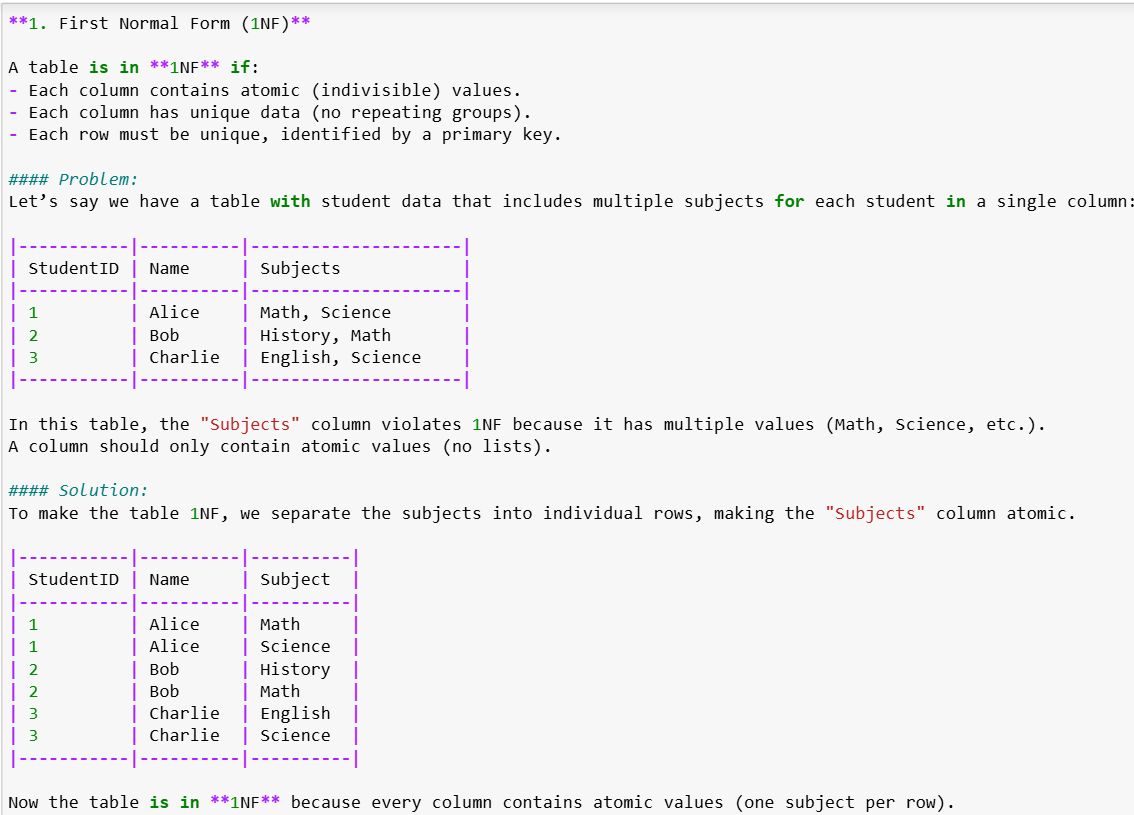
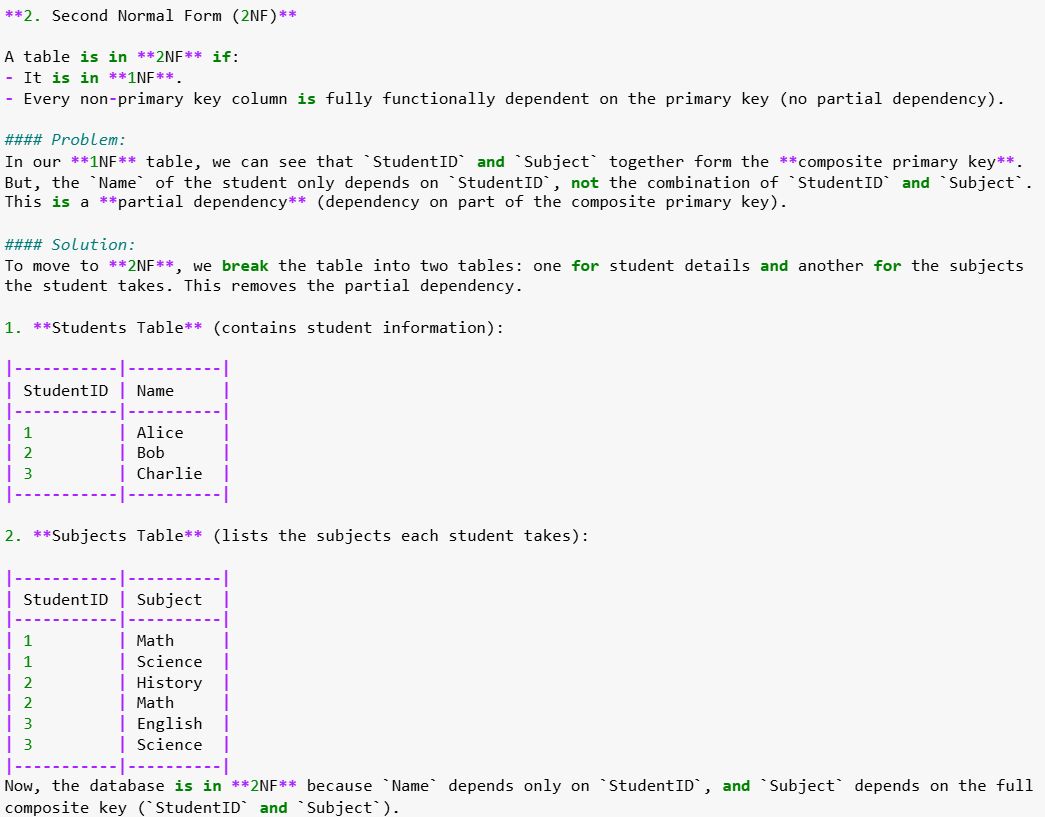
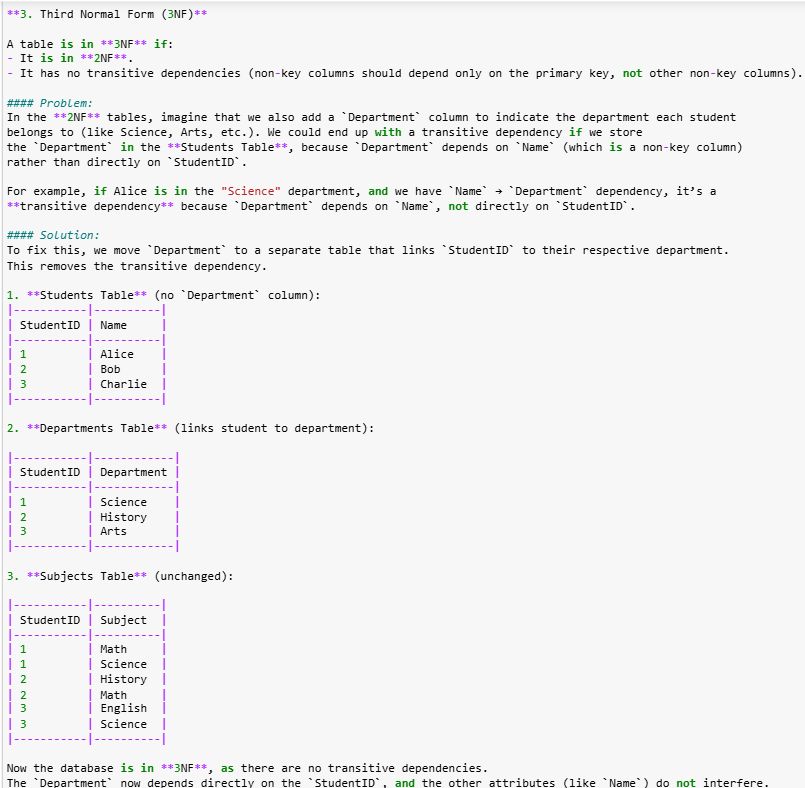
𝐖𝐡𝐚𝐭 𝐚𝐫𝐞 𝐭𝐡𝐞 𝐝𝐢𝐟𝐟𝐞𝐫𝐞𝐧𝐜𝐞𝐬 𝐛𝐞𝐭𝐰𝐞𝐞𝐧 𝐬𝐜𝐝𝟏 𝐯𝐬 𝐬𝐜𝐝𝟐 𝐯𝐬 𝐬𝐜𝐝𝟑 𝐢𝐧 𝐝𝐚𝐭𝐚 𝐦𝐨𝐝𝐞𝐥𝐥𝐢𝐧𝐠?  
  
𝟏. 𝐒𝐂𝐃𝟏 (𝐒𝐥𝐨𝐰𝐥𝐲 𝐂𝐡𝐚𝐧𝐠𝐢𝐧𝐠 𝐃𝐢𝐦𝐞𝐧𝐬𝐢𝐨𝐧 𝐓𝐲𝐩𝐞 𝟏):  
  
Purpose: Overwrite the existing record with the new value when a change occurs.  
  
Handling Changes: In SCD1, when there is an update or change in a dimension attribute, the old value is simply replaced with the new value.  
  
Historical Data: No history is maintained. Only the current value of the dimension is stored.  
  
Example: If a customer’s address changes, the old address is replaced by the new address, and no record of the old address is kept.  
  
𝐏𝐫𝐨𝐬:  
  
Simpler to implement.  
  
Less storage required since only the current data is kept.  
  
𝐂𝐨𝐧𝐬:  
  
No historical data is available for analysis.  
  
Cannot track the changes in the past.  
  
𝟐. 𝐒𝐂𝐃𝟐 (𝐒𝐥𝐨𝐰𝐥𝐲 𝐂𝐡𝐚𝐧𝐠𝐢𝐧𝐠 𝐃𝐢𝐦𝐞𝐧𝐬𝐢𝐨𝐧 𝐓𝐲𝐩𝐞 𝟐):  
  
Purpose: Maintain full history of changes by creating new records for each change.  
  
Handling Changes: When a change occurs, a new record is created with the new value, while the old record is preserved with an effective date (start and end date).  
  
Historical Data: Full history is maintained. The old and new values are both stored, along with the time period during which each record was valid.  
  
Example: If a customer’s address changes, a new record is created with the new address and a date range indicating the period when the old address was valid. The old record would still exist, but it will be marked as no longer valid (based on dates).  
  
𝐏𝐫𝐨𝐬:  
  
Full history is captured, which allows for better historical reporting and analysis.  
  
Ideal for situations where you need to track changes over time.  
  
𝐂𝐨𝐧𝐬:  
  
More complex to implement.  
  
Increased storage requirements due to multiple records for the same entity.  
Potential performance impact if the dataset grows significantly.  
  
𝟑. 𝐒𝐂𝐃𝟑 (𝐒𝐥𝐨𝐰𝐥𝐲 𝐂𝐡𝐚𝐧𝐠𝐢𝐧𝐠 𝐃𝐢𝐦𝐞𝐧𝐬𝐢𝐨𝐧 𝐓𝐲𝐩𝐞 𝟑):  
  
Purpose: Track only a limited history (usually just one previous value) by adding new columns for the changed attributes.  
  
Handling Changes: When a change occurs, a new column is added to store the previous value of the attribute. This way, both the current and the previous values are available.  
  
Historical Data: Only a limited history is kept — typically just the current and one previous value.  
  
Example: If a customer’s address changes, the current address would remain in the original column, and the previous address would be moved to a new column (e.g., "Previous Address").  
  
𝐏𝐫𝐨𝐬:  
  
Simpler than SCD2, as it only tracks limited history.  
  
Less storage required compared to SCD2.  
  
𝐂𝐨𝐧𝐬:  
  
Does not store full history, limiting its use in some scenarios.

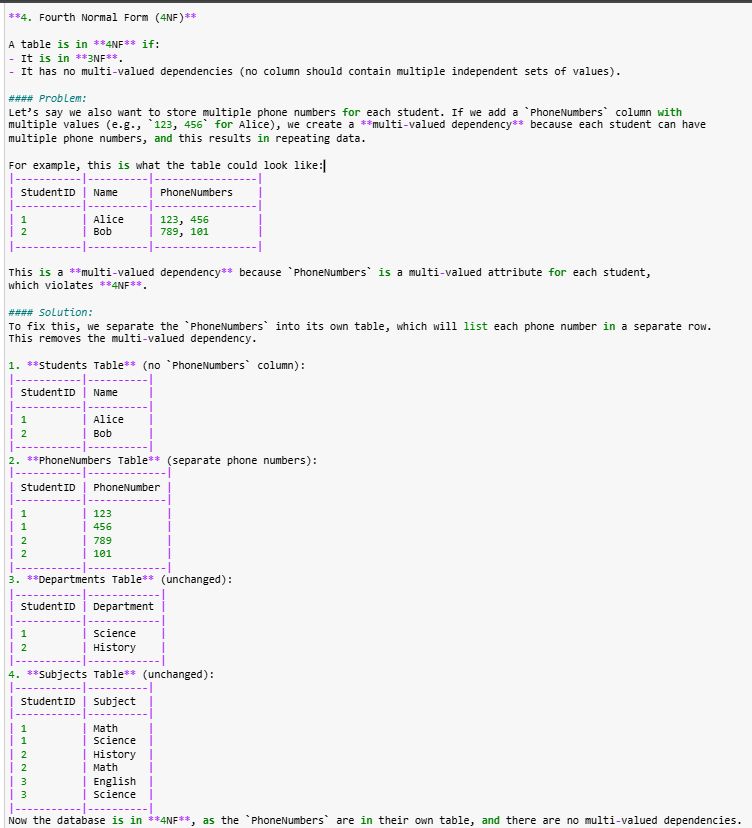
𝐖𝐡𝐚𝐭 𝐚𝐫𝐞 𝐭𝐡𝐞 𝐝𝐢𝐟𝐟𝐞𝐫𝐞𝐧𝐜𝐞𝐬 𝐛𝐞𝐭𝐰𝐞𝐞𝐧 𝟏𝐍𝐅, 𝟐𝐍𝐅, 𝟑𝐍𝐅, 𝐚𝐧𝐝 𝟒𝐍𝐅 𝐢𝐧 𝐝𝐚𝐭𝐚𝐛𝐚𝐬𝐞 𝐧𝐨𝐫𝐦𝐚𝐥𝐢𝐳𝐚𝐭𝐢𝐨𝐧?  
  
In database normalization, the terms 1NF (First Normal Form), 2NF (Second Normal Form), 3NF (Third Normal Form), and 4NF (Fourth Normal Form) refer to different stages of organizing data to minimize redundancy and ensure consistency. Each form builds on the previous one, addressing specific types of problems. Here's a breakdown with examples:  
  
𝟏. 𝐅𝐢𝐫𝐬𝐭 𝐍𝐨𝐫𝐦𝐚𝐥 𝐅𝐨𝐫𝐦 (𝟏𝐍𝐅):  
  
Each table has a primary key (a unique identifier for each row).  
  
Each column contains atomic (indivisible) values.  
  
Each entry in a column must be of the same type.  
  
𝟐. 𝐒𝐞𝐜𝐨𝐧𝐝 𝐍𝐨𝐫𝐦𝐚𝐥 𝐅𝐨𝐫𝐦 (𝟐𝐍𝐅):  
  
It is in 1NF.  
  
Every non-primary key column is fully functionally dependent on the primary key. This means there are no partial dependencies (i.e., no non-key columns depend only on part of a composite primary key).  
  
𝟑. 𝐓𝐡𝐢𝐫𝐝 𝐍𝐨𝐫𝐦𝐚𝐥 𝐅𝐨𝐫𝐦 (𝟑𝐍𝐅):  
  
It is in 2NF.  
  
There are no transitive dependencies. This means no non-primary key column depends on another non-primary key column.,  
  
𝟒. 𝐅𝐨𝐮𝐫𝐭𝐡 𝐍𝐨𝐫𝐦𝐚𝐥 𝐅𝐨𝐫𝐦 (𝟒𝐍𝐅):  
  
It is in 3NF.  
  
It has no multi-valued dependencies. A multi-valued dependency occurs when one attribute determines multiple independent attributes.

1NF: Ensure atomic values in every column (no multiple values in one column).  
  
2NF: Remove partial dependencies by ensuring that non-primary key columns depend fully on the primary key.  
  
3NF: Remove transitive dependencies by ensuring that non-primary key columns depend only on the primary key.  
  
4NF: Remove multi-valued dependencies by ensuring that each column contains only single-valued attributes.









𝐖𝐡𝐚𝐭 𝐢𝐬 𝐭𝐡𝐞 𝐨𝐮𝐭𝐩𝐮𝐭 𝐰𝐡𝐞𝐧 𝐩𝐞𝐫𝐟𝐨𝐫𝐦𝐢𝐧𝐠 𝐈𝐧𝐧𝐞𝐫 𝐉𝐨𝐢𝐧, 𝐋𝐞𝐟𝐭 𝐉𝐨𝐢𝐧, 𝐑𝐢𝐠𝐡𝐭 𝐉𝐨𝐢𝐧, 𝐚𝐧𝐝 𝐂𝐫𝐨𝐬𝐬 𝐉𝐨𝐢𝐧 𝐨𝐧 𝐭𝐡𝐞 𝐟𝐨𝐥𝐥𝐨𝐰𝐢𝐧𝐠 𝐭𝐚𝐛𝐥𝐞𝐬 𝐜𝐨𝐧𝐭𝐚𝐢𝐧𝐢𝐧𝐠 𝐍𝐔𝐋𝐋 𝐯𝐚𝐥𝐮𝐞𝐬?  
  
-- 𝐒𝐭𝐫𝐮𝐜𝐭𝐮𝐫𝐞 𝐟𝐨𝐫 𝐭𝐚𝐛𝐥𝐞𝟏  
  
CREATE TABLE table1 (  
 id INT  
);  
  
  
-- 𝐈𝐧𝐬𝐞𝐫𝐭 𝐝𝐚𝐭𝐚 𝐢𝐧𝐭𝐨 𝐭𝐚𝐛𝐥𝐞𝟏  
  
INSERT INTO table1 (id) VALUES  
(NULL),  
(NULL);  
  
-- 𝐒𝐭𝐫𝐮𝐜𝐭𝐮𝐫𝐞 𝐟𝐨𝐫 𝐭𝐚𝐛𝐥𝐞𝟐  
  
CREATE TABLE table2 (  
 id INT  
);  
  
-- 𝐈𝐧𝐬𝐞𝐫𝐭 𝐝𝐚𝐭𝐚 𝐢𝐧𝐭𝐨 𝐭𝐚𝐛𝐥𝐞𝟐  
  
INSERT INTO table2 (id) VALUES  
(NULL),  
(NULL),  
(NULL);  
  
  
-- 𝐃𝐢𝐬𝐩𝐥𝐚𝐲 𝐭𝐚𝐛𝐥𝐞𝟏  
  
SELECT \* FROM table1;  
  
-- 𝐃𝐢𝐬𝐩𝐥𝐚𝐲 𝐭𝐚𝐛𝐥𝐞𝟐  
  
SELECT \* FROM table2;  
  
-- 𝐈𝐧𝐧𝐞𝐫 𝐉𝐨𝐢𝐧 (𝐈𝐍𝐍𝐄𝐑 𝐉𝐎𝐈𝐍)  
  
SELECT [table1.id](http://table1.id/)

AS table1\_id, [table2.id](http://table2.id/)

AS table2\_id  
FROM table1  
INNER JOIN table2 ON [table1.id](http://table1.id/)

= [table2.id](http://table2.id/)

;  
  
-- 𝐋𝐞𝐟𝐭 𝐉𝐨𝐢𝐧 (𝐋𝐄𝐅𝐓 𝐎𝐔𝐓𝐄𝐑 𝐉𝐎𝐈𝐍)  
  
SELECT [table1.id](http://table1.id/)

AS table1\_id, [table2.id](http://table2.id/)

AS table2\_id  
FROM table1  
LEFT JOIN table2 ON [table1.id](http://table1.id/)

= [table2.id](http://table2.id/)

;  
  
-- 𝐑𝐢𝐠𝐡𝐭 𝐉𝐨𝐢𝐧 (𝐑𝐈𝐆𝐇𝐓 𝐎𝐔𝐓𝐄𝐑 𝐉𝐎𝐈𝐍)  
  
SELECT [table1.id](http://table1.id/)

AS table1\_id, [table2.id](http://table2.id/)

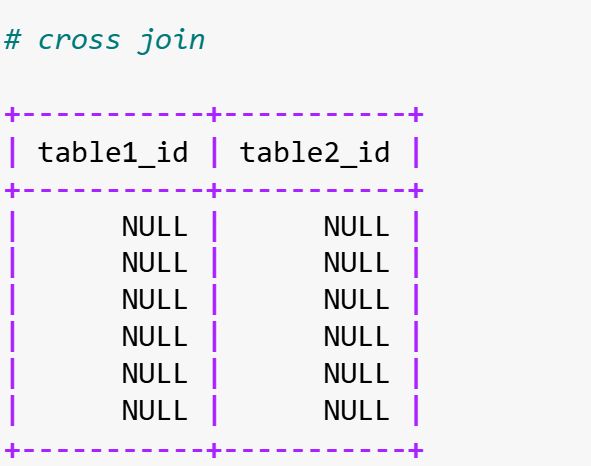
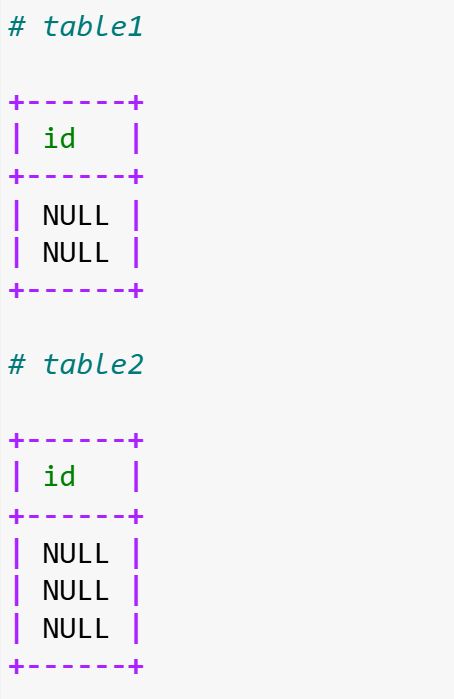
AS table2\_id  
FROM table1  
RIGHT JOIN table2 ON [table1.id](http://table1.id/)

= [table2.id](http://table2.id/)

;  
  
-- 𝐂𝐫𝐨𝐬𝐬 𝐉𝐨𝐢𝐧 (𝐂𝐑𝐎𝐒𝐒 𝐉𝐎𝐈𝐍)  
  
SELECT [table1.id](http://table1.id/)

AS table1\_id, [table2.id](http://table2.id/)

AS table2\_id  
FROM table1  
CROSS JOIN table2;



𝐂𝐎𝐔𝐍𝐓(\*) 𝐯𝐬 𝐂𝐎𝐔𝐍𝐓(𝟏) 𝐢𝐧 𝐒𝐐𝐋?  
  
𝟏. 𝐂𝐎𝐔𝐍𝐓(\*)  
  
COUNT(\*) counts all rows in a table, including rows with NULL values in any column. It does not consider the actual contents of the rows, just the number of rows.  
  
How it works: SQL engines will count every row without worrying about any specific column’s value, so it's typically the most efficient way to count all rows in a table.  
  
𝐄𝐱𝐚𝐦𝐩𝐥𝐞:  
  
SELECT COUNT(\*) FROM employees;  
  
This counts all rows in the employees table, regardless of any NULL values.  
  
𝟐. 𝐂𝐎𝐔𝐍𝐓(𝟏)  
  
COUNT(1) counts the number of rows, but instead of counting all rows indiscriminately, it counts the number of times the expression (in this case, 1) is not NULL. Since 1 is a constant and never NULL, this effectively counts all the rows, similar to COUNT(\*).  
  
How it works: COUNT(1) evaluates the expression for every row. However, the 1 is not a column or an expression that could be NULL, so it counts all rows just like COUNT(\*).  
  
𝐄𝐱𝐚𝐦𝐩𝐥𝐞:  
  
SELECT COUNT(1) FROM employees;  
  
This also counts all rows in the employees table, as 1 is never NULL.  
  
𝐊𝐞𝐲 𝐃𝐢𝐟𝐟𝐞𝐫𝐞𝐧𝐜𝐞𝐬:  
  
𝐏𝐞𝐫𝐟𝐨𝐫𝐦𝐚𝐧𝐜𝐞:  
  
In most modern SQL databases, there is no significant performance difference between COUNT(\*) and COUNT(1) because both typically involve counting all rows, and the database engine optimizes both queries similarly.  
In some rare cases, older database systems might optimize COUNT(\*) better than COUNT(1), but this difference is negligible on modern systems.  
  
𝐂𝐨𝐧𝐜𝐞𝐩𝐭𝐮𝐚𝐥 𝐃𝐢𝐟𝐟𝐞𝐫𝐞𝐧𝐜𝐞:  
  
COUNT(\*) counts all rows, regardless of column values.  
  
COUNT(1) counts rows where the expression is not NULL, but since 1 is a constant, it behaves like COUNT(\*).  
  
In practice, use COUNT(\*) if you simply want to count all rows in a table.  
COUNT(1) can be used, but it is semantically the same as COUNT(\*) in most cases, so there’s no real benefit in choosing one over the other in most modern SQL engines. However, it’s best to stick with COUNT(\*) for clarity.

𝟏. 𝐖𝐡𝐚𝐭 𝐢𝐬 𝐇𝐃𝐅𝐒, 𝐚𝐧𝐝 𝐡𝐨𝐰 𝐝𝐨𝐞𝐬 𝐢𝐭 𝐝𝐢𝐟𝐟𝐞𝐫 𝐟𝐫𝐨𝐦 𝐚 𝐭𝐫𝐚𝐝𝐢𝐭𝐢𝐨𝐧𝐚𝐥 𝐟𝐢𝐥𝐞 𝐬𝐲𝐬𝐭𝐞𝐦?  
  
HDFS (Hadoop Distributed File System) is designed to store large datasets across multiple machines in a Hadoop cluster. It splits large files into fixed-size blocks and distributes them across nodes for parallel processing. Key features include:  
  
Fault Tolerance: HDFS replicates blocks (usually 3 copies) to ensure data availability in case of node failure.  
  
Scalability: Can scale horizontally by adding more nodes to the cluster.  
  
Optimized for Large Files: Best suited for sequential access of large files.  
Traditional File Systems (e.g., NTFS, ext4) store data on a single machine and are designed for random access to small files. They typically don't provide the same level of fault tolerance, scalability, or parallel processing optimization as HDFS.  
  
𝟐. 𝐇𝐨𝐰 𝐰𝐨𝐮𝐥𝐝 𝐲𝐨𝐮 𝐨𝐩𝐭𝐢𝐦𝐢𝐳𝐞 𝐚 𝐒𝐐𝐋 𝐪𝐮𝐞𝐫𝐲 𝐭𝐡𝐚𝐭 𝐢𝐬 𝐫𝐮𝐧𝐧𝐢𝐧𝐠 𝐬𝐥𝐨𝐰𝐥𝐲?  
  
To optimize a slow SQL query:  
Indexing: Ensure relevant columns in WHERE, JOIN, ORDER BY, and GROUP BY clauses are indexed.  
  
Query Refactoring: Avoid SELECT \*, only retrieve necessary columns.  
Examine Execution Plan: Use EXPLAIN to analyze query performance and detect bottlenecks.  
  
Optimize Joins: Ensure joins are done on indexed columns and use appropriate join types.  
  
Limit Data: Use LIMIT or pagination to reduce the number of rows processed.  
Avoid Functions on Indexed Columns: Functions on columns in WHERE can prevent the use of indexes.  
  
Database Tuning: Optimize database settings like buffer pool size and query cache.  
  
𝟑. 𝐄𝐱𝐩𝐥𝐚𝐢𝐧 𝐭𝐡𝐞 𝐂𝐀𝐏 𝐭𝐡𝐞𝐨𝐫𝐞𝐦 𝐚𝐧𝐝 𝐡𝐨𝐰 𝐢𝐭 𝐚𝐩𝐩𝐥𝐢𝐞𝐬 𝐭𝐨 𝐝𝐢𝐬𝐭𝐫𝐢𝐛𝐮𝐭𝐞𝐝 𝐬𝐲𝐬𝐭𝐞𝐦𝐬?  
  
The CAP Theorem states that in a distributed system, you can only achieve two out of three guarantees:  
  
Consistency: Every read returns the latest written value.  
Availability: Every request (read/write) receives a response, even if some nodes are down.  
  
Partition Tolerance: The system continues to function even if there’s a network partition.  
  
𝐓𝐡𝐞 𝐭𝐡𝐫𝐞𝐞 𝐩𝐨𝐬𝐬𝐢𝐛𝐥𝐞 𝐬𝐲𝐬𝐭𝐞𝐦 𝐭𝐲𝐩𝐞𝐬 𝐚𝐫𝐞:  
𝐂𝐏 (𝐂𝐨𝐧𝐬𝐢𝐬𝐭𝐞𝐧𝐜𝐲 𝐚𝐧𝐝 𝐏𝐚𝐫𝐭𝐢𝐭𝐢𝐨𝐧 𝐓𝐨𝐥𝐞𝐫𝐚𝐧𝐜𝐞): Ensures data consistency, but may sacrifice availability during partitions (e.g., HBase).  
  
𝐀𝐏 (𝐀𝐯𝐚𝐢𝐥𝐚𝐛𝐢𝐥𝐢𝐭𝐲 𝐚𝐧𝐝 𝐏𝐚𝐫𝐭𝐢𝐭𝐢𝐨𝐧 𝐓𝐨𝐥𝐞𝐫𝐚𝐧𝐜𝐞): Ensures the system remains available, even during network partitions, but may result in inconsistent data (e.g., CouchDB, Cassandra).  
  
𝐂𝐀 (𝐂𝐨𝐧𝐬𝐢𝐬𝐭𝐞𝐧𝐜𝐲 𝐚𝐧𝐝 𝐀𝐯𝐚𝐢𝐥𝐚𝐛𝐢𝐥𝐢𝐭𝐲): Guarantees data consistency and availability but doesn’t handle network partitions well (rare in distributed systems).

𝐖𝐡𝐚𝐭 𝐢𝐬 𝐭𝐡𝐞 𝐝𝐢𝐟𝐟𝐞𝐫𝐞𝐧𝐜𝐞 𝐛𝐞𝐭𝐰𝐞𝐞𝐧 𝐚 𝐫𝐞𝐥𝐚𝐭𝐢𝐨𝐧𝐚𝐥 𝐚𝐧𝐝 𝐚 𝐍𝐨𝐒𝐐𝐋 𝐝𝐚𝐭𝐚𝐛𝐚𝐬𝐞? 𝐖𝐡𝐞𝐧 𝐰𝐨𝐮𝐥𝐝 𝐲𝐨𝐮 𝐮𝐬𝐞 𝐞𝐚𝐜𝐡?  
  
𝟏. 𝐃𝐚𝐭𝐚 𝐒𝐭𝐫𝐮𝐜𝐭𝐮𝐫𝐞:  
𝐑𝐞𝐥𝐚𝐭𝐢𝐨𝐧𝐚𝐥 𝐃𝐚𝐭𝐚𝐛𝐚𝐬𝐞  
Uses tables (rows and columns) to organize data. Data is structured and follows a strict schema (fixed columns and data types).  
Examples: MySQL, PostgreSQL, Oracle, SQL Server.  
  
𝐍𝐨𝐒𝐐𝐋 𝐃𝐚𝐭𝐚𝐛𝐚𝐬𝐞:  
Data is stored in flexible, schema-less formats such as key-value pairs, documents, wide-column stores, or graphs.  
Examples: MongoDB, Cassandra, Redis, CouchDB, Neo4j.  
  
𝟐. 𝐒𝐜𝐡𝐞𝐦𝐚:  
𝐑𝐞𝐥𝐚𝐭𝐢𝐨𝐧𝐚𝐥 𝐃𝐚𝐭𝐚𝐛𝐚𝐬𝐞  
Requires a predefined schema before storing data. Tables and columns are defined, and data must conform to this structure.  
  
𝐍𝐨𝐒𝐐𝐋 𝐃𝐚𝐭𝐚𝐛𝐚𝐬𝐞:  
Schema is flexible or schema-less. It allows storing different types of data, including unstructured or semi-structured data (e.g., JSON, XML).  
  
𝟑. 𝐒𝐜𝐚𝐥𝐚𝐛𝐢𝐥𝐢𝐭𝐲:  
𝐑𝐞𝐥𝐚𝐭𝐢𝐨𝐧𝐚𝐥 𝐃𝐚𝐭𝐚𝐛𝐚𝐬𝐞  
Vertical scaling (scaling by adding more power to a single server, like CPU, RAM, etc.).  
Scaling may be limited as the database grows, especially with very large datasets.  
  
𝐍𝐨𝐒𝐐𝐋 𝐃𝐚𝐭𝐚𝐛𝐚𝐬𝐞:  
Horizontal scaling (scaling by adding more servers to distribute the load).  
Designed for high availability and scalability, often used in distributed environments.  
  
𝟒. 𝐓𝐫𝐚𝐧𝐬𝐚𝐜𝐭𝐢𝐨𝐧𝐬:  
𝐑𝐞𝐥𝐚𝐭𝐢𝐨𝐧𝐚𝐥 𝐃𝐚𝐭𝐚𝐛𝐚𝐬𝐞  
Supports ACID properties (Atomicity, Consistency, Isolation, Durability) for transactions, ensuring strong consistency and reliability.  
  
𝐍𝐨𝐒𝐐𝐋 𝐃𝐚𝐭𝐚𝐛𝐚𝐬𝐞:  
Many NoSQL databases do not fully support ACID properties but offer eventual consistency for scalability (BASE properties - Basically Available, Soft state, Eventually consistent).  
  
𝐖𝐡𝐞𝐧 𝐭𝐨 𝐔𝐬𝐞 𝐄𝐚𝐜𝐡:  
  
𝐔𝐬𝐞 𝐑𝐞𝐥𝐚𝐭𝐢𝐨𝐧𝐚𝐥 𝐃𝐚𝐭𝐚𝐛𝐚𝐬𝐞𝐬 𝐖𝐡𝐞𝐧:  
Data is Structured: When you have structured data with well-defined relationships (e.g., customer data, transactions).  
  
Complex Queries: If your application needs to perform complex queries, joins, and aggregations.  
  
Transactional Integrity: When your application requires full ACID compliance to ensure transaction consistency (e.g., e-commerce transactions, payroll systems).  
  
𝐔𝐬𝐞 𝐍𝐨𝐒𝐐𝐋 𝐃𝐚𝐭𝐚𝐛𝐚𝐬𝐞𝐬 𝐖𝐡𝐞𝐧:  
Scalability: If you need horizontal scaling to handle large amounts of data or high traffic loads (e.g., social media platforms, web analytics).  
  
Unstructured or Semi-Structured Data: If you need to store unstructured data like JSON, documents, or key-value pairs (e.g., logs, user profiles).  
  
Flexibility in Schema: If your data model is evolving, or you do not know the exact structure of your data in advance (e.g., content management, IoT devices).

𝐈𝐧𝐭𝐞𝐫𝐯𝐢𝐞𝐰 𝐏𝐫𝐨𝐜𝐞𝐝𝐮𝐫𝐞  
  
Round 1 (L1)   
Round 2 (L2)  
Managerial Round  
  
𝐑𝐨𝐮𝐧𝐝 𝟏 (𝐋𝟏)   
  
1, What is the output of INNER JOIN, LEFT JOIN, and RIGHT JOIN, and how many rows will they produce?  
  
2, What is the difference between RANK(), DENSE\_RANK(), and ROW\_NUMBER()?  
  
3, Explain how Spark works internally when a job is submitted?  
  
4, Explain the difference between Spark and MapReduce?  
  
5, Write a query to find the second highest salary of an employee?  
  
6, Explain how Spark and Hadoop handle large datasets?  
  
7, What are the key differences between Spark and Hadoop?  
  
9, What is the role of SparkContext in Spark?  
  
10, How do you optimize Spark jobs for better performance?  
  
  
𝐑𝐨𝐮𝐧𝐝 𝟐 (𝐋𝟐)  
  
1, Explain your previous project experience, the technologies you’ve worked with, and any challenges you faced?  
  
2, Write a query to list customers who spent more than ₹1000 on their orders in the last month?  
  
3, Write a query to find customers who spent the most on their orders in the last month?  
  
4, Write a query to join two dataframes with different schemas, where the left table has more rows?  
  
6, Explain how you handled a specific challenge in your previous project, particularly related to large-scale data processing?  
  
7, Explain the use of window functions in SQL and how they were applied in your project?  
  
9, How do you handle missing or corrupted data in large datasets?  
  
10, How do you ensure data quality and consistency in a distributed system?  
  
11, Describe the process of data ingestion in a data pipeline using Spark?  
  
12, Explain the differences between batch processing and real-time data processing?  
  
13, What strategies do you use to handle skewed data in Spark jobs?  
  
14, How do you ensure the scalability of your data processing system?  
  
15, What is the role of a DataFrame in Spark, and how is it different from an RDD?  
  
16, How do you design and implement a data pipeline for incremental data processing?  
  
17, Explain how you would use a window function to calculate running totals or rank data in Spark?  
  
18, How do you monitor and debug data pipeline jobs in Spark?  
  
19, What is the importance of partitioning in Spark, and how do you decide the partitioning strategy for a job?  
  
𝐌𝐚𝐧𝐚𝐠𝐞𝐫𝐢𝐚𝐥 𝐑𝐨𝐮𝐧𝐝  
  
1, Can you explain your previous project in detail? What were the objectives, the technologies you used, and the challenges you faced?  
  
2, How do you prioritize tasks and manage deadlines in a fast-paced environment, especially when working on multiple data projects simultaneously?  
  
3, Can you describe a situation where you had to mentor or guide a junior team member? How did you handle it, and what was the outcome?  
  
4, How do you handle conflicts within the team, especially when it comes to disagreements about data approaches or methodologies? Can you share an example of how you resolved such a situation?