-01-05	15	35	23
7	102	2025-02-15	30	35	45
10	102	2025-03-15	60	35	60
5	104	2025-02-01	25	38	38
6	104	2025-02-05	50	38	50

4	105	2025-01-20	60	60	60

Overall Total:
SUM(sales) OVER ()

Total Per Groups:
SUM(sales) OVER (PARTITION BY productid)

Running Total:
SUM(sales) OVER (ORDER BY Month)

Rolling Total:
SUM(sales) OVER (
ORDER BY Month
ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
)

Use with any aggregate functions.

Window Functions

Rank & Value

Rank:

Integer-based Ranking:
Discrete Values, start from 1, goes up.
Position of value in a list.
Top/Bottom N Analysis

Has:
ROW_NUMBER, RANK, DENSE_RANK, and NTILE

Percentage-based Ranking:
Continuous Values, from 0 to 1, infinitely many values in between.
Distribution Analysis

Has:
CUME_DIST and PERCENT_RANK

RANK() OVER (PARTITTION BY Column ORDER BY Column)

^^^^^^ ^^^^^^^^^^^^^^ ^^^^^^^^^^^^
Must be empty Optional Required
^^^^^^
(Unless NTILE, number)

Frame clause not allowed.

ROW_NUMBER

Assign a unique number to each in a window.
Doesn't handle ties, if rows have same value, doesn't have same rank.
No gaps in ranking.

Example:

```
SELECT
    sales,
    ROW_NUMBER() OVER (ORDER BY sales) AS RowNumber
FROM orders;
```

sales	RowNumber
10	1
15	2
20	3
20	4
25	5
30	6
50	7
60	8
60	9
90	10

-- From highest to lowest:

```
SELECT
	sales,
	ROW_NUMBER() OVER (ORDER BY sales DESC) AS RowNumber
FROM orders;
```

sales	RowNumber
90	1
60	2
60	3
50	4
30	5
25	6
20	7
20	8
15	9
10	10

Use Cases:

--Find the top highest sales for each product (Focus on best product):

```
SELECT
	orderid,
	productid,
```

```
        sales,
        ROW_NUMBER() OVER (PARTITION BY productid ORDER BY sales DESC) AS
RankByProduct
FROM orders;
```

orderid	productid	sales	RankByProduct
8	101	90	1
3	101	20	2
9	101	20	3
1	101	10	4
10	102	60	1
7	102	30	2
2	102	15	3
6	104	50	1
5	104	25	2
4	105	60	1

Need top:

```
WITH RankByProduct AS (
        SELECT
                orderid,
                productid,
                sales,
                ROW_NUMBER() OVER (PARTITION BY productid ORDER BY sales DESC) AS
RankByProduct
        FROM orders
) SELECT *
        FROM RankByProduct
        WHERE RankByProduct = 1;
```

orderid	productid	sales	RankByProduct
8	101	90	1
10	102	60	1
6	104	50	1
4	105	60	1

-- Find the bottom-n performers:

```
WITH RankCustomers AS (
        SELECT
                customerid,
                SUM(sales) AS totalsales,
```

```
        ROW_NUMBER() OVER (ORDER BY SUM(sales)) AS RankCustomers
    FROM orders
    GROUP BY customerid
) SELECT *
    FROM RankCustomers
    WHERE RankCustomers <= 2;
```

customerid	totalsales	RankCustomers
2	55	1
4	90	2

--Generate unique ids

Assign unique IDs to the rows of the 'Orders Archive' table

```
SELECT
    orderid,
    orderdate,
    ROW_NUMBER() OVER (ORDER BY orderid, orderdate) AS UniqueID
FROM orders_archive;
```

orderid	orderdate	UniqueID
1	2024-04-01	1
2	2024-04-05	2
3	2024-04-10	3
4	2024-04-20	4
4	2024-04-20	5
5	2024-05-01	6
6	2024-05-05	7
6	2024-05-05	8
6	2024-05-05	9
7	2024-06-15	10

Paginating:

The process of breaking down a large data into smaller, more manageable chunks.

Improve importing and exporting data.

--Identify duplicates:

Identify duplicate rows in the table 'Orders Archive' (orders_archive) and return a clean result without any duplicates.

```
WITH rn AS (
	SELECT
		orderid,
		productid,
		customerid,
		shipaddress,
		billaddress,
		creationtime,
		ROW_NUMBER() OVER (PARTITION BY orderid ORDER BY creationtime DESC)
AS rn
	FROM orders_archive
) SELECT *
	FROM rn
	WHERE rn = 1;
```

orderid	productid	customerid	shipaddress	billaddress	creationtime
1	101	2	123 Main St	456 Billing St	2024-04-01 12:34:56
2	102	3	456 Elm St	789 Billing St	2024-04-05 23:22:04
3	101	1	789 Maple St	789 Maple St	2024-04-10 18:24:08
4	105	1	987 Victory Lane		2024-04-20 14:50:33
5	104	2	345 Oak St	678 Pine St	2024-05-01 14:02:41

6	101	3	543 Belmont Rd.	3768 Door Way	2024-05-12 20:36:55
7	102	3	111 Main St	222 Billing St	2024-06-16 23:25:15

-- To get the bad data:

```
WITH rn AS (
	SELECT
		orderid,
		productid,
		customerid,
		shipaddress,
		billaddress,
		creationtime,
		ROW_NUMBER() OVER (PARTITION BY orderid ORDER BY creationtime DESC)
AS rn
	FROM orders_archive
) SELECT *
	FROM rn
	WHERE rn > 1;
```

orderid	productid	customerid	shipaddress	billaddress	creationtime
4	105	1	987 Victory Lane		2024-04-20 05:50:33
6	104	3	543 Belmont Rd.	3768 Door Way	2024-05-07 13:22:05
6	104	3	543 Belmont Rd.	NULL	2024-05-06 15:34:57

RANK

Assign a rank to each row in a window, with gaps.
Handles ties, rows with same value have same rank.
If tied, both get same rank but following value doesn't get next rank in line, gets the next next rank instead.

Example:

```
SELECT
    sales,
    RANK() OVER (ORDER BY sales DESC) AS SalesRank
FROM orders;
```

sales	SalesRank	(Explanation)
90	1	
60	2	<- Tie
60	2	<- Tie
50	4	<- Following rank not 3 due to previous tie!
30	5	
25	6	
20	7	
20	7	
15	9	
10	10	

DENSE_RANK

Assign a rank to each row in a window, without gaps.
Handles ties, same values share rank.

Example:

```
SELECT
	sales,
	DENSE_RANK() OVER (ORDER BY sales DESC) AS SalesDenseRank
FROM orders;
```

sales	SalesDenseRank	(Explanation)
90	1	
60	2	<- Tie
60	2	<- Tie
50	3	<- Following tie, next value the very next rank, no skip
30	4	
25	5	
20	6	
20	6	
15	7	
10	8	

CUME_DIST

Calculates the cumulative distribution of a value within a set of values.

Position # / # of Rows

Example:

```
SELECT
    sales,
    CUME_DIST() OVER (ORDER BY sales) AS SalesCumeDist
FROM orders;
```

sales	SalesCumeDist
10	0.1
15	0.2
20	0.4
20	0.4
25	0.5
30	0.6
50	0.7
60	0.9
60	0.9
90	1

--Highest to lowest:

```
SELECT
    sales,
```

```
    CUME_DIST() OVER (ORDER BY sales DESC) AS SalesCumeDist
FROM orders;
```

sales	SalesCumeDist	(Explanation)
90	0.1	-> 1 / 10
60	0.3	-> 2 / 10 = 1 / 5
60	0.3	
50	0.4	
30	0.5	
25	0.6	
20	0.8	
20	0.8	
15	0.9	
10	1	

PERCENT_RANK

Returns the percentile ranking number of a row.

Percent_Rank = Position # - 1 / # of Rows - 1

Tie Rule: The position of the first occurence of the same value.

Always 0 -> 1.

Example:

```
SELECT
    sales,
    CAST(PERCENT_RANK() OVER (ORDER BY sales) AS DECIMAL(10, 3)) AS
SalesPercentRank
FROM orders;
```

sales	SalesPercentRank
10	0.000
15	0.111
20	0.222
20	0.222
25	0.444
30	0.556
50	0.667
60	0.778

60	0.778
90	1.000

--Highest to lowest:

```
SELECT
	sales,
	CAST(PERCENT_RANK() OVER (ORDER BY sales DESC) AS DECIMAL(10, 3)) AS
SalesPercentRank
FROM orders;
```

sales	SalesPercentRank	(Explanation)
90	0.000	<- 1 - 1 / 10 - 1 = 0
60	0.111	<- 2 - 1 / 10 - 1 = 1 / 9
60	0.111	
50	0.333	
30	0.444	
25	0.556	
20	0.667	
20	0.667	
15	0.889	
10	1.000	

Use Cases:

Find the products that fall within the highest 40% of the prices:

```
WITH DistRank AS (
	SELECT
```

```
            product,
            price,
            CUME_DIST() OVER (ORDER BY price DESC) AS DistRank
        FROM products
) SELECT *
        FROM DistRank
        WHERE DistRank <= 0.4;
```

product	price	DistRank
Gloves	30	0.2
Caps	25	0.4

NTILE

Divides the rows into a specified number of approx. equal groups (Buckets).
Bucket Size = # of rows / # of buckets

Example:

```
SELECT
        sales,
        NTILE(2) OVER (ORDER BY sales) AS SalesNTile
FROM orders;
```

sales	SalesNTile
10	1
15	1
20	1
20	1
25	1
30	2
50	2
60	2
60	2
90	2

(Bucket size = 10 / 2 = 5)

```
SELECT
	sales,
	NTILE(4) OVER (ORDER BY sales) AS SalesNTile
FROM orders;
```

sales	SalesNTile
10	1
15	1
20	1
20	2
25	2
30	2
50	3
60	3
60	4
90	4

(Bucket size = 10 / 4 = 2.5 (2-3))
(With fractional, larger groups come first, then smaller groups)

- Highest to lowest:

```
SELECT
	sales,
	NTILE(2) OVER (ORDER BY sales DESC) AS SalesNTile
FROM orders;
```

sales	SalesNTile

90	1
60	1
60	1
50	1
30	1
25	2
20	2
20	2
15	2
10	2

Use Cases:

Data Analyst: Data Segmentation

Divides a dataset into distinct subsets based on certain criteria.

--Segment all orders into 3 categories: high, medium, and low sales:

```
WITH Buckets AS (
	SELECT
		orderid,
		sales,
		NTILE(3) OVER (ORDER BY sales DESC) AS Buckets
	FROM orders
) SELECT
	orderid,
	sales,
	CASE
		WHEN Buckets = 1 THEN 'High'
		WHEN Buckets = 2 THEN 'Medium'
		ELSE 'Low'
	END AS Buckets
FROM Buckets;
```

orderid	sales	Buckets
8	90	High
4	60	High
10	60	High
6	50	High
7	30	Medium

5	25	Medium
3	20	Medium
9	20	Low
2	15	Low
1	10	Low

Data Engineer: Equalizing load processing

Split into buckets to make load processing more manageable, less stress on networks.

-- In order to export the data, divide the orders into 2 groups:
(Best to ORDER BY primary key)

```
SELECT
        NTILE(2) OVER (ORDER BY orderid) AS Buckets,
        orderid,
        productid,
        customerid,
        salespersonid,
        orderdate
FROM orders;
```

Buckets	orderid	productid	customerid	salespersonid	orderdate
1	1	101	2	3	2025-01-01
1	2	102	3	3	2025-01-05
1	3	101	1	5	2025-01-10
1	4	105	1	3	2025-01-20
1	5	104	2	5	2025-02-01
2	6	104	3	5	2025-02-05
2	7	102	1	1	2025-02-15
2	8	101	4	3	2025-02-18
2	9	101	2	3	2025-03-10
2	10	102	3	5	2025-03-15

Split by more buckets if needed, deal with each bucket separately when transferring or whatever is needed by the task.

Value (Analytics):

Can use to access a value from another row.
Ex.: Compare sales of current month to previous month (LAG).
Or with next month (LEAD), or with first (FIRST_VALUE), or with last (LAST_VALUE).

	Expressions	Partitions	Order	Frame
LEAD	All Data Types	Optional	Required	Not Allowed
LAG	"	"	"	"
FIRST_VALUE	Optional	"	"	"
LAST_VALUE	Should Be Used	"	"	"

LEAD

Returns value from a previous row.

LEAD(Expression[, offset][, Default_Value])

Expression is required (Any data type).
Offset is number of rows forward or backward from current row (default = 1).
Default_Value returns value if next/previous row is not available (Default = NULL).

LAG

Returns value from a subsequent row.

Examples:

```
SELECT
	MONTH(orderdate),
	sales,
	LEAD(sales) OVER (ORDER BY MONTH(orderdate)) AS SalesByNextMonth,
	LAG(sales) OVER (ORDER BY MONTH(orderdate)) AS SalesByPrevMonth
FROM orders;
```

MONTH(orderdate)	sales	SalesByNextMonth	SalesByPrevMonth
1	10	15	NULL
1	15	20	10
1	20	60	15
1	60	25	20
2	25	50	60
2	50	30	25
2	30	90	50
2	90	20	30
3	20	60	90
3	60	NULL	20

"Time Series Analysis": The process of analyzing the data to understand patterns, trends, and behaviors over time.

Year-over-Year (YoY): Analyze the overall growth or decline of the business's performance over time.

Month-over-Month (MoM): Analyze short-term trends and discover patterns in seasonality.

SQL Task: Analyze the month-over-month (MoM) performance by finding the percentage change in sales between the current and previous month:

```
WITH MonthSales AS (
    SELECT
        MONTH(orderdate) AS OrderMonth,
        SUM(sales) AS CurrentMonthSales,
        LAG(SUM(sales)) OVER (ORDER BY MONTH(orderdate)) AS PrevMonthSales
    FROM orders
    GROUP BY MONTH(orderdate)
) SELECT
    *,
    CONCAT(
        CAST(
            (CurrentMonthSales - PrevMonthSales) / PrevMonthSales * 100
            AS DECIMAL(10, 3)
        ), '%'
    ) AS MoM_Change
FROM MonthSales;
```

OrderMonth	CurrentMonthSales	PrevMonthSales	MoM_Change
1	105	NULL	NULL
2	195	105	85.714%
3	80	195	-58.974%

Customer Retention Analysis

Measure customer's behavior and loyalty to help businesses build strong relationships with customers.

SQL Task:

Analyze customer loyalty by ranking customers based on the average number of days between orders:

```
WITH Something AS (
	SELECT
		orderid,
		customerid,
		orderdate,
		LEAD(orderdate) OVER (PARTITION BY customerid ORDER BY orderdate) AS NextOrder,
		DATEDIFF(LEAD(orderdate) OVER (PARTITION BY customerid ORDER BY orderdate), orderdate) AS DaysDifferent
	FROM orders
	ORDER BY customerid, orderdate
) SELECT
	customerid,
	AVG(DaysDifferent) AS AvgDays,
	RANK() OVER (ORDER BY COALESCE(AVG(DaysDifferent), 999999)) AS AvgDaysRank
FROM Something
GROUP BY customerid;
```

customerid	AvgDays	AvgDaysRank
1	18.0000	1
2	34.0000	2

3	34.5000	3
4	NULL	4

FIRST_VALUE

Access a value from the first row within a window.

LAST_VALUE

Access a value from the last row within a window.

Default Frame:

RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

Example:

```
SELECT
    sales,
    MONTH(orderdate) AS Month,
    FIRST_VALUE(sales) OVER (ORDER BY MONTH(orderdate)) AS first,
    LAST_VALUE(sales) OVER (ORDER BY MONTH(orderdate)) AS last
FROM orders;
```

sales	Month	first	last
10	1	10	60
15	1	10	60
20	1	10	60
60	1	10	60
25	2	10	90
50	2	10	90
30	2	10	90

90	2	10	90
20	3	10	60
60	3	10	60

LAST_VALUE Must specify Frame or doesn't work properly!

```
SELECT
	sales,
	MONTH(orderdate) AS Month,
	FIRST_VALUE(sales) OVER (ORDER BY MONTH(orderdate)) AS first,
	LAST_VALUE(sales) OVER (
		ORDER BY MONTH(orderdate)
		RANGE BETWEEN
			CURRENT ROW
			AND
			UNBOUNDED FOLLOWING
	) AS last
FROM orders;
```

sales	Month	first	last
10	1	10	60
15	1	10	60
20	1	10	60
60	1	10	60
25	2	10	60
50	2	10	60
30	2	10	60
90	2	10	60
20	3	10	60
60	3	10	60

-- Find the lowest and highest sales for each product:

```
SELECT
        orderid,
        productid,
        sales,
        FIRST_VALUE(sales) OVER (PARTITION BY productid ORDER BY sales) AS LowestSales,
        LAST_VALUE(sales) OVER (
                PARTITION BY productid
                ORDER BY sales
                RANGE BETWEEN
                        CURRENT ROW
                        AND
                        UNBOUNDED FOLLOWING
        ) AS HighestSales
FROM orders;
```

orderid	productid	sales	LowestSales	HighestSales
1	101	10	10	90
3	101	20	10	90
9	101	20	10	90
8	101	90	10	90
2	102	15	15	60
7	102	30	15	60
10	102	60	15	60
5	104	25	25	50
6	104	50	25	50

4	105	60	60	60

(Use DESC in ORDER BY to change order to highest to lowest)

Can get exact same results using MIN and MAX aggregrate Window functions.
(But less flexibility)

Use Cases:

Time-Series Analysis: MoM + YoY

Time Gaps Analysis: Customer Retention

Comparison Analysis: Extreme: Highest, Lowest

Advanced SQL Techniques

Many different analysts writing complex queries to access a database,
along with data engineers: Extract, Transform, Load, put into Data Warehouse.

Data Warehouse: A special database that collects and integrates data from different sources to enable analytics and support decision-making.

Data Analyst may query this DW, or it is queried to put into a visual report.

This system is called a 'Data Warehouse System' or 'Business Intelligence System'.

Also, Data Scientists may query and manipulate data for using models and possibly machine learning and AI.

Results of Data Analysis may query these results for other visual presentations.

Challenges:

Each person is making their own queries, not communicating with each other.

But, a lot of redundant logic is being implemented by all of them.

All of these queries can hurt performance.

A data model may be prepared and optimized for only one purpose, so other analysts may have a ton of questions for their own purposes, making things even more complex as it has to be constantly explained.

This is all hard to maintain.

The DB also receives a lot of stress, and can crash.

Security can always be a problem as well. (DROP TABLE users SQL injection, anyone?)

Simplified Database Architecture:

Database Engine: The brain of the database, executing multiple operations such as storing, retrieving, and managing data within the database.

Disk Storage: Long-term memory, where data is stored permanently.

+Large Capacity
-Slow to read and write

User Data Storage: The main content of the database. This is where the actual data that users care about is stored.

System Catalog: DB's internal storage for its own information.
A blueprint that keeps track of everything about the database itself, not the user data.
Holds metadata information about DB.
metadata: 'Data about data'. (Such as the table schema)

Temp Data Storage: Temporary space used by the database for short-term tasks, like processing queries or sorting data. Once these tasks are done, the storage is cleared.

Cache: Short-term memory, store data temporarily.

+Fast to read and write
-Can only store smaller amounts of memory

Data Engineer sends query to database engine.

Checks if it is in cache.

No? Then checks disk, queries, then sends it back to DE.

[Subqueries]

A query inside another query.

Subquery is inside main query.

Results of subquery stays inside main query, where the main query uses it in different ways.

SQL drops results of subquery when it is no longer in use.

Why subqueries?

1. JOIN Tables
2. Filtering
3. Transformations
4. Aggregations

Can make for long and complicated query.

Instead, one query for each step, all in one main query.

Main Query AKA Outer Query.

Subquery AKA Inner Query.

DB Engine executes the subquery first, result gets stored in the cache, where it is fast to retrieve. Now it executes the main query, accessing the cache to get results of subquery. Then, after reporting results back, the cache is cleared.

Dependency:

Non-Correlated Subquery:
Independent from the main query.

Execution is straight-forward.
Executed once and its results are used by the main query.
Can be executed on its own.
Better performance.

Static comparisons, filtering with constants.

Correlated Subquery:

Dependent on the main query.
Relies on values from the main query.

Cannot be executed on its own.
Bad performance.

Do row-by-row comparison, more dynamic filters.

Executes main query first, gets results row-by-row.
Passes value to subquery, gets results.
Now main query checks results, if given, output row in final result.
Repeat process until all rows processed.

Example:

-- Show all customer details and find the total orders of each customer:

```
SELECT
	*,
	(
		SELECT
			COUNT(*)
		FROM orders o
		WHERE o.customerid = c.customerid
	) AS TotalSales
FROM customers c;
```

customerid	firstname	lastname	country	score	TotalSales
1	Jossef	Goldberg	Germany	350	3
2	Kevin	Brown	USA	900	3
3	Mary	NULL	USA	750	3
4	Mark	Schwarz	Germany	500	1
5	Anna	Adams	USA	NULL	0

Result Types:

Scalar

Like:

SELECT AVG(sales) FROM orders;

AVG(sales)
38.0000

Row

Like:

SELECT customerid FROM orders;

customerid
1
1
1
2
2
2
3
3
3
4

Table

Like:

SELECT customerid, sales FROM orders;

customerid	sales

2	10
3	15
1	20
1	60
2	25
3	50
1	30
4	90
2	20
3	60

Location|Clauses:

SELECT

Used to aggregate the data side by side with the main query's data, allowing for direct comparison.

Only Scalar subqueries are allowed to be used!

```
SELECT
	productid,
	product,
	price,
	(SELECT COUNT(*) FROM orders) AS TotalOrders
FROM products;
```

productid	product	price	TotalOrders
101	Bottle	10	10
102	Tire	15	10
103	Socks	20	10
104	Caps	25	10
105	Gloves	30	10

FROM

Use as temporary table for the main query.

Crap Example:

SELECT sales FROM (SELECT customerid, sales FROM orders) AS some_alias;

sales
10
15
20
60
25
50
30
90
20
60

-- Find the products that have a price higher than the average price of all products:

```
SELECT * FROM (
	SELECT
		productid,
		price,
		AVG(price) OVER () AS AvgPrice
	FROM products
) AS subquery
WHERE price > AvgPrice;
```

productid	price	AvgPrice
104	25	20.0000
105	30	20.0000

--Rank the customers based on thier otal amount of sales:

```
SELECT
	*,
	RANK() OVER (ORDER BY TotalSales DESC) AS CustomerRank
	FROM (
		SELECT
			customerid,
			SUM(sales) AS TotalSales
		FROM orders
		GROUP BY customerid
	) AS subquery;
```

customerid	TotalSales	CustomerRank

3	125	1
1	110	2
4	90	3
2	55	4

JOIN

Used to prepare the data (filtering or aggregation) before joining it with other tables.

-- Show all customer details and find the total orders of each customer:

```
SELECT
	c.*,
	o.TotalOrders
FROM customers c
LEFT JOIN (
	SELECT
		customerid,
		COUNT(*) AS TotalOrders
	FROM orders
	GROUP BY customerid
) AS o
ON o.customerid = c.customerid;
```

customerid	firstname	lastname	country	score	TotalOrders
1	Jossef	Goldberg	Germany	350	3
2	Kevin	Brown	USA	900	3
3	Mary	NULL	USA	750	3
4	Mark	Schwarz	Germany	500	1
5	Anna	Adams	USA	NULL	NULL

WHERE (Comparison or Logical operators)

Used for complex filtering logic and makes query more flexible and dynamic.

No aliases needed for subquery!

Comparison:

Only scalar subqueries are allowed to be used!

-- Sales higher than average:

```
SELECT
	customerid,
	sales FROM orders
WHERE sales > (
	SELECT AVG(sales)
	FROM orders
);
```

customerid	sales
1	60
3	50
4	90
3	60

-- Find the products that have a price higher than the average price of all products

```
SELECT
	productid,
	price
FROM products
WHERE price > (
	SELECT AVG(price)
	FROM products
);
```

productid	price

104	25
105	30

Logical:

The subquery is allowed to have multiple rows.

IN

Checks whether a value matches any value from a list.

-- Show the details of orders made by customers in Germany:

```
SELECT
	customerid,
	orderid,
	orderdate
FROM orders
WHERE customerid IN (
	SELECT
		customerid
	FROM customers
	WHERE country = 'Germany'
);
```

customerid	orderid	orderdate
1	3	2025-01-10
1	4	2025-01-20
1	7	2025-02-15
4	8	2025-02-18

-- Not from Germany:

```
SELECT
	customerid,
	orderid,
	orderdate
FROM orders
WHERE customerid NOT IN (
	SELECT
		customerid
```

```
	FROM customers
	WHERE country = 'Germany'
);
```

customerid	orderid	orderdate
2	1	2025-01-01
3	2	2025-01-05
2	5	2025-02-01
3	6	2025-02-05
2	9	2025-03-10
3	10	2025-03-15

ANY

Checks if a value matches ANY value within a list.

Used to check if a value is true for AT LEAST one of the values in a list.

-- Find female employees whose salaries are greater than the salaries of any male employee

```
SELECT
	employeeid,
	firstname,
	salary
FROM employees
WHERE gender = 'F'
AND salary > ANY (
	SELECT
		salary
	FROM employees
	WHERE gender = 'M'
);
```

employeeid	firstname	salary
3	Mary	75000

ALL

Checks if a value matches ALL values within a list.

-- Find male employees whose salaries are greater than the salaries of all female employees:

```
SELECT
	employeeid,
```

```
	firstname,
	salary
FROM employees
WHERE gender = 'M'
AND salary > ALL (
	SELECT
		salary
	FROM employees
	WHERE gender = 'F'
);
```

employeeid	firstname	salary
4	Michael	90000

EXISTS

Check if a subquery returns any rows.

For each row in Main Query:

Run subquery

No result? Row of main query excluded.

Returns value? Row of main query is included.

Often SELECT 1 is used because its value doesn't really matter and a static value is easier.

-- Show the order details for customers in Germany:

```
SELECT
	orderid,
	customerid,
	orderdate
FROM orders o
WHERE EXISTS (
	SELECT
		1
	FROM customers c
	WHERE country = 'Germany' AND o.customerid = c.customerid
);
```

orderid	customerid	orderdate
3	1	2025-01-10
4	1	2025-01-20

7	1	2025-02-15
8	4	2025-02-18

Use **NOT EXISTS** for customer not in Germany:

```
SELECT
	orderid,
	customerid,
	orderdate
FROM orders o
WHERE NOT EXISTS (
	SELECT
		1
	FROM customers c
	WHERE country = 'Germany' AND o.customerid = c.customerid
);
```

orderid	customerid	orderdate
1	2	2025-01-01
2	3	2025-01-05
5	2	2025-02-01
6	3	2025-02-05
9	2	2025-03-10
10	3	2025-03-15

CTE: Common Table Expression

AKA Temporary named result set (virtual table) that can be used multiple times within your query to simplify and organize complex query.

Executes CTE Query first, get intermediate, temporary result table.
Main query can retrieve data from temporary result table any time.
Once main query ends, SQL destroys temporary table.

Outside query cannot access this CTE table, dedicated only to inside main query.

CTE written top to bottom, while subquery is written bottom to top.

How are they really different?

Subquery result can only be used once in one place,
while CTE can be used as many times as needed, in any part of the main query.

When to use?

If repeating the same steps more than once, better to use a CTE that can be pulled from as many times as needed. Using subqueries instead would create redundancy, CTE removes the redundancy.

If many complex tasks, can break them up into pieces to work with one at a time (Divide & Conquer), and can be used again in other queries. (Readability, Modularity, Reusability)

CTE table also gets stored in the cache making data retrieval from it very efficient.

Cannot use ORDER BY clause in CTE! (But can in main query)

Best practices:

People tend to overuse the CTE.

Try to not add a CTE for each new calculation or each new column. It ends up being a massive mess that is hard to understand. Might also end up having bad performance.

Better to rethink and refactor multiple CTEs into one.

Probably don't use more than 5 CTEs in one query.

Example:

```
WITH CTE_Example AS (                          <- CTE
        SELECT
                orderid,
                orderdate FROM orders
        )
        SELECT *                                       <- Main query
        FROM CTE_Example
        ORDER BY orderdate DESC;
```

orderid	orderdate
10	2025-03-15
9	2025-03-10
8	2025-02-18
7	2025-02-15
6	2025-02-05
5	2025-02-01
4	2025-01-20
3	2025-01-10
2	2025-01-05
1	2025-01-01

Non-Recursive CTE:

CTE Executed only once without any repetition.

Standalone CTE

Defined and Used independently.

Runs independently as it's self-contained and doesn't rely on other CTEs or queries.

CTE queries DB, gets intermediate result. Then, main query, which is dependent on the intermediate result, uses it to get the final result.

Examples:

```
WITH CTE_TotalSales AS (
	SELECT
		customerid,
		SUM(sales) AS TotalSales
	FROM orders
	GROUP BY customerid
)
SELECT
	c.customerid,
	firstname,
	lastname,
	cts.TotalSales
FROM customers c
LEFT JOIN CTE_TotalSales cts
ON cts.customerid = c.customerid;
```

customerid	firstname	lastname	TotalSales

1	Jossef	Goldberg	110
2	Kevin	Brown	55
3	Mary	NULL	125
4	Mark	Schwarz	90
5	Anna	Adams	NULL

Multiple Standalone CTE:

```
WITH CTE_TotalSales AS (
	SELECT
		customerid,
		SUM(sales) AS TotalSales
	FROM orders
	GROUP BY customerid
),
CTE_LastOrder AS (
	SELECT
		customerid,
		MAX(orderdate) AS LastOrder
	FROM orders
	GROUP BY customerid
)
SELECT
	c.customerid,
	firstname,
	lastname,
	cts.TotalSales,
	clo.LastOrder
FROM customers c
LEFT JOIN CTE_TotalSales cts
ON cts.customerid = c.customerid
LEFT JOIN CTE_LastOrder clo
ON clo.customerid = cts.customerid;
```

customerid	firstname	lastname	TotalSales	LastOrder
1	Jossef	Goldberg	110	2025-02-15
2	Kevin	Brown	55	2025-03-10

3	Mary	NULL	125	2025-03-15
4	Mark	Schwarz	90	2025-02-18
5	Anna	Adams	NULL	NULL

Nested CTE

CTE inside another CTE. A nested CTE uses the result of another CTE, so it can't run independently.

DB <-> #1 CTE -> Intermediate 1 <-> #2 CTE -> Intermediate 2 (final) <-> Main Query -> Final Result

```
^^^^^^^^^^^^^^    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Standalone CTE          Nested CTE
```

Example:

```
WITH CTE_TotalSales AS (
	SELECT
		customerid,
		SUM(sales) AS TotalSales
	FROM orders
	GROUP BY customerid
),
CTE_LastOrder AS (
	SELECT
		customerid,
		MAX(orderdate) AS LastOrder
	FROM orders
	GROUP BY customerid
),
CTE_CustomerRank AS (
	SELECT
		customerid,
		TotalSales,
		RANK() OVER (ORDER BY TotalSales DESC) AS CustomerRank
	FROM CTE_TotalSales
)
SELECT
	c.customerid,
	firstname,
	lastname,
	cts.TotalSales,
	clo.LastOrder,
	ccr.CustomerRank
FROM customers c
LEFT JOIN CTE_TotalSales cts
ON cts.customerid = c.customerid
```

```
LEFT JOIN CTE_LastOrder clo
ON clo.customerid = c.customerid
LEFT JOIN CTE_CustomerRank ccr
ON ccr.customerid = c.customerid;
```

customerid	firstname	lastname	TotalSales	LastOrder	CustomerRank
1	Jossef	Goldberg	110	2025-02-15	2
2	Kevin	Brown	55	2025-03-10	4
3	Mary	NULL	125	2025-03-15	1
4	Mark	Schwarz	90	2025-02-18	3
5	Anna	Adams	NULL	NULL	NULL

Another:

```
WITH CTE_TotalSales AS (
	SELECT
		customerid,
		SUM(sales) AS TotalSales
	FROM orders
	GROUP BY customerid
),
CTE_LastOrder AS (
	SELECT
		customerid,
		MAX(orderdate) AS LastOrder
	FROM orders
	GROUP BY customerid
),
CTE_CustomerRank AS (
	SELECT
		customerid,
		TotalSales,
		RANK() OVER (ORDER BY TotalSales DESC) AS CustomerRank
	FROM CTE_TotalSales
),
CTE_CustomerSegments AS (
	SELECT
		customerid,
		CASE
			WHEN TotalSales > 100 THEN 'High'
			WHEN TotalSales > 50 THEN 'Medium'
			ELSE 'Low'
		END AS CustomerSegments
	FROM CTE_TotalSales
```

```
) SELECT
	c.customerid,
	firstname,
	lastname,
	cts.TotalSales,
	clo.LastOrder,
	ccr.CustomerRank,
	ccs.CustomerSegments
FROM customers c
LEFT JOIN CTE_TotalSales cts
ON cts.customerid = c.customerid
LEFT JOIN CTE_LastOrder clo
ON clo.customerid = c.customerid
LEFT JOIN CTE_CustomerRank ccr
ON ccr.customerid = c.customerid
LEFT JOIN CTE_CustomerSegments ccs
ON ccs.customerid = c.customerid;
```

customerid	firstname	lastname	TotalSales	LastOrder	CustomerRank	CustomerSegments
1	Jossef	Goldberg	110	2025-02-15	2	High
2	Kevin	Brown	55	2025-03-10	4	Medium
3	Mary	NULL	125	2025-03-15	1	High
4	Mark	Schwarz	90	2025-02-18	3	Medium
5	Anna	Adams	NULL	NULL	NULL	NULL

Recursive CTE

Self-referencing query that repeatedly processes data until a specific condition is met.

Used to travel through hierarchical structures.

Syntax:

```
WITH CTE_Name AS (
	SELECT ...			<- Anchor query
	FROM ...
	WHERE ...

	UNION ALL

	SELECT ...			<- Recursive query
	FROM CTE_Name
	WHERE (Break Condition)
```

```
)

SELECT ...                              <- Main query
FROM CTE_Name
WHERE ...
```

Example:

-- Generate a sequence of numbers from 1 to 20:

```
WITH RECURSIVE Series AS (
        SELECT 1 AS MyNumber
        UNION ALL
        SELECT MyNumber + 1
        FROM Series
        WHERE MyNumber < 20
)
SELECT * FROM Series;
```

MyNumber
1
2
3
4
5
6
7
8
9
10
11
12

13	
14	
15	
16	
17	
18	
19	
20	

-- Show the employee hierarchy by displaying each employees level within the organization:

```
WITH RECURSIVE CTE_EmployeeTree AS (
	SELECT
		employeeid,
		firstname,
		managerid,
		1 AS Level
	FROM employees
	WHERE managerid IS NULL
	UNION ALL
	SELECT
		e.employeeid,
		e.firstname,
		e.managerid,
		Level + 1
	FROM employees AS e
	INNER JOIN CTE_EmployeeTree cet
	ON e.managerid = cet.employeeid
)
SELECT * FROM CTE_EmployeeTree;
```

employeeid	firstname	managerid	Level
1	Frank	NULL	1
2	Kevin	1	2

3	Mary	1	2
4	Michael	2	3
5	Carol	3	3

Levels:

```
          1
       /     \
      /       \
     2         2
     |         |
     3         3
```

Views

Database Structure

DDL (Data Definition Language):
A set of commands that allows us to define and manage the structure of a database.

Database Server:
Stores manages, and provides access to databases for users or applications.

|
v

Database:
Collection of information that is stored in a structured way.

|
v

Schema:
Logical layer that groups related objects together.

|
v

Table:
A place where data is stored and organized into rows and columns.

OR

View:

A virtual table that shows data without storing it physically.
Doesn't store data permanently.

Database 3 Level Architecture (3 Abstraction levels):
(More abstract going up from Physical)

View Level (AKA External)

Users:

- Business Analysts
- POWER BI
- End Users
- etc.

Only views

Logical Level (AKA Conceptual)

App Developer (Data Engineer)

- Tables
- Relationships
- Views
- Indexes
- Procedures
- Functions

Physical Level (AKA Internal)

DB Administrator

- DB
- Data Files
- Partittions
- Logs

Catalog
Blocks
Caches

View:

Virtual table based on the result set of a query, without storing the data in the DB.
Views are persisted SQL queries in DB.

VIEWS	TABLES
No persistance	Persisted Data
Easy to maintain	Hard to maintain
Slow Response	Fast Response
Read only	Read/Write

Use Cases:

Central Query Logic: Store central, complex query logic in the database,
for access by multiple queries, reducing project complexity.

If multiple users do the same task such as with a CTE, maybe should store the CTE as
a view so all users can just use that view instead of all writing the same CTE each time.

VIEWS	CTE
Reduce redundancy in multi-queries	Reduce redundancy in 1 query
Improve reusability in multi-queries	Improve reusability in 1 query
Persisted Logic	Temporary logic (on the fly)
Need to maintain (Create/Drop)	No maintenance (Auto cleanup)

SQL Views:

CREATE

CREATE VIEW View_Name AS (Query)

If a Table or View is created without specifying a schema, it defaults to the DBO.
(Maybe in SQL Server, but doesn't seem to in MySQL?)
(Found under 'salesdb.')

Example:

```
CREATE VIEW V_Monthly_Summary AS (
	SELECT
		MONTH(orderdate) AS OrderMonth,
		SUM(sales) AS TotalSales, COUNT(orderid) AS TotalOrders,
		SUM(quantity) AS TotalQuantity
	FROM orders
	GROUP BY MONTH(orderdate)
);
Query OK, 0 rows affected (0.17 sec)
SELECT * FROM V_Monthly_Summary;
```

OrderMonth	TotalSales	TotalOrders	TotalQuantity
1	105	4	6
2	195	4	8
3	80	2	2

```
SELECT
	OrderMonth,
	SUM(TotalSales) OVER (ORDER BY OrderMonth) AS RunningTotal
FROM V_Monthly_Summary;
```

OrderMonth	RunningTotal
1	105
2	300
3	380

UPDATE

In MySQL:

```
mysql> CREATE OR REPLACE VIEW V_Monthly_Summary AS (
	SELECT
		MONTH(orderdate) AS OrderMonth,
		SUM(sales) AS TotalSales,
		COUNT(orderid) AS TotalOrders,
		SUM(quantity) AS TotalQuantity
	FROM orders
	GROUP BY MONTH(orderdate)
);
Query OK, 0 rows affected (0.15 sec)
```

DROP

DROP VIEW View_Name

Example:

DROP VIEW V_Monthly_Summary;
Query OK, 0 rows affected (0.24 sec)

Data Engineer (DE) -> CREATE VIEW -> DB Engine -> Disk: Catalog (Metadata)

Data Analyst (DA) -> Query -> DB Engine -> Disk: Catalog -> View
View -> DB Engine -> View Query -> Disk: User -> View Query -> DB Engine -> Query -> DA

DROP VIEW only drops metadata, doesn't drop any tables or actual data contained.

Use Case:

Views can be used to hide the complexity of database tables and offers users more friendly and easy-to-consume objects.

Some DBs can be very complex with cryptic table names and relationships between them.

DBA can create views that are an abstraction of these complexities.

Example:

-- Provide a view that combines details from orders, products, customers, and employees:

```
CREATE VIEW V_Order_Details AS (
	SELECT
		o.orderid,
		o.orderdate,
		p.product,
		p.category,
		CONCAT(COALESCE(c.firstname, ''), ' ', COALESCE(c.lastname, '')) AS CustName,
		CONCAT(COALESCE(e.firstname, ''), ' ', COALESCE(e.lastname, '')) AS SalesName,
		e.department,
		o.sales,
		o.quantity
	FROM orders o
	LEFT JOIN products p
	ON p.productid = o.productid
	LEFT JOIN customers c
	ON c.customerid = o.customerid
	LEFT JOIN employees e
```

```
    ON e.employeeid = o.salespersonid
);
```

mysql> SELECT * FROM V_Order_Details;

orderid	orderdate	product	category	CustName	SalesName	department	sales	quantity
1	2025-01-01	Bottle	Accessories	Kevin Brown	Mary	Sales	10	1
2	2025-01-05	Tire	Accessories	Mary	Mary	Sales	15	1
3	2025-01-10	Bottle	Accessories	Jossef Goldberg	Carol Baker	Sales	20	2
4	2025-01-20	Gloves	Clothing	Jossef Goldberg	Mary	Sales	60	2
5	2025-02-01	Caps	Clothing	Kevin Brown	Carol Baker	Sales	25	1
6	2025-02-05	Caps	Clothing	Mary	Carol Baker	Sales	50	2
7	2025-02-15	Tire	Accessories	Jossef Goldberg	Frank Lee	Marketing	30	2
8	2025-02-18	Bottle	Accessories	Mark Schwarz	Mary	Sales	90	3
9	2025-03-10	Bottle	Accessories	Kevin Brown	Mary	Sales	20	2
10	2025-03-15	Tire	Accessories	Mary	Carol Baker	Sales	60	0

Views Use Case:

Use views to enforce security and protect sensitive data by hiding columns and/or rows from tables.

Shouldn't give same level of access to all users, remove access to actual, physical tables by using views.

Create different view for each level of user.

All Data, Column-Security, Column-Security/Row-Security.

Example:

--Provide a view for EU Sales Team that combines the details from all tables AND excludes data related to the USA:

```
CREATE VIEW V_Order_Details_EU AS (
	SELECT
		o.orderid,
		o.orderdate,
		p.product,
		p.category,
		CONCAT(COALESCE(c.firstname, ''), ' ', COALESCE(c.lastname, '')) AS CustName,
		CONCAT(COALESCE(e.firstname, ''), ' ', COALESCE(e.lastname, '')) AS SalesName,
		e.department,
		o.sales,
		o.quantity
	FROM orders o
	LEFT JOIN products p
	ON p.productid = o.productid
	LEFT JOIN customers c
	ON c.customerid = o.customerid
	LEFT JOIN employees e
	ON e.employeeid = o.salespersonid
```

```
    WHERE c.country != 'USA'
);

SELECT * FROM V_Order_Details_EU;
```

orderid	orderdate	product	category	CustName	SalesName	department	sales	quantity
3	2025-01-10	Bottle	Accessories	Jossef Goldberg	Carol Baker	Sales	20	2
4	2025-01-20	Gloves	Clothing	Jossef Goldberg	Mary	Sales	60	2
7	2025-02-15	Tire	Accessories	Jossef Goldberg	Frank Lee	Marketing	30	2
8	2025-02-18	Bottle	Accessories	Mark Schwarz	Mary	Sales	90	3

Use Case:

More flexibility in project, doing lots of changes in tables.
That means all the users now have to change their complex queries with every change.
So, changes can't be made without talking to tons of users.

Solution: Make view given old structure users are using, but make changes to actual tables.

Use Case:

Can make several versions (views) of the data model, one for each language (English, Spanish, German, etc.).

This includes changing names of tables, columns, and so on.
Use Case:

Views can be used as 'Data Marts' in Data Warehouse System because they provide a flexible and efficient way to present data.

Source Systems -> Data Warehouse -> Data Marts (Sales Mart, Finance Mart, etc.) -> Reporting

If building Data Marts, should use VIEWs, have as Virtual Layer as opposed to the Physical Layer of the Data Warehouse.

CTAS And Temp Tables

Create/Insert

1. Create: Define the structure of table.
    CREATE empty table.

2. Insert: Insert data into the table.
    INSERT data into the new table.

Create from scratch.

CTAS (Create Table As Select)

Create a new table based on the result of an SQL query.

1. Execute query on table, use result to create new table.

Create given existing table(s).

In VIEWs, query of view is executed each time it is used, while CTAS query has already been executed.

CTAS don't need to execute an extra query on original, source table every time it is used.

So, CTAS perform faster than VIEWs.

Problem: Modifying the original source tables modifies the results of the VIEW, but not the CTAS!

VIEW: Fresh to order pizza.

CTAS: Frozen, already made pizza.

CTAS Syntax:

```
CREATE TABLE Name AS (
    SELECT ...
    FROM ...
    WHERE ...
)
```

Simple Example:

```
CREATE TABLE Derp AS (
    SELECT
        orderid,
        orderdate
    FROM orders
    WHERE MONTH(orderdate) = 1
```

);
Query OK, 4 rows affected (0.72 sec)
Records: 4 Duplicates: 0 Warnings: 0

SELECT * FROM Derp;

orderid	orderdate
1	2025-01-01
2	2025-01-05
3	2025-01-10
4	2025-01-20

Use Cases:

Use CTAS to take workload behind the scenes, then end users have faster experience.
In comparison to VIEWs, where the workload can end up on the user end.

Example:

```
CREATE TABLE MonthlyOrders AS (
        SELECT
                DATE_FORMAT(orderdate, '%M') AS OrderMonth,
                COUNT(orderid) AS TotalOrders
        FROM orders
        GROUP BY DATE_FORMAT(orderdate, '%M')
);
Query OK, 3 rows affected (0.68 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

SELECT * FROM MonthlyOrders;

OrderMonth	TotalOrders
January	4
February	4
March	2

Refresh CTAS:

DROP Table and run CREATE TABLE again.

Use Case: Create a snapshot of data at a specific time

Use Case: Persisting the Data Marts of a DWH improves the speed of data retrieval compared to using views.

Using VIEWS can end up slowing everything down.

Use CTAS from Data Warehouse to Data Mart instead (still slow), but then speeds up the exchange between Data Mart and Reporting.

Temporary Tables

Stores intermediate results in temporary storage within the database during the session.
The database will drop all temporary tables once the session ends.

Temp Table query uses existing table, creates new table, but then this table is dropped automatically after the session ends.

User -> Query -> DB Engine -> Server: Disk: (Metadata) Catalog / (Table) TEMP
Ending session removes catalog metadata and temp table.

Can extract data from source DB, puts into temp table.
Now, transformation can be done onto temp table.
Results get put into DWH DB.
Now, temp table no longer needed, can be dropped.

This way, no need to combine transformation and extract step into one, slow process.

Syntax (MySQL):

```
CREATE TEMPORARY TABLE Name AS (
	SELECT ...
	FROM ...
	WHERE ...
)
```

Example:

```
CREATE TEMPORARY TABLE Derp2 AS (
	SELECT
		orderid,
		orderdate
	FROM orders
	WHERE MONTH(orderdate) = 1
);
Query OK, 4 rows affected (0.01 sec)
Records: 4  Duplicates: 0  Warnings: 0
```

```
SELECT * FROM Derp2;
```

orderid	orderdate
1	2025-01-01
2	2025-01-05
3	2025-01-10
4	2025-01-20

(Drops after logging out of session)

-- Do anything to table:

```
DELETE FROM Derp2 WHERE DAY(orderdate) > 5;
Query OK, 2 rows affected (0.00 sec)
```

```
SELECT * FROM Derp2;
```

orderid	orderdate
1	2025-01-01
2	2025-01-05

-- If you decide to make it more permanent:

CREATE TABLE Derp3 AS (SELECT * FROM Derp2);
Query OK, 2 rows affected (1.24 sec)
Records: 2 Duplicates: 0 Warnings: 0

Tip:

CTE can be used for intermediate result for one query, if deciding it is more important, put into CTAS. Not really much need for temp table in real usage.

[Compare Advanced Techniques]

Property	CTAS	VIEW	Subquery	CTE	Temp
Storage:	Disk	No Storage	Memory	Memory	Disk
Lifetime:	Temp	Temp	Temp	Perm	Perm
When Delete:	Query	Query	Session	Drop	Drop
Scope:	Single-Query	Single-Query	Multi	Multi	Multi
Re-usability:	1 place, 1 query	*place, 1 query	*query, 1 session	*query	*query
up2date:	Snapshot	Always	Always	Always	Snapsh

					o t

* = More than 1

Stored Procedures

Writing procedure: Ordering at coffee shop, orders string of different specific parts.

Stored procedure: I want my usual! (No need to specify)

Stored procedures stored inside DB. Call with EXEC.

Stored Procedure	Query
Many interactions	1 interaction
Like a program: Loops, control flow, parameters, error handling	Request

Stored Procedure	Python code
Has connection to DB	Has to create connection to DB
Pre-Compiled	Interpreted every run
Less flexible	Flexibility, Version control
Hard to deal with complex logic	Complex logic is easier

Python is the preferred alternative to using stored procedures, especially as it gets more complex and working with many people.

Syntax:

```
CREATE PROCEDURE ProcName AS
BEGIN
	--SQL Statements here
END

-- Execute procedure:

EXEC ProcName
```

Example (In MySQL):

-- Need to change delimiter if doing this on command line:

```
mysql> delimiter //
mysql> CREATE PROCEDURE GetCustomerSummary()
BEGIN
	SELECT
		COUNT(*) AS TotalCustomers,
		AVG(score) AS AvgScore
	FROM customers
	WHERE country = 'USA';
END //
mysql> delimiter ;

mysql> CALL GetCustomerSummary();
```

TotalCustomers	AvgScore
3	825.0000

-- With input paramater:

```
mysql> delimiter //
mysql> CREATE PROCEDURE GetCustomerSummary2(
	IN add_num int
) BEGIN
	SELECT
		COUNT(*) AS TotalCustomers,
		AVG(score) AS AvgScore,
		AVG(score) + add_num AS AddedAvg
	FROM customers
	WHERE country = 'USA';
END //
mysql> delimiter ;
```

mysql> CALL GetCustomerSummary2(100);

TotalCustomers	AvgScore	AddedAvg
3	825.0000	925.0000

Example using all parameter types
(Not using the usual salesdb database, just for example):

```
DELIMITER //

CREATE PROCEDURE GetCustomerDetails(
    IN customer_id INT,
    OUT customer_name VARCHAR(255),
    INOUT total_orders INT
)
BEGIN
    SELECT name INTO customer_name FROM customers WHERE id = customer_id;
    SELECT COUNT(*) INTO total_orders FROM orders WHERE customer_id = customer_id;
    SET total_orders = total_orders + 1; -- Example of modifying INOUT parameter
END //

DELIMITER ;
```

-- Calling:

```
SET @name = '';
SET @orders = 0;
CALL GetCustomerDetails(101, @name, @orders);
SELECT @name, @orders;
```

-- Can already see these stored procedures seem more trouble than worth...
ERROR 1267 (HY000): Illegal mix of collations (utf8mb4_0900_ai_ci,IMPLICIT) and (utf8mb4_unicode_ci,IMPLICIT) for operation '='

-- Just going to take notes without deep examples:

-- Can have multiple queries in one stored procedure.

```
mysql> delimiter //
mysql> CREATE PROCEDURE GetCustomerSummary2()
BEGIN
	SELECT
		COUNT(*) AS TotalCustomers,
		AVG(score) AS AvgScore
	FROM customers
	WHERE country = 'USA';

	SELECT
		COUNT(*) AS TotalCustomers,
		AVG(score) AS AvgScore
	FROM customers
	WHERE country = 'Germany';
END //

mysql> delimiter ;

mysql> CALL GetCustomerSummary2;
```

TotalCustomers	AvgScore
3	825.0000

TotalCustomers	AvgScore
2	425.0000

Declare variables with:

DECLARE @Something TYPE, @SomethingElse TYPE2;

@Something = SCALAR

Control flow:

IF EXISTS (Some Query, does it return anything?)
BEGIN
    --Do something, maybe UPDATE certain rows related to query in IF?
END

ELSE
BEGIN
    --Otherwise, do something else
END

Error handling:

BEGIN TRY
    -- SQL statements that might cause an error
END TRY

BEGIN CATCH
    -- SQL statements to handle the error
END CATCH

Tips:

Use indentation inside every BEGIN/END block.
Same with BEGIN TRY / END TRY block.

Add comments to make things clearer, throw in separators, etc.

Conclusion: Would much rather let Python handle this kind of stuff, way easier to work with.

Triggers

Trigger can be set for after or before event.
Can be used for audit logs, for example.

Types:

DML: INSERT|UPDATE|DELETE
DDL: CREATE|ALTER|DROP
LOGGON

Syntax:

```
CREATE TRIGGER TriggerName ON TableName

AFTER|BEFORE|INSTEAD OF INSERT|UPDATE|DELETE

BEGIN
	-- SQL Statements here
END
```

Simple Example:

-- Create example tables:

```
mysql> CREATE TABLE timestampers (times TIMESTAMP);

mysql> SELECT * FROM table_that_triggers; SELECT * FROM timestampers;

Empty set (0.00 sec)

Empty set (0.01 sec)
```

-- Create trigger:

```
mysql> DELIMITER //
mysql> CREATE TRIGGER timestamp_trigger
```

```
AFTER UPDATE ON table_that_triggers
FOR EACH ROW
        BEGIN
                INSERT INTO timestampers (times) VALUES (NOW());
        END //
mysql> DELIMITER ;
```

-- Put one row in table:

mysql> INSERT INTO table_that_triggers (some_id) VALUES (8);

mysql> SELECT * FROM table_that_triggers; SELECT * FROM timestampers;

some_id
8

Empty set (0.01 sec)

-- Update one of the example tables to activate the trigger:

mysql> UPDATE table_that_triggers SET some_id = 7 WHERE some_id = 8;
Query OK, 1 row affected (0.29 sec)
Rows matched: 1 Changed: 1 Warnings: 0

mysql> SELECT * FROM table_that_triggers; SELECT * FROM timestampers;

some_id
7

times
2025-09-12 14:48:33

Performance Optimization

Indexes

Index: Data structure that provides quick access to data, optimizing the speed of your queries.

Like an index at the back of a book to find exactly what you are looking for.

OR

Using a map to find a specific hotel room instead of looking room by room.

There is always a trade-off when using indexes: Some indexes are better for reading,
others are better for writing performance.

Page: The smallest unit of data storage in a database (8kb).
It can store anything (Data, Metadata, Indexes, etc.)
Types: Data, Index page.

Data Page
Header FileID:Page Number
Row1: 1, Bob, USA
Row2 2, Anna, Germany
....
Offset (Where row begins)

Inserting data, inserting into page until full, then inserts new page, and so on.
This is a heap structure, a table without clustered index.
Fast write, slow read, nothing is in order, all just tossed in there.

Table Full Scan: Scans entire table page by page and row by row, searching for data.
For a large table, this is a real performance problem.

Index Types:

Structure:

Clustered Index

Physically sort the data on something, such as the column id.
Builds a B-Tree (Balance Tree): Hierarchical structure storing data at leaves,
to help quickly locate data.

```
                        Root Node
                      /           \
     Intermediate Node    Intermediate Node
     /                                 \
  Leaf                                 Leaf
```

Leaf: Data pages.

(1-5)
1:100
1, Bob
2, Anna
3, John
4, Chris
...

Intermediate Nodes: Index Page: It stores key values (Pointers) to another page.
It doesn't store actual rows.

(1-10)
1:200
Key Value (Pointer to Data Page)
1 -> 1:100 (Go to 1-5)
...
6 -> 1:101

Root Node is also an index page:
1:300
1 -> 1:200 (Go to 1-10)
...
11 -> 1:201

Non-Clustered Index

Starting from heap, a non-clustered index won't reorganize or change anything on the data page.

Builds an index page:

1:200
CustomerID -> FileID:Page Number:Row Offset
1 -> 1:102:96

Points to exact location.

'Row locator' index pages, index pages sorted, but not data pages.

Also uses B-Tree, but the Leaf Nodes are the row locator index pages that point to the Data Pages.

One extra layer in comparison to the Clustered Index.

Clustered Index: Like Table of Contents in book, doesn't point to exactly where everything is.
Non-Clustered Index: Like Index at end of book, points to exactly where something is.

Will find both in one table!
NCI Leafs can point to CI Leafs!

Can find many NCI, but only 1 CI per table.

CI: Main Index.

CI trades write performance for read and storage performance.
NCI trades read and storage performance for write performance.

CI should sort on column that doesn't get modified, such as primary keys.
NCI uses columns frequently used in search conditions and joins.

	CI	NCI
Definition:	Physically sorts & stores rows	Separate structure with pointers to data
# of Indexes:	1 Index per table	Multiple Allowed
Read Performance:	Faster	Slower
Write Performance:	Slower	Faster
Storage:	Efficient	Requires additional space

Syntax for creating index:

Default
vvvvvvvvvvvv
CREATE [CLUSTERED | NONCLUSTERED] INDEX index_name ON table_name (column1, column2, ...)

Examples:

CREATE CLUSTERED INDEX IX_Customers_ID ON Customers (ID)
CREATE NONCLUSTERED INDEX IX_Customers_City ON Customers (City)
CREATE INDEX IX_Customers_Name ON Customers (LastName ASC, FirstName DESC)

MySQL View Indexes on Table:

SHOW INDEXES FROM Table;

A Primary Key automatically creates a clustered index by default.

!!!! **MySQL Note** !!!!

Uses a different setup:

CREATE [UNIQUE | FULLTEXT | SPATIAL] INDEX index_name
ON table_name (column1 [ASC | DESC] [, column2 [ASC | DESC], ...])
[USING BTREE | HASH];

Explanation of Components:

CREATE INDEX: This keyword initiates the creation of an index.

[UNIQUE | FULLTEXT | SPATIAL]: These are optional keywords to specify the type of index:

UNIQUE: Ensures that all values in the indexed columns are unique. If you try to insert a duplicate value, an error will occur.

FULLTEXT: Used for full-text searches on text-based columns (e.g., VARCHAR, TEXT).

SPATIAL: Used for indexing spatial data types (e.g., GEOMETRY).

index_name: The name you assign to the index. This name must be unique within the table.

ON table_name: Specifies the table on which the index will be created.

(column1 [ASC | DESC] [, column2 [ASC | DESC], ...]): This lists the columns to be included in the index.
You can specify one or more columns to create a composite index.
ASC: (default) or DESC: Specifies the sorting order for the index.

[USING BTREE | HASH]: This optional clause specifies the index type:

BTREE: (default): The standard index type suitable for most cases.

HASH: Can provide faster lookups for equality comparisons but is primarily used with MEMORY tables.

Example:

```
mysql> CREATE TABLE dbcustomers AS (SELECT * FROM customers);
Query OK, 5 rows affected (1.50 sec)
Records: 5  Duplicates: 0  Warnings: 0

mysql> SHOW INDEXES FROM dbcustomers;
Empty set (0.01 sec)

mysql> CREATE INDEX idx_dbcustomers_customerid ON dbcustomers (customerid);
Query OK, 0 rows affected (0.93 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> DROP INDEX idx_dbcustomers_customerid ON dbcustomers;
Query OK, 0 rows affected (0.33 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Can create indexes for columns often used in WHERE clause to speed up performance.

Composite index uses multiple columns.
The columns of index order must match the order in your query!

Column in index: A, B, C, D

--Index will be used:

A
A, B

--Index will not be used:
B
A, C
A, B, D

Storage:

Rowstore Index

Organize data row-by-row, multiple rows on each data page.
Rowstore Heap Table is default.

ColumnStore Index

Split tables into different columns on each data page.

#1. Split into row groups.
#2. Split row groups into column segments.
#3. Data compression. (Reason why ColumnStore is so fast and storage efficient).
#4. Store as Large oBject Page (LoB Page)

LoB Page

Header 1:200
Segment Header
    Segment ID = 1
    Rowgroup ID = 20
    Dictionary ID = 1:050

    Data Stream [1,1,2,2,1,2,1,1,1 ... 2,1,1] (Used with dictionary page)

Dictionary Page:
    1:050
    'A'->1
    'B'->2

Formats everything column-wide, restructures table.

Using rowstore reads through everything row-by-row, then filters data, aggregates.
Using columnstore uses only the relevant column, no need to pen extra data pages.

	RowStore	ColumnStore
Definition:	Row-By-Row	Column-By-Column
Storage:	Less efficient	Highly efficient with compression
Read/Write:	Fair speed for both	Fast read, slow write
I/O Efficiency:	Lower (Retrieves all)	Higher (Retrieves specific)
Best For:	OLTP (Transactional)	OLAP (Analytical)
Use Case:	High Freq. transaction apps	Big Data analytics, large dataset scans, fast aggr.

Syntax (SQL Server):

CREATE [CLUSTERED | NONCLUSTERED] [COLUMNSTORE] INDEX index_name ON table_name (column1, column2, ...)

If CLUSTERED, cannot use columns in statement.

Functions:

Unique Index

Ensures no duplicate values exist in specific column.

Benefits:

1. Enforce uniqueness.
2. Slightly increase query performance.

Writing to a unique index is slower than non-unique, but reading is faster.

Syntax (SQL Server):

CREATE [UNIQUE] [CLUSTERED | NONCLUSTERED] [COLUMNSTORE] INDEX index_name ON table_name (column1, column2, ...)

MySQL:

Use option UNIQUE after CREATE.

Doesn't work if column contains duplicates:

mysql> CREATE UNIQUE INDEX idx_dbcustomers_customerid ON dbcustomers (country);
ERROR 1062 (23000): Duplicate entry 'USA' for key 'dbcustomers.idx_dbcustomers_customerid'

Filtered Index

An index that includes only rows meeting the specified conditions.

Benefits:
- Targeted Optimization
- Reduced storage: Less data in the index

Syntax (SQL Server):

```
CREATE [UNIQUE] [CLUSTERED | NONCLUSTERED] [COLUMNSTORE] INDEX index_name
ON table_name (column1, column2, ...)
WHERE [Condition]
```

Rules:

Cannot create on a clustered index.
Cannot create on a columnstore index.

Doesn't work in MySQL:

```
mysql> CREATE INDEX idx_dbcustomers_customerid2 ON dbcustomers (country) WHERE country
= 'USA';
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that
corresponds to your MySQL server version for the right syntax to use near
'WHERE country = 'USA'' at line 1
```

Gemini:

"
MySQL does not directly support filtered indexes (also known as partial indexes) in the same way that some other database systems like SQL Server or PostgreSQL do. Filtered indexes allow you to create an index on a subset of rows in a table, based on a WHERE clause condition. However, similar effects can be achieved in MySQL through alternative methods:

Generated (Virtual or Stored) Columns with an Index:

You can create a generated column that stores the result of a condition, and then create an index on this generated column. This effectively creates an index only for the rows that meet the condition.

Code

```
ALTER TABLE your_table
ADD COLUMN is_active_generated TINYINT AS (
            CASE WHEN status = 'active' THEN 1 ELSE 0 END
      ) VIRTUAL;

CREATE INDEX idx_is_active_generated ON your_table (is_active_generated);
```

Prefix indexing:
For TEXT or VARCHAR columns, you can create an index on a prefix of the column, which can be useful for filtering based on the beginning of a string.

Code

```
CREATE INDEX idx_column_prefix ON your_table (your_text_column(10));
      -- Indexes the first 10 characters
```
"

Example:

```
mysql> ALTER TABLE dbcustomers
      ADD COLUMN is_usa TINYINT AS (
            CASE WHEN country = 'USA' THEN 1 ELSE 0 END
      ) VIRTUAL;
Query OK, 0 rows affected (0.51 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX idx_dbcustomers_is_usa ON dbcustomers (is_usa);
Query OK, 0 rows affected (0.52 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

HEAP: Fast Inserts (For staging tables)

Clustered Index: For primary keys. If not, then for date columns. (OLTP)

Columnstore Index: For analytical queries. Reduce size of large table. (OLAP)

Non-Clustered Index: For non-PK columns (Foreign keys, JOINs, and filters)

Filtered Index: Target subset of data. Reduce size of index.

Unique Index: Enforce uniqueness. Improve query speed.

Index Management and Monitoring

Indexes can become fragmented over time, needs regular maintenance.

1. Monitor index usage

List of all indexes (SQL Server):

```
SELECT
    object_id,
    tbl.name AS TableName,
    idx.name AS IndexName,
    idx.type_desc AS IndexType,
    idx.is_primary_key AS IsPrimaryKey,
    idx.is_unique AS IsUnique,
    idx.is_disabled AS IsDisabled
    FROM sys.indexes idx
    JOIN sys.tables tbl
    ON idx.object_id = tbl.object_id
    ORDER BY tbl.name, idx.name
```

Dynamic Management View (DMV):

Provides real-time insights into database performance and system health.

```
SELECT * FROM sys.dm_db_index_usage_stats
```

MySQL:

"

MySQL provides information about index usage primarily through the Performance Schema and the sys schema.

1. Performance Schema:
The performance_schema.table_io_waits_summary_by_index_usage table offers insights into index usage.

Code

```
SELECT
    object_schema,
    object_name,
    index_name,
    count_star,
    count_read,
    count_write
FROM
    performance_schema.table_io_waits_summary_by_index_usage
WHERE
    object_schema = 'your_database_name';
```

object_schema: The name of the database.
object_name: The name of the table.
index_name: The name of the index.
count_star: The total number of times the index has been used for I/O operations (reads and writes) since the last MySQL server restart.
count_read: The number of times the index was used for read operations.
count_write: The number of times the index was used for write operations.

2. sys.schema_unused_indexes:

The sys.schema_unused_indexes view lists indexes that have not recorded any usage events since the last server restart. This can help identify potentially redundant or unused indexes.

Code

```
SELECT
    object_schema,
    object_name,
    index_name
FROM
    sys.schema_unused_indexes;
```

Performance Schema Activation:
The Performance Schema must be enabled and configured to collect this data.

Server Restarts:
The usage statistics in performance_schema.table_io_waits_summary_by_index_usage and sys.schema_unused_indexes are reset upon a MySQL server restart.
"

Example:

```
mysql> SELECT
        object_schema,
        object_name,
        index_name,
        count_star,
        count_read,
        count_write
FROM
        performance_schema.table_io_waits_summary_by_index_usage
WHERE
        object_schema = 'salesdb';
```

object_schema	object_name	index_name	count_star	count_read	count_write
salesdb	orders	PRIMARY	0	0	0
salesdb	orders	productid	0	0	0
salesdb	orders	customerid	0	0	0
salesdb	orders	salespersonid	0	0	0
salesdb	products	PRIMARY	0	0	0
salesdb	customers	PRIMARY	0	0	0
salesdb	employees	PRIMARY	0	0	0
salesdb	employees	managerid	0	0	0
salesdb	dbcustomers	idx_dbcustomers_is_usa	0	0	0
salesdb	orders_archive	productid	0	0	0
salesdb	orders_archive	customerid	0	0	0
salesdb	orders_archive	salespersonid	0	0	0

```
mysql> SELECT
    object_schema,
    object_name,
    index_name
FROM
    sys.schema_unused_indexes;
```

object_schema	object_name	index_name
salesdb	dbcustomers	idx_dbcustomers_is_usa
salesdb	employees	managerid
salesdb	orders	productid
salesdb	orders	customerid
salesdb	orders	salespersonid
salesdb	orders_archive	productid
salesdb	orders_archive	customerid
salesdb	orders_archive	salespersonid

Use the index:

mysql> SELECT * FROM dbcustomers WHERE is_usa = 1;

```
mysql> SELECT
        object_schema,
        object_name,
        index_name,
        count_star,
        count_read,
        count_write
FROM
        performance_schema.table_io_waits_summary_by_index_usage
        WHERE
                object_schema = 'salesdb'
                AND
                index_name = 'idx_dbcustomers_is_usa';
```

object_schema	object_name	index_name	count_star	count_read	count_write
salesdb	dbcustomers	idx_dbcustomers_is_usa	3	3	0

Can then remove indexes if they aren't really being used regularly.
90% of indexes end up being unused in real projects.
Removing them saves storage and speeds up write performance.

2. Monitor Missing Indexes:

View index suggestions (SQL Server):

SELECT * FROM sys.dm_db_missing_index_details

Evaluate recommendations before creating any index.

MySQL:

"
Here's how you can find missing indexes or identify areas for index suggestions in MySQL:

Analyze Query Execution Plans with EXPLAIN:
Use the EXPLAIN statement before your SELECT queries to see how MySQL executes them.

Look for type values like ALL (full table scan) or index (full index scan), especially in large tables, as these often indicate the absence of an appropriate index.

Examine the Extra column for warnings like "Using filesort" or "Using temporary," which suggest that MySQL is performing expensive operations that could be optimized with indexes.

Code

```
EXPLAIN SELECT * FROM your_table WHERE your_column = 'some_value';
```

Monitor Slow Queries:

Enable the slow query log in your MySQL configuration to capture queries that exceed a specified execution time.

Analyze the slow query log to identify frequently executed or particularly slow queries that could benefit from indexing.
"

Example:

mysql> EXPLAIN SELECT * FROM dbcustomers WHERE country = 'USA';

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	dbcustomers	NULL	ALL	NULL	NULL	NULL	NULL	5	20.00	Using where

1 row in set, 1 warning (0.01 sec)

3. Monitor duplicate indexes:

View the information on indexes, use window function COUNT(*) with tbl.name and col.name to create flag.
If > 1, has duplicates.

4. Updating statistics:

When executing query, DB Engine has to create execution plan, does so by reading the statistics of the table.
Statistics doesn't automatically get update after, for example, adding a million rows to a table that previously only had a few.

SQL Server:

```
SELECT
	SCHEMA_NAME(t.schema_id) AS SchemaName,
	t.name AS TableName,
	s.name AS StatisticName,
	sp.last_updated AS LAstUpdate,
	DATEDIFF(day, sp.last_updated, GETDATE()) AS LastUpdateDay,
	sp.rows AS 'Rows',
	sp.modification_counter AS ModificationsSinceLastUpdate
FROM sys.stats AS s
JOIN sys.tables AS t
ON s.object_id = t.object_id
CROSS APPLY sys.dm_db_stats_properties(s.object_id, s.stats_id) AS sp
ORDER BY sp.modification_counter DESC;
```

Update 1 statistic:

```
UPDATE STATISTICS dbcustomers StatisticName
```

Or All:

```
EXEC sp_updatestats
```

Very expensive for large database!

MySQL:

MySQL allows you to view when statistics were last updated, similar to SQL Server, though the specific tables and columns differ.

For InnoDB tables in MySQL, you can query the mysql.innodb_table_stats and mysql.innodb_index_stats tables. These tables contain a last_update column that indicates the timestamp when the statistics for the respective table or index were last updated.

Here's an example of how to retrieve this information:

Code

```
SELECT
   table_name,
   last_update
FROM
   mysql.innodb_table_stats
WHERE
   database_name = 'your_database_name';
```

Replace 'your_database_name' with the actual name of your database. You can also filter by table_name or query mysql.innodb_index_stats for index-specific statistics update times.

Example:

```
mysql> SELECT
        table_name,
        last_update
    FROM
        mysql.innodb_table_stats
    WHERE
        database_name = 'salesdb';
```

table_name	last_update
Derp	2025-09-12 12:32:36
Derp3	2025-09-12 12:58:31
MonthlyOrders	2025-09-12 12:38:05
customers	2025-08-31 09:50:58
dbcustomers	2025-09-15 05:48:39
employees	2025-08-31 09:50:59
orders	2025-08-31 09:51:01
orders_archive	2025-08-31 09:51:12
products	2025-08-31 09:51:00
table_that_triggers	2025-09-12 14:43:15
timestampers	2025-09-12 14:43:44

Update:

```
mysql> ANALYZE TABLE employees;
```

Table	Op	Msg_type	Msg_text
salesdb.employees	analyze	status	OK

```
mysql> SELECT
                table_name,
                last_update
        FROM
                mysql.innodb_table_stats
        WHERE
                database_name = 'salesdb';
```

table_name	last_update	(Explanation)
Derp	2025-09-12 12:32:36	
Derp3	2025-09-12 12:58:31	
MonthlyOrders	2025-09-12 12:38:05	
customers	2025-08-31 09:50:58	
dbcustomers	2025-09-15 05:48:39	
employees	2025-09-15 06:43:57	<- Current now
orders	2025-08-31 09:51:01	
orders_archive	2025-08-31 09:51:12	
products	2025-08-31 09:51:00	
table_that_triggers	2025-09-12 14:43:15	
timestampers	2025-09-12 14:43:44	

Could use a script to loop through each table name and update each one.

Or could just do this:

"
The mysqlcheck utility is a command-line tool that can perform various maintenance operations, including analyzing tables.
To analyze all tables in a specific database:

Code

```
mysqlcheck -a your_database_name -u your_username -p
```
"

Updating Statistics

1. Weekly job to update statistice on weekends.
2. After migrating data.

5. Monitoring fragmentation

Fragmentation:

Unused spaces in data pages
Data pages are out of order

Fragmentation Methods

Reorganize:

Defragments leaf nodes to keep them sorted
"Light" operation

Rebuild:

Recreates index from scratch
"Heavy" operation

SQL Server:

SELECT * FROM sys.dm_db_index_physical_stats (DB_ID(), NULL, NULL, NULL, 'LIMITED')

avg_fragmentation_in_percent: Indicates how out-of-order pages are within the index.
0% means none, perfect.
100% completely out of order, worst.

JOIN with sys.tables and sys.indexes to get other information about indexes, such as table and index names.

When to defragment:

<10% No action needed

10-30% Reorganize

>30% Rebuild

Reorganize:

ALTER INDEX IndexName ON ColumnName REORGANIZE

Rebuild:

ALTER INDEX IndexName ON ColumnName REBUILD

Execution Plan

Roadmap generated by a database on how it will execute your query step by step.

How to get the data from the table
Which type of JOINs to be done
SELECT statement

Stores execution plan in the cache, so it can be used with same or similar queries.

User can view execution plan.

SQL Server:

Can get estimated execution plan, by clicking a button without running the query.
Or get actual execution plan, or live execution plan.

Estimated vs Actual Plans

If the predictions don't match the actual execution plan, this indicates issues like inaccurate statistics or outdated indexes, leading to poor performance.

After creating a new index, check if database is actually using new index.

Types of Scan:

Table Scan: Reads every row in a table.

Index Scan: Reads all entries in an index to find results.

Index Seek: Quickly locates specific rows in an index.

Join Algorithms (Not JOIN):

Nested Loops: Compares tables row by row, best for small tables.

Hash Match: Matches rows using a hash table, best for large tables.

Merge Join: Merge two sorted tables, efficient when both are sorted.

Purposes for viewing execution plan:

Understand how SQL executes your query.

How many resources does your query consume?

Check if your new indexes are actually being used.

Testing and experimenting with indexes.

SQL Hints:

Commands you add to a query to force the database to run it in a specific way for better performance.

```
SELECT ...
FROM ...
LEFT JOIN ...
ON ...
OPTION (HASH JOIN)
```

```
SELECT ...
FROM ...
LEFT JOIN ... WITH (FORCESEEK)
ON ...
```

```
SELECT ...
FROM ...
LEFT JOIN ... WITH (INDEX([Index_Name]))
ON ...
```

Tips:

1. Test hints in all project environments (DEV, PROD) as experience may vary.
2. Hints are quick fixes (Workaround not solution). You still have to find the cause and fix it.

Indexing Strategy

There isn't just one strategy that fits all projects.

Golden Rule: Avoid over indexing!
More is usually not better, actually ends up slowing everything down.
When data is inserted, updated, or deleted, database has to update indexes.
In general, too many slows performance and confuses execution plan.

Less is more!

A few effective indexes are better than many that don't work for the database.

MySQL:

"
The primary tool for this in MySQL is the EXPLAIN statement. When you prepend EXPLAIN to a SELECT, INSERT, UPDATE, DELETE, or REPLACE statement, MySQL will return information about how it intends to execute the query, rather than actually executing it and returning the result set.

EXPLAIN ANALYZE:
For a more in-depth analysis, EXPLAIN ANALYZE (available in newer MySQL versions) actually executes the query while planning, instrumenting, and measuring the time spent at various points in the execution plan. This provides the actual costs and measurements, offering a more accurate view of performance bottlenecks.
"

Example:

mysql> EXPLAIN ANALYZE SELECT * FROM cdb WHERE country = 'USA';

EXPLAIN

-> Filter: (cdb.country = 'USA') (cost=0.75 rows=1) (actual time=0.0711..0.0862 rows=3 loops=1)
-> Table scan on cdb (cost=0.75 rows=5) (actual time=0.0573..0.0777 rows=5 loops=1)

After creating index on flag:

mysql> EXPLAIN ANALYZE SELECT * FROM cdb WHERE is_usa = 1;

EXPLAIN

-> Index lookup on cdb using idx_dbcustomers_is_usa (is_usa=1) (cost=0.8 rows=3) (actual time=0.0496..0.0699 rows=3 loops=1)

Phase 1. Initial Indexing Strategy:

OLAP: Online Analytical Processing (Data Warehouse)

OLTP: Online Transaction Processing (Transaction, such as banking)

Either optimize read or write.

OLAP read, OLTP write.

OLAP: Columnstore index.

OLTP: Clustered index PK. Be more careful with adding indexes.

Phase 2. Usage Patterns Indexing:

1. Identify frequently used tables and columns.

2. Choose the right index.

3. Test the index.

Phase 3. Scenario-Based Indexing:

1. Identify slow queries.

2. Check execution plan.

3. Choose right index.

4. Test (compare) execution plans.

Phase 4. Monitoring & Maintenance:

1. Monitor index usage.

2. Monitor missing indexes.

3. Monitor duplicate indexes.

4. Update statistics.

5. Monitor fragmentations.

Partitions

Divides big table into smaller partitions while still being treated as a single,
logical table.

Reading a big table is slow, creating a big index tree could end up being slow, too.

Can partition the table into say 'old data' and 'new data'. Rarely need data from years
2023-2024, but maybe need data all the time from 2025.

If using parallel processing, can use partitions to process each partition at the same time,
improving performance.

Indexes can also be applied to separate partitions, making thier trees smaller and faster to use.

Partition Function

Define the logic on how to divide your data into partitions. Based on partition key (Column, Region,
Date)

Boundaries: Such as last day of the year.

Left or Right, which partition does the boundary belong to?

In this example, left.

Example (SQL Server):

```
CREATE PARTITION FUNCTION PartitionByYear (DATE) AS RANGE LEFT FOR VALUES
('2023-12-31', '2024-12-31', '2025-12-31')
```

List all partition functions in use:

```
SELECT
	name,
	function_id,
	type,
	type_desc,
	boundary_value_on_right
FROM sys.partition_functions
```

Should check before creating any new partition function.

Filegroups: Logical container of one or more data files to help organize partitions:

```
ALTER DATABASE salesdb ADD FILEGROUP FG_2023;
...
ALTER DATABASE salesdb ADD FILEGROUP FG_2026;
```

Instead of ADD, REMOVE to get rid of it.

Check them:

```
SELECT * FROM sys.filegroups WHERE type = 'FG'
```

Create data files:

```
ALTER DATABASE salesdb ADD FILE (
	NAME = P_2023, --Logical Name
	FILENAME = 'C:\COMPLETE_FILE_PATH_THAT_DATABASE_IS_IN\P_2023.ndf'
) TO FILEGROUP FG_2023;
...
ALTER DATABASE salesdb ADD FILE (
	NAME = P_2026, --Logical Name
	FILENAME = 'C:\COMPLETE_FILE_PATH_THAT_DATABASE_IS_IN\P_2026.ndf'
) TO FILEGROUP FG_2026;
```

Check the metadata:

```
SELECT
	fg.name AS FileGroupName,
	mf.name AS LogicalFileName,
	mf.physical_name AS PhysicalFilePath,
	mf.size / 128 AS SizeInMB
FROM
	sys.filegroups fg
JOIN
	sys.master_files mf ON fg.data_space_id = mf.data_space_id
WHERE
	mf.database_id = DB_ID('SalesDB');
```

Create partition schema:

```
CREATE PARTITION SCHEME SchemePartitionByYear
AS PARTITION PArtitionByYear
TO (FG_2023, ..., FG_2026) --Order is important! 3 boundaries, 4 partitions, 4 filegroups!
```

Use sys.partition_schemes to view all schemes.

Create the partitioned table (Can reuse for any table that fits):

```
CREATE TABLE Orders_Partitioned (
	orderid INT,
	orderdate DATE,
	sales INT
) ON SchemePartitionByYear (orderdate)
```

Insert data into the partitioned table:

```
INSERT INTO Orders_Partitioned VALUES (1, '2023-05-15', 100)
```

MySQL:

"
In MySQL, you do not directly create a "partition function" as a standalone object like in some other database systems (e.g., SQL Server). Instead, you define the partitioning strategy and rules as part of the CREATE TABLE statement when you create a new table, or by using ALTER TABLE to modify an existing table.

The "partition function" in MySQL is implicitly defined by the PARTITION BY clause and its associated expressions or column lists within the CREATE TABLE or ALTER TABLE statement.

Here's how you define partitioning in MySQL:

1. Choose a Partitioning Type:
MySQL supports several partitioning types:

RANGE Partitioning: Divides data based on ranges of values in a specific column (e.g., by year, month, or a numerical range).

LIST Partitioning: Divides data based on explicit lists of values in a specific column.

HASH Partitioning: Distributes data evenly across a specified number of partitions based on a hash function applied to a column.

KEY Partitioning: Similar to HASH, but uses MySQL's internal hashing function on the primary key or a unique key.

Code

```
CREATE TABLE sales (
    id INT NOT NULL,
    sale_date DATE NOT NULL,
    amount DECIMAL(10, 2)
)
PARTITION BY RANGE (YEAR(sale_date)) (
    PARTITION p0 VALUES LESS THAN (2020),
    PARTITION p1 VALUES LESS THAN (2022),
    PARTITION p2 VALUES LESS THAN (2024),
    PARTITION p3 VALUES LESS THAN (MAXVALUE)
);
```

2. Define Partitioning Rules:

Within the PARTITION BY clause, you specify the column(s) or expression(s) that determine how data is distributed, along with the specific rules for each partition (e.g., VALUES LESS THAN for RANGE, VALUES IN for LIST, or the number of PARTITIONS for HASH and KEY).

In summary: You don't create a separate "partition function" in MySQL. Instead, the partitioning logic is embedded directly within the CREATE TABLE or ALTER TABLE statements using the PARTITION BY clause and its associated options.

In MySQL, filegroups are not a concept used in conjunction with partitioning. The concept of filegroups is specific to SQL Server and other database systems, where they are used to manage the physical storage of data files and can be used to distribute partitions across different physical storage locations.

MySQL handles partitioning differently. When you partition a table in MySQL, the data for each partition is stored within the data files of the chosen storage engine (typically InnoDB).

While you can configure the data directory for the entire MySQL instance, or even for individual databases, there isn't a mechanism like SQL Server's filegroups to explicitly map individual partitions to distinct physical files or file system locations within MySQL.
"

Example:

```
ALTER TABLE orders PARTITION BY RANGE (YEAR(order_date)) (
        PARTITION p0 VALUES LESS THAN (2023),
    PARTITION p1 VALUES LESS THAN (2024),
    PARTITION p2 VALUES LESS THAN (2025),
    PARTITION p3 VALUES LESS THAN (2026)
)
```

Seems only to be able to be used with primary keys:

ERROR 1506 (HY000): Foreign keys are not yet supported in conjunction with partitioning

Performance Tips x30

For small-medium tables, the query optimizer may react similarly to different query styles. Might not notice any difference using these rules unless using larger tables.

Golden Rule: Always check the execution plan to confirm performance improvements when optimizing your query. If there's no improvement, then just focus on readability.

#1. Select only what you need.

#2. Avoid unnecessary DISTINCT and ORDER BY.

#3. For exploration purposes, limit rows!

#4. Create nonclustered index on frequently used columns in WHERE clause.

#5. Avoid applying functions to columns in WHERE clause, especially a waste when it avoids using an index.

#6. Avoid leading wildcards as they prevent index usage.

#7. Use IN instead of multiple OR.

#8. Understand the speed of joins and use INNER JOIN when possible. In order fastest to slowest: INNER, LEFT/RIGHT, OUTER.

#9. Use explicit JOIN (ANSI Join) instead of implicit JOIN (non-ANSI JOIN).

#10. Make sure to index the columns used in the ON clause.

#11. Filter before joining (Big Tables). During JOIN (ON ... AND), or before INNER JOIN (SELECT ...)

#12. Aggregate before JOINing (Big Tables), correlated subqueries are bad practice.

#13. Use UNION instead of OR in JOINs. OR is often a performance killer.

#14. Check for nested loops and use SQL Hints.

#15. Use UNION ALL instead of using UNION if duplicates are acceptable.

#16. Use UNION ALL + DISTINCT instead of using UNION if duplicates are not acceptable.

#17. Use columnstore index for aggregations on large tables.

#18. Pre-aggregate data and store it in a new table for reporting.

#19. JOIN is best practice if its performance will match EXISTS, but EXISTS is often better for larger tables.

#20. Avoid redundant logic in your query.

#21. Avoid Data Types VARCHAR & TEXT (If have to, VARCHAR > TEXT)

#22. Avoid MAX or unnecessarily large lengths in data type.

#23. Use the NOT NULL constraint where applicable.

#24. Ensure all your tables have a clustered primary key.

#25. Create a non-clustered index for foreign keys that are used frequently.

#26. Avoid over indexing, too many slows everything down.

#27. Drop unused indexes, they take up unnecessary space.

#28. Update statistics (Weekly).

#29. Reorganize & rebuild indexes (Weekly).

#30. Partition large tables (facts) to improve performance. Next, apply a columnstore index for the best results.

Extra: Focus on writing clear queries.
Optimize performance only when necessary.
Always test using execution plan.