Author: Muhammad Jahanzaib

Interview Preparation &

Essential Concepts for

Introduction to Databases

Relational Databases - Topics

Instructions for Learning and Interview Preparation

- ➢ This presentation aims to provide a comprehensive understanding of Relational Database concepts to help you in your study and interview preparation.
- ➢ Each topic will have a brief introduction, followed by examples and practice questions to enhance your learning.
- ➢ Take your time to understand each concept thoroughly and practice coding examples to improve your skills.
- ➢ Pay attention to the common interview questions related to each topic, as they can help you prepare for technical interviews.
- ➢ Don't hesitate to ask questions or seek help from your peers or instructors if you face any difficulty understanding a concept.
- ➢ We hope that this presentation will help you build a strong foundation in relational database concepts and prepare you for your future endeavors.

Remember, databases are vast and complex topics that take time and practice to fully understand. Don't get discouraged if you encounter difficulties. Instead, keep practicing and seek help when needed.
Good luck!

Author: Muhammad Jahanzaib

Introduction to Databases

Definition:

A database is a collection of data that is organized in a way that allows easy access, retrieval, and management of data. It can be stored and managed using computer software and is designed to meet the specific needs of an organization or application.

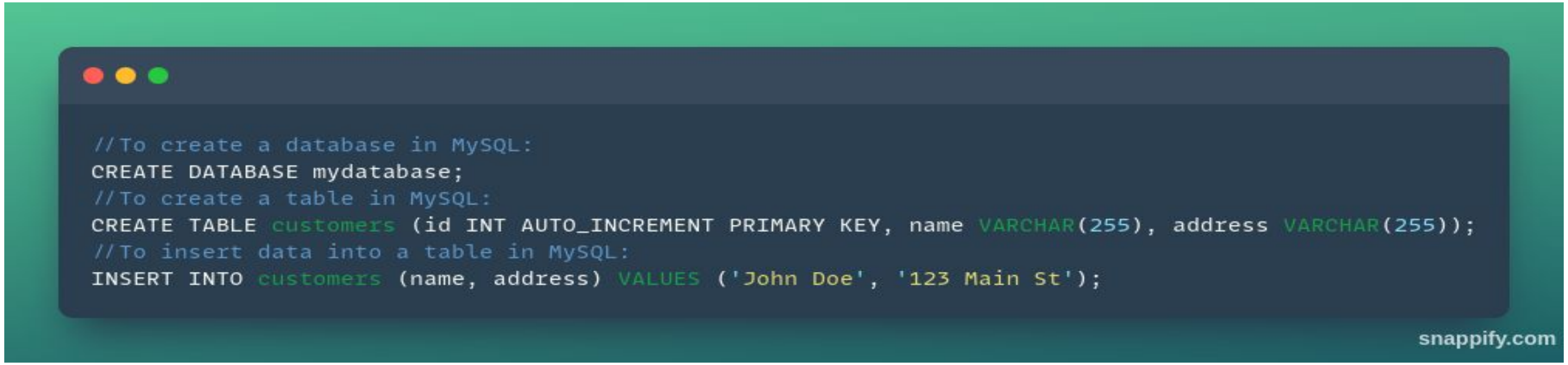
Key Points:

- Databases are used to store and manage large amounts of data efficiently and securely.
- Databases can be relational or non-relational, depending on how they organize data.
- Relational databases are the most common type of database and use tables to store data.
- Non-relational databases, such as document stores and key-value stores, use different data models to organize data.

Introduction to Databases

Examples:

- A customer database that stores information about customers such as their name, address, and purchase history.
- A healthcare database that stores patient information such as medical history and test results.
- A social media platform's database that stores user information, posts, and interactions.



```
//To create a database in MySQL:
CREATE DATABASE mydatabase;
//To create a table in MySQL:
CREATE TABLE customers (id INT AUTO_INCREMENT PRIMARY KEY, name VARCHAR(255), address VARCHAR(255));
//To insert data into a table in MySQL:
INSERT INTO customers (name, address) VALUES ('John Doe', '123 Main St');
```

Author: Muhammad Jahanzaib

Introduction to Databases

Best Practices::

- Normalization: Design databases with normalized tables to minimize data redundancy and improve performance.
- Security: Implement access controls, encryption, and backups to protect data from unauthorized access and data loss.
- Performance: Optimize database performance by using indexes, caching, and query optimization techniques.

Author: Muhammad Jahanzaib

Relational *vs* Non-relational databases

Feature	Relational Databases	Non-Relational Databases
Structure	Data is stored in tables with rows and columns	Data is stored in documents, key-value pairs, or graph structures
Scalability	Vertical scaling is common, but can also scale horizontally	Designed for horizontal scaling
Data Integrity	Enforce referential integrity through constraints	No enforced data integrity
Query Language	SQL (Structured Query Language)	Language specific to database
ACID Compliance	ACID (Atomicity, Consistency, Isolation, Durability) compliant	Not necessarily ACID compliant
Data Normalization	Follows normalization rules to reduce redundancy	Does not necessarily follow normalization rules
Use cases	Best suited for structured data with well-defined relationships between data elements	Best suited for unstructured data or semi-structured data, such as log files, social media data, or documents
Examples	MySQL. PostgreSQL. Oracle Database. Microsoft SQL Server. IBM Db2. MariaDB.	MongoDB, Apache Cassandra, Redis, Couchbase and Apache HBase.

Author: Muhammad Jahanzaib

Relational Databases

Definition:

Relational databases are databases that organize data into one or more tables, with each table consisting of columns and rows. The relationships between tables are established through primary keys and foreign keys.

Key points:

- Tables are the fundamental structure of a relational database and store data in a structured format.
- Rows represent a single record, and columns represent attributes of that record.
- Primary keys are unique identifiers for each row in a table, while foreign keys establish relationships between tables.
- Joins allow data to be combined from multiple tables.
- Indexes are used to improve query performance by creating a data structure that allows for faster data retrieval.
- SQL is the language used to interact with relational databases.
- CRUD operations (Create, Read, Update, Delete) are the basic operations used to manipulate data in a relational database.

Author: Muhammad Jahanzaib

Introduction to Databases

Examples:

- A customer database that stores customer information in one table and order information in another, with a foreign key linking the two tables.
- An employee database that stores employee information in one table and department information in another, with a foreign key linking the two tables.
- A product database that stores product information in one table and inventory information in another, with a foreign key linking the two tables.

Best Practices::

- Use consistent naming conventions for tables, columns, and keys.
- Use appropriate data types for columns to optimize performance and prevent data errors.
- Avoid using too many indexes, as this can degrade performance.
- Use parameterized queries to prevent SQL injection attacks.

Normalization

Definition:

Normalization is the process of organizing data in a database in a way that minimizes redundancy and dependency. This helps improve data consistency, integrity, and efficiency.

Key points:

- First Normal Form (1NF) requires that each table have a primary key and that each column in the table be atomic (i.e., not contain multiple values).
- Second Normal Form (2NF) requires that each non-key column in the table be fully dependent on the primary key.
- Third Normal Form (3NF) requires that each non-key column in the table be independent of other non-key columns.

Normalization

Examples:

- A customer table in a retail database that includes a customer's name, address, and order history. Normalization would involve breaking this table into multiple tables to eliminate redundancy and dependency.
- A product table in an inventory database that includes a product's name, price, and supplier information. Normalization would involve separating supplier information into its own table to eliminate redundancy and improve data consistency.

Best Practices:

- Follow the normal forms as closely as possible to minimize redundancy and dependency.
- Use foreign keys and relationships to ensure data consistency and integrity.
- Regularly monitor and optimize database performance.

Normalization - First Normal Form (1NF)

Definition:

First Normal Form (1NF) is a database normalization rule that requires each table to have a primary key and that each column in the table be atomic (i.e., not contain multiple values).

Key points:

- Each row in the table must have a unique identifier, or primary key.
- Each column in the table must contain only atomic values (i.e., no repeating groups or arrays).
- Each column in the table must have a unique name.

Normalization - First Normal Form (1NF)

Examples:

- A customer table in a retail database that includes a customer's name, address, and order history. The table would be in 1NF if each row has a unique identifier and each column contains only a single value (e.g., the address column does not include street, city, state, and zip code all in one).
- A product table in an inventory database that includes a product's name, price, and supplier information. The table would be in 1NF if each row has a unique identifier and each column contains only a single value (e.g., the supplier column does not include multiple supplier names).

Best Practices::

- Always include a primary key in each table to ensure each row can be uniquely identified.
- Avoid repeating groups or arrays in table columns, as they violate the atomicity requirement of 1NF.
- Use consistent naming conventions for columns to ensure uniqueness.

Author: Muhammad Jahanzaib

Normalization - First Normal Form (1NF) - Interview Questions

1. What is First Normal Form (1NF), and why is it important?
2. What are the requirements for a table to satisfy 1NF?
3. Can you provide an example of a table that violates 1NF, and explain how you would modify it to satisfy the rule?
4. What are some potential drawbacks of enforcing 1NF?

Author: Muhammad Jahanzaib

Normalization - Second Normal Form (2NF)

Definition:

Second Normal Form (2NF) is a database normalization rule that requires each non-key column in a table to be dependent on the entire primary key, not just a part of it.

Key points:

- A table must satisfy 1NF before it can satisfy 2NF.
- A table must have a composite primary key that includes multiple columns.
- Each non-key column in the table must be functionally dependent on the entire composite primary key.

Author: Muhammad Jahanzaib

Normalization - Second Normal Form (2NF)

Examples:

- A customer order table in a retail database that includes columns for order ID, customer ID, product ID, quantity, and price. The table would be in 2NF if each non-key column (i.e., quantity and price) is dependent on the entire composite primary key (i.e., order ID and product ID).
- A student course enrollment table in a university database that includes columns for student ID, course ID, course name, and course instructor. The table would be in 2NF if the course name and instructor columns are dependent on the entire composite primary key (i.e., student ID and course ID).

Best Practices::

- Use a composite primary key for tables with multiple related columns.
- Ensure each non-key column is dependent on the entire primary key to avoid partial dependencies.

Author: Muhammad Jahanzaib

Normalization - Second Normal Form (2NF) - Interview Questions

1. What is Second Normal Form (2NF), and why is it important?
2. Can you provide an example of a table that violates 2NF, and explain how you would modify it to satisfy the rule?
3. How is 2NF different from 1NF?
4. What are some potential drawbacks of enforcing 2NF?

Normalization - Third Normal Form (3NF)

Definition:

Third Normal Form (3NF) is a database normalization rule that requires all non-key columns in a table to be dependent only on the primary key or other non-key columns, but not on each other.

Key points:

- A table must satisfy 2NF before it can satisfy 3NF.
- Each non-key column in the table must be dependent only on the primary key or other non-key columns.
- If a non-key column is dependent on another non-key column, it should be separated into a separate table.

Normalization - Third Normal Form (3NF)

Examples:

- A product sales table in a retail database that includes columns for order ID, product ID, customer ID, product name, price, and quantity. The table would be in 3NF if the product name and price columns are moved to a separate product table, and the product ID is used as a foreign key to link the two tables.
- A student course enrollment table in a university database that includes columns for student ID, course ID, course name, instructor name, and department. The table would be in 3NF if the department column is moved to a separate department table, and the department ID is used as a foreign key to link the two tables.

Best Practices::

- Normalize tables to 3NF to reduce data redundancy and improve data consistency.
- Use foreign keys to establish relationships between tables.

Author: Muhammad Jahanzaib

Normalization - Third Normal Form (3NF) - Interview Questions

1. What is Third Normal Form (3NF), and why is it important?
2. Can you provide an example of a table that violates 3NF, and explain how you would modify it to satisfy the rule?
3. How is 3NF different from 2NF?
4. What are some potential drawbacks of enforcing 3NF?
5. What is the difference between 1NF and 2NF? Can you provide an example to illustrate this difference?
6. Explain the concept of transitive dependency and how it can be eliminated through normalization.
7. Can you explain the difference between functional dependencies and multi-valued dependencies in the context of normalization?
8. What is the purpose of normalization? Is it always necessary to normalize a database to 3NF?
9. In what situations would you consider breaking normalization rules? Can you provide an example?

Joins

Definition:

Joins are used in relational databases to combine data from multiple tables based on a related column between them.

Types of joins:

There are several types of joins in relational databases, including:

1. INNER JOIN
2. LEFT JOIN (or LEFT OUTER JOIN)
3. RIGHT JOIN (or RIGHT OUTER JOIN)
4. FULL OUTER JOIN
5. CROSS JOIN (or CARTESIAN JOIN)
6. SELF JOIN

Joins - INNER JOIN

Definition:

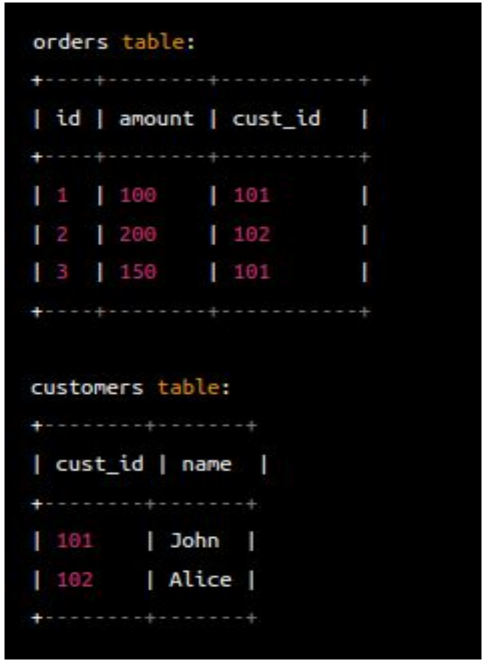
It returns only the rows that have matching values in both tables being joined. Example: SELECT * FROM customers INNER JOIN orders ON customers.customer_id = orders.customer_id;

Key Points:

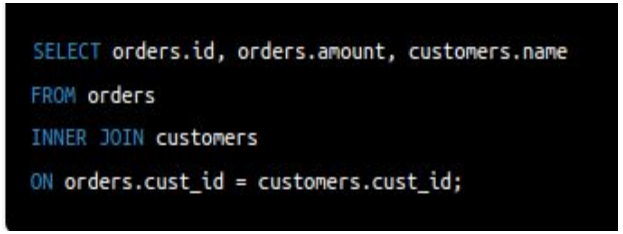
- An INNER JOIN is the most commonly used join operation in SQL.
- It returns only the matching rows from both tables involved in the join.
- The join condition is specified using the ON keyword.
- In an INNER JOIN, only the rows that satisfy the join condition are included in the result set.
- If there are no matching rows in either table, then no rows are returned in the result set.

Author: Muhammad Jahanzaib

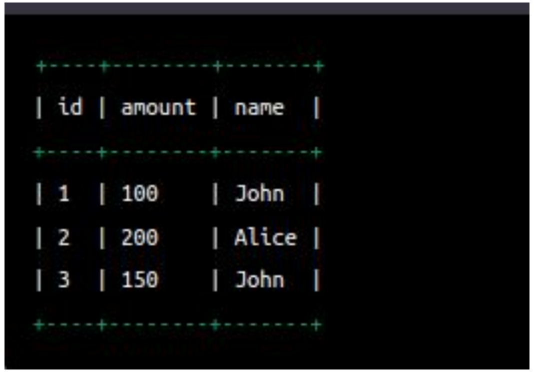
Joins - INNER JOIN



```
orders table:
+----+--------+-----------+
| id | amount | cust_id   |
+----+--------+-----------+
| 1  | 100    | 101       |
| 2  | 200    | 102       |
| 3  | 150    | 101       |
+----+--------+-----------+

customers table:
+---------+-------+
| cust_id | name  |
+---------+-------+
| 101     | John  |
| 102     | Alice |
+---------+-------+
```



```sql
SELECT orders.id, orders.amount, customers.name
FROM orders
INNER JOIN customers
ON orders.cust_id = customers.cust_id;
```



```
+----+---------+-------+
| id | amount  | name  |
+----+---------+-------+
| 1  | 100     | John  |
| 2  | 200     | Alice |
| 3  | 150     | John  |
+----+---------+-------+
```

Joins - LEFT JOIN (or LEFT OUTER JOIN)

Definition:

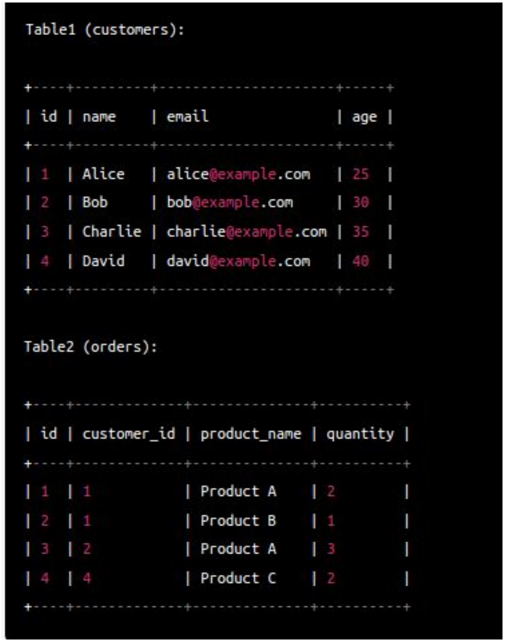
LEFT JOIN, also known as LEFT OUTER JOIN, is a type of join operation in relational databases that combines the matching rows from two tables, as well as any unmatched rows from the left (or first) table.

Key Points:

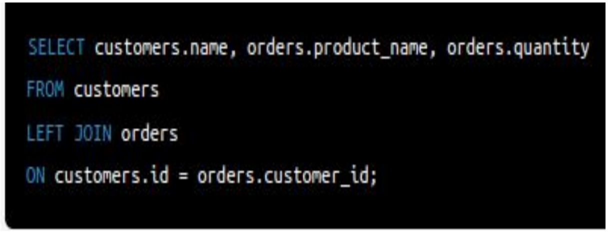
- The result of a LEFT JOIN includes all the rows from the left table and the matching rows from the right table. If there are no matching rows in the right table, the result will still include the row from the left table but with NULL values for the columns from the right table.
- In a LEFT JOIN, the left table is always the first table in the JOIN clause, while the right table is the second table.
- The JOIN condition in a LEFT JOIN specifies how the rows from the two tables are matched. If there is no matching row in the right table, the NULL values will be used for the columns of the right table in the result set.

Author: Muhammad Jahanzaib

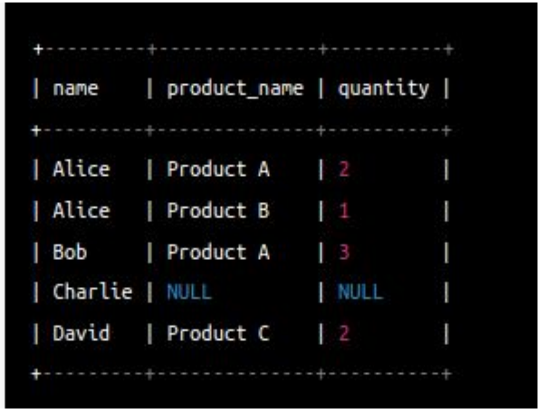
Joins - LEFT JOIN (or LEFT OUTER JOIN)



```
Table1 (customers):

+----+---------+-------------------+-----+
| id | name    | email             | age |
+----+---------+-------------------+-----+
| 1  | Alice   | alice@example.com | 25  |
| 2  | Bob     | bob@example.com   | 30  |
| 3  | Charlie | charlie@example.com | 35  |
| 4  | David   | david@example.com | 40  |
+----+---------+-------------------+-----+

Table2 (orders):

+----+-------------+--------------+----------+
| id | customer_id | product_name | quantity |
+----+-------------+--------------+----------+
| 1  | 1           | Product A    | 2        |
| 2  | 1           | Product B    | 1        |
| 3  | 2           | Product A    | 3        |
| 4  | 4           | Product C    | 2        |
+----+-------------+--------------+----------+
```



```sql
SELECT customers.name, orders.product_name, orders.quantity
FROM customers
LEFT JOIN orders
ON customers.id = orders.customer_id;
```



```
+---------+--------------+----------+
| name    | product_name | quantity |
+---------+--------------+----------+
| Alice   | Product A    | 2        |
| Alice   | Product B    | 1        |
| Bob     | Product A    | 3        |
| Charlie | NULL         | NULL     |
| David   | Product C    | 2        |
+---------+--------------+----------+
```

Joins - RIGHT JOIN (or RIGHT OUTER JOIN)

Definition:

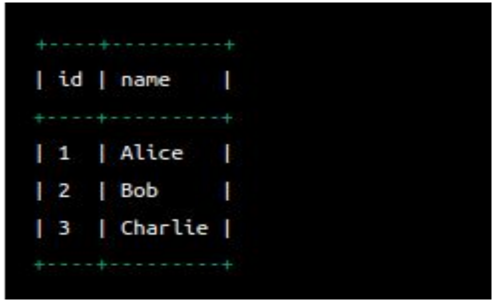
A RIGHT JOIN (or RIGHT OUTER JOIN) is a type of join operation in relational databases that returns all the rows from the right table and matching rows from the left table based on a specified join condition. In other words, a RIGHT JOIN returns all the rows from the right table, and if there are matching rows in the left table, those rows are also included in the result set. If there are no matching rows in the left table, NULL values are used for the corresponding columns in the result set.

Key Points:

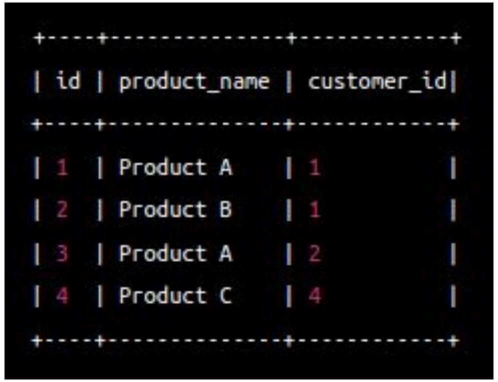
- RIGHT JOIN returns all the rows from the right table and matching rows from the left table. If there is no match, NULL values are returned for the left table columns.
- It is useful when we want to include all the rows from the right table in the result set, even if there are no matches in the left table.
- The syntax for RIGHT JOIN is similar to that of INNER JOIN and LEFT JOIN. The only difference is the keyword used.
- We should use RIGHT JOIN when we want to include all the rows from the right table, and we only want to see the matching rows from the left table.

Author: Muhammad Jahanzaib

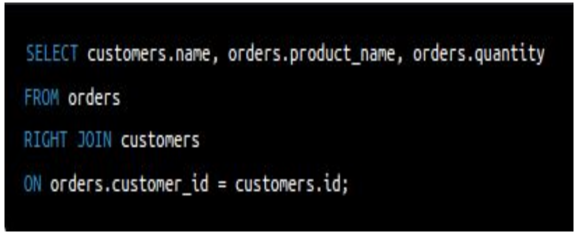
Joins - RIGHT JOIN (or RIGHT OUTER JOIN)



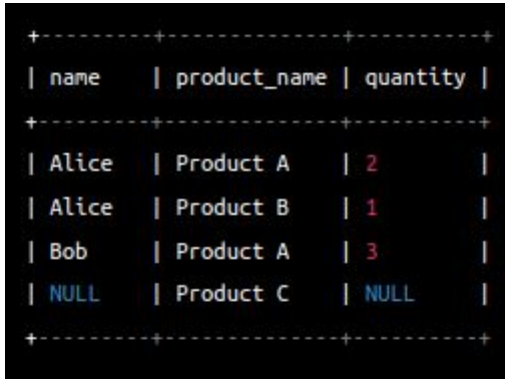
```
+----+---------+
| id | name    |
+----+---------+
| 1  | Alice   |
| 2  | Bob     |
| 3  | Charlie |
+----+---------+
```



```
+----+--------------+-----------+
| id | product_name | customer_id|
+----+--------------+-----------+
| 1  | Product A    | 1         |
| 2  | Product B    | 1         |
| 3  | Product A    | 2         |
| 4  | Product C    | 4         |
+----+--------------+-----------+
```



```
SELECT customers.name, orders.product_name, orders.quantity
FROM orders
RIGHT JOIN customers
ON orders.customer_id = customers.id;
```



```
+---------+--------------+----------+
| name    | product_name | quantity |
+---------+--------------+----------+
| Alice   | Product A    | 2        |
| Alice   | Product B    | 1        |
| Bob     | Product A    | 3        |
| NULL    | Product C    | NULL     |
+---------+--------------+----------+
```

In this result table, you can see that all rows from the orders table are included, along with matching rows from the customers table based on the customer_id foreign key relationship. Rows from the customers table that don't have matching rows in the orders table are still included in the result, but with NULL values in the columns from the orders table.

Author: Muhammad Jahanzaib

Joins - FULL OUTER JOIN

Definition:

A FULL OUTER JOIN returns all the matching rows from both tables, as well as the non-matching rows from both tables. If there are no matches, NULL values are returned.

Key Points:

- Returns all the matching and non-matching rows from both tables
- Non-matching rows are filled with NULL values
- Syntax: SELECT * FROM table1 FULL OUTER JOIN table2 ON condition;

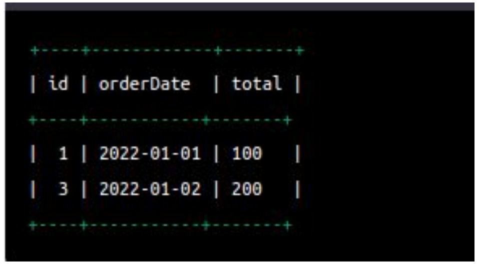
Best Practices:

- Use FULL OUTER JOIN when you want to include all the matching and non-matching rows from both tables.
- Be careful when using FULL OUTER JOIN with large tables as it can significantly impact performance.

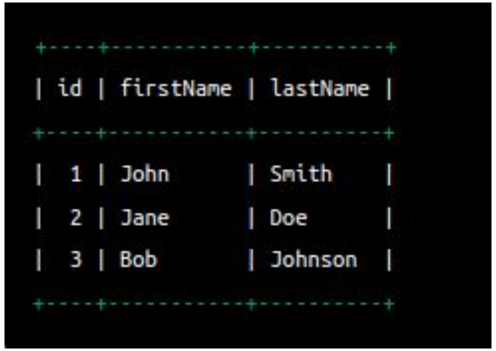
Author: Muhammad Jahanzaib

Joins - FULL OUTER JOIN

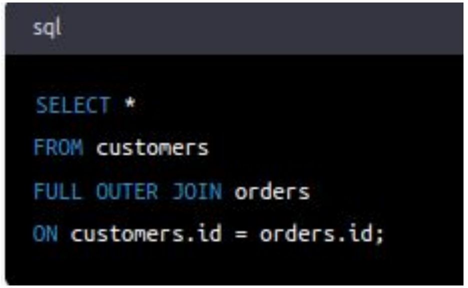
Consider two tables, ***customers*** and ***orders***, with the following structures:



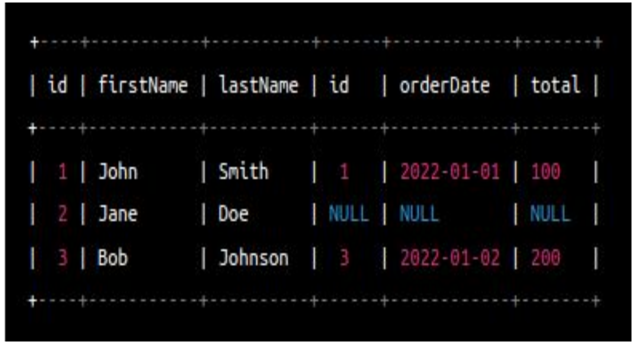
```
+----+------------+-------+
| id | orderDate  | total |
+----+------------+-------+
|  1 | 2022-01-01 | 100   |
|  3 | 2022-01-02 | 200   |
+----+------------+-------+
```



```
+----+-----------+----------+
| id | firstName | lastName |
+----+-----------+----------+
|  1 | John      | Smith    |
|  2 | Jane      | Doe      |
|  3 | Bob       | Johnson  |
+----+-----------+----------+
```



```sql
SELECT *
FROM customers
FULL OUTER JOIN orders
ON customers.id = orders.id;
```



```
+----+-----------+----------+------+------------+-------+
| id | firstName | lastName | id   | orderDate  | total |
+----+-----------+----------+------+------------+-------+
|  1 | John      | Smith    |  1   | 2022-01-01 | 100   |
|  2 | Jane      | Doe      | NULL | NULL       | NULL  |
|  3 | Bob       | Johnson  |  3   | 2022-01-02 | 200   |
+----+-----------+----------+------+------------+-------+
```

Joins - CROSS JOIN (or CARTESIAN JOIN)

Definition:

A Cross Join, also known as a Cartesian Join, is a type of join in which every row from one table is joined with every row from another table. In other words, it creates a Cartesian product of the two tables.

Key Points:

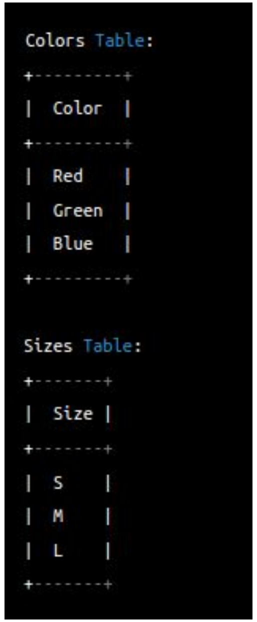
- Cross Join is used when you want to combine every row from one table with every row from another table.
- The resulting table will have the total number of rows equal to the product of the number of rows in both tables.
- Cross Join does not use any join condition or ON clause.

Best Practices:

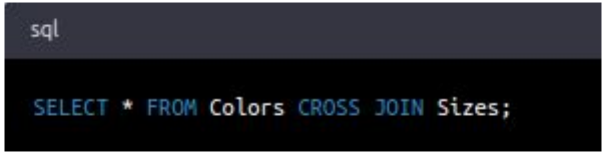
- Cross Join should be used with caution as it can create a large number of rows in the resulting table.
- Always specify the tables to be joined explicitly to avoid unintentional Cross Joins.
- Cross Joins are not commonly used in day-to-day operations, but they can be useful in specific scenarios like generating test data or analyzing data.

Joins - CROSS JOIN (or CARTESIAN JOIN)

Consider two tables, ***colors*** and ***sizes***, with the following structures:

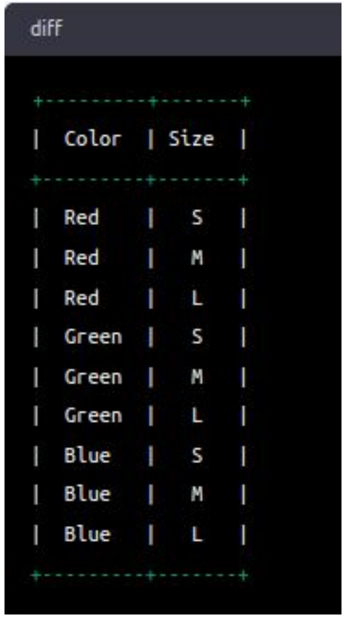


```
Colors Table:
+---------+
| Color   |
+---------+
| Red     |
| Green   |
| Blue    |
+---------+

Sizes Table:
+-------+
| Size |
+-------+
| S     |
| M     |
| L     |
+-------+
```



sql

```sql
SELECT * FROM Colors CROSS JOIN Sizes;
```



diff

```
+---------+-------+
| Color   | Size  |
+---------+-------+
| Red     |   S   |
| Red     |   M   |
| Red     |   L   |
| Green   |   S   |
| Green   |   M   |
| Green   |   L   |
| Blue    |   S   |
| Blue    |   M   |
| Blue    |   L   |
+---------+-------+
```

Joins - SELF JOIN

Definition:

A self join is a regular join, but the table is joined with itself. It is useful when a table has a hierarchical structure or when there is a need to compare rows within the same table.

Key Points:

- In a self join, the table is referenced twice with different aliases to distinguish between the two copies of the table.
- Self join can be used for several scenarios, including finding hierarchical relationships between data, comparing rows within the same table, and finding duplicates in a table.
- Self join can be performed using different types of join, including inner join, left join, right join, and full outer join.

Best Practices:

- Use meaningful aliases to distinguish between the copies of the table in the self join.
- Avoid creating circular references in the table structure to prevent infinite loops.
- Optimize the query by using indexes on the columns used in the join condition.

Keys in Databases

Keys in databases are used to uniquely identify a row in a table or to establish a relationship between two or more tables. They are constraints that enforce data integrity and ensure that the data is consistent and accurate

Following are the types of keys:

- Primary key
- Foreign key
- Unique key
- Candidate key
- Composite key
- Compound Key
- Surrogate key
- Super key

Author: Muhammad Jahanzaib

Keys - Primary Key

Definition:

A primary key is a column or combination of columns in a table that uniquely identifies each row in the table. It is used to enforce data integrity and ensure that no two rows in a table are identical.

Key Points:

- Must contain unique values
- Cannot contain NULL values
- Can consist of one or more columns

Best Practices:

- Choose a column or combination of columns that are always present and unique to the table.
- Avoid using columns that are frequently updated or have a high number of NULL values.

Author: Muhammad Jahanzaib

Keys - Foreign key

Definition:

A foreign key is a field or set of fields in one table that uniquely identifies a row of another table or the same table.

Key Points:

- It is used to link tables together.
- It helps to maintain referential integrity by ensuring that rows in one table that refer to rows in another table actually correspond to real rows in the referenced table.
- The foreign key constraint specifies that the values in one table must match the values in another table.

Best Practices:

- Choose meaningful column names for foreign keys to make it easier to understand the relationship between tables.
- Always create a foreign key constraint when defining relationships between tables to ensure referential integrity.

Keys - Unique key

Definition:

A Unique Key is a database constraint that ensures that the values in a column or a group of columns are unique across all the rows in the table. Unlike the primary key, a table can have multiple unique keys. A unique key constraint ensures that no two rows in a table have the same values in the specified column(s).

Key Points:

- A unique key constraint can be applied to one or more columns in a table.
- A unique key constraint can be applied to a NULL column. In that case, the constraint ensures that only one NULL value is allowed in the column.
- A unique key is a type of index in a database that provides faster data retrieval.
- A unique key constraint is automatically created when a column is designated as a PRIMARY KEY.

Best Practices:

- Use a unique key constraint on columns that are frequently used in search operations.
- Avoid using a unique key constraint on columns that have a low degree of uniqueness, as it can negatively impact performance.

Keys - Composite key

Definition:

A composite key is a combination of two or more columns in a table that together uniquely identifies each row in the table. It is also referred to as a composite primary key. In other words, a composite key is a set of columns that, when combined, uniquely identify a row in a table.

Key Points:

- A composite key is made up of two or more columns in a table.
- The combination of columns must be unique for each row in the table.
- A composite key is used when no single column can uniquely identify a row.
- A composite key can consist of any data type.
- A composite key can be used as a foreign key in another table.

Best Practices:

- Keep the number of columns in the composite key to a minimum.
- Choose columns that are not likely to change.
- Consider the performance impact of using a composite key, as it can affect insert, update, and delete operations.
- Document the composite key in the table definition to make it clear which columns are included.

Author: Muhammad Jahanzaib

Keys - Composite key

Definition:

A composite key is a primary key that consists of multiple columns in a table. Instead of using a single column to uniquely identify each row in a table, a composite key uses two or more columns to form a unique identifier.

Key Points:

- A composite key must consist of two or more columns in a table.
- Each column in a composite key plays an important role in uniquely identifying each row in the table.
- Composite keys are often used when a single column cannot provide a unique identifier for a row.
- The columns in a composite key can be of different data types.
- The order of the columns in a composite key is important as it determines the order in which the data is stored in the table.

Best Practices:

- Use composite keys sparingly, only when necessary. In general, it's best to use a single column primary key whenever possible.
- Make sure that the columns in a composite key are relevant to the table and provide a logical way to identify rows.
- Keep the number of columns in a composite key to a minimum to avoid unnecessary complexity.
- Avoid using columns with variable-length data types in a composite key as this can have an impact on performance.
- Ensure that the order of the columns in a composite key is consistent across all tables in a database.

Author: Muhammad Jahanzaib

Keys - Super key

Definition:

A super key is a set of one or more attributes that, taken collectively, allows us to identify a unique record within a table. It is a combination of attributes that can be used to uniquely identify each record in a table. A super key can contain additional attributes beyond those that are minimally required for unique identification.

Key Points:

- A super key is a set of one or more attributes that can uniquely identify a record in a table.
- It can have additional attributes that are not strictly required for unique identification.
- A super key can contain other keys as a subset.
- A table can have multiple super keys.
- Super keys are not necessarily minimal and may contain redundant attributes.

Best Practices:

- It is recommended to use the minimal set of attributes required to uniquely identify a record as a primary key rather than a super key.
- Super keys can be used to enforce uniqueness constraints or as an index for efficient querying.
- It is important to avoid redundant attributes in a super key to prevent unnecessary storage usage and potential inconsistencies in data.

Author: Muhammad Jahanzaib

Indexes

Definition:

Indexes in databases are database objects that allow for fast data retrieval operations. They work by creating a data structure that contains a subset of the data in the table, sorted in a way that makes it easy to search for specific values. This data structure is then used by the database engine to quickly locate the rows that match a specific query.

Key Points:

- Indexes can be created on one or more columns in a table.
- Indexes can be created on either a unique or non-unique set of values.
- Indexes can improve query performance by reducing the amount of data that needs to be scanned.
- Indexes can have a downside in that they take up disk space and require additional processing time to maintain.
- It is important to choose the appropriate columns to index based on the queries that will be run against the table.

Indexes

Following are some types of indexes:

Single-Column Index: This type of index is created on a single column of a table. For example, if you have a table with a large number of rows, and you frequently search for data in a specific column, you can create a single-column index on that column to speed up the search.

Unique Index: A unique index is similar to a single-column index, but it enforces a unique constraint on the indexed column or columns. This means that the indexed columns must contain only unique values. Unique indexes are commonly used for primary and foreign keys.

Composite Index: A composite index is created on two or more columns of a table. This type of index is useful when you frequently search for data based on multiple columns. For example, if you have a table with columns for first name, last name, and age, you can create a composite index on all three columns to speed up searches that involve all three fields.

Full-Text Index: A full-text index is used to search for text-based data, such as articles, blog posts, or other types of unstructured text. It allows you to search for keywords and phrases within the text, and can be useful for applications such as search engines and content management systems.

Author: Muhammad Jahanzaib

Indexes - Clustered vs Non-Clustered indexes

Structure

Clustered indexes determine the physical order of data rows in a table while non-clustered indexes have a separate structure that maps the index key values to the corresponding data rows.

Number of indexes

A table can have only one clustered index but multiple non-clustered indexes.

Query performance

Clustered indexes are generally faster for queries that retrieve a range of data in the clustered index order. Non-clustered indexes are better suited for queries that retrieve a small set of rows based on specific column values.

Maintenance

When you insert new data into a table with a clustered index, the data is physically inserted in the corresponding order in the clustered index. This means that the entire index may need to be rebuilt periodically. Non-clustered indexes are less affected by inserts and updates as they only contain a copy of the index key values and a pointer to the corresponding data rows.

Author: Muhammad Jahanzaib

Indexes - Clustered vs Non-Clustered indexes

Storage

Clustered indexes consume less storage than non-clustered indexes as they are integrated with the table structure.

Primary key

Clustered indexes are typically created on the primary key of a table, while non-clustered indexes can be created on any column.

Sorting

Clustered indexes maintain the sort order of the table rows while non-clustered indexes do not.

Fragmentation

Non-clustered indexes can suffer from fragmentation which can impact performance. Clustered indexes are less susceptible to fragmentation.

Transactions

Definition:

Transactions are a fundamental concept in databases that ensure the consistency and reliability of data. A transaction is a series of operations or actions that are performed as a single unit of work. These actions must be executed in full or not at all. *Some key points about transactions include:*

Atomicity

A transaction is an atomic unit of work, meaning that it must either succeed completely or fail completely. If a transaction fails, all the changes made during that transaction must be rolled back to their previous state.

Consistency

A transaction must leave the database in a consistent state. This means that the database must follow all the integrity constraints, such as primary key, foreign key, unique key, etc.

Isolation

Transactions should be isolated from each other to avoid conflicts. Each transaction should be able to work independently without affecting others.

Durability

Once a transaction is committed, the changes made during that transaction should be permanent and not affected by any system failure.

What is SQL?

Definition:

- SQL stands for Structured Query Language.
- SQL lets you access and manipulate databases.
- SQL became a standard of the American National Standards Institute (ANSI) in 1986, and of the International Organization for Standardization (ISO) in 1987

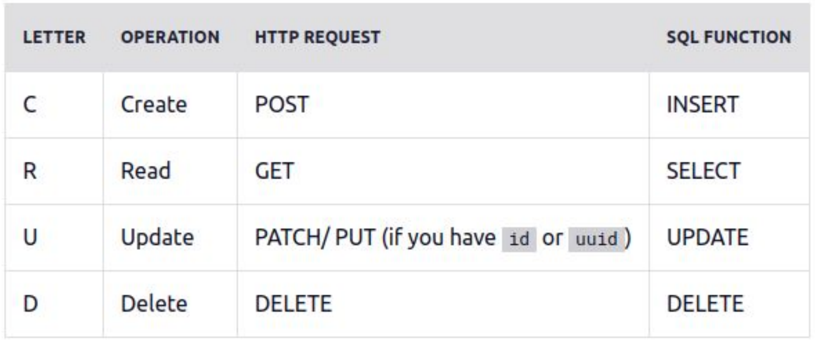
Key Points:

- SQL can execute queries against a database
- SQL can retrieve data from a database
- SQL can insert records in a database
- SQL can update records in a database
- SQL can delete records from a database
- SQL can create new databases
- SQL can create new tables in a database
- SQL can create stored procedures in a database
- SQL can create views in a database
- SQL can set permissions on tables, procedures, and views

SQL - CRUD Operations

Definition:

CRUD is an acronym for Create, Read, Update and Delete. Each of these performs different operations, but they all aim to track and manage data, from a database, API, or whatever.
When creating a database or building APIs, you will want users to be able to manipulate any data available either by fetching these data, updating the data, deleting them, or adding more data. These operations are made possible through CRUD operations.



LETTER	OPERATION	HTTP REQUEST	SQL FUNCTION
C	Create	POST	INSERT
R	Read	GET	SELECT
U	Update	PATCH/ PUT (if you have `id` or `uuid`)	UPDATE
D	Delete	DELETE	DELETE

Deep dive

Click here

Transactions - Example

Let's say a customer wants to transfer $100 from their checking account to their savings account. Here are the steps involved in this transaction:

1. **Begin Transaction:** The transaction starts with a BEGIN TRANSACTION statement. This informs the database system that a transaction is beginning.
2. **Update Checking Account:** The system deducts $100 from the customer's checking account using an UPDATE statement.
3. **Update Savings Account:** The system then adds $100 to the customer's savings account using another UPDATE statement.
4. **Commit Transaction:** Once both updates are completed successfully, the transaction is committed using the COMMIT statement. This indicates to the system that the transaction is complete and all updates should be saved to the database.
5. **Rollback Transaction:** If there was an issue during the transaction, such as a system failure or a database error, the transaction can be rolled back using the ROLLBACK statement. This undoes all updates made during the transaction, ensuring that the database remains consistent.

Author: Muhammad Jahanzaib

Transactions - Interview Questions

1. What is a database transaction, and why is it important? Can you give an example of a situation where a transaction would be used?
2. What is the difference between a committed transaction and a rolled-back transaction? Can you give an example of how you would use each one?
3. Can you explain the concept of isolation levels in database transactions? What are the different levels, and how do they affect transactions?
4. How do you start and end a transaction in SQL? What are the required commands and the required permissions?
5. Can you explain the concept of a savepoint in a transaction? When would you use a savepoint, and how does it work?
6. What is the purpose of the ACID properties in a database transaction? Can you explain each property and how it relates to transactions?
7. Can you give an example of how you would handle concurrent access to a database table in a transaction? How would you prevent conflicts and ensure data consistency?
8. Can you explain the concept of a distributed transaction? How is it different from a regular transaction, and what are the challenges of implementing it?
9. Can you explain the concept of a two-phase commit protocol? How does it ensure transaction consistency in a distributed system?
10. How do you handle errors in a transaction?

Author: Muhammad Jahanzaib

Triggers

Definition:

Triggers are special types of stored procedures that are automatically executed in response to certain database events or changes. They are used to enforce business rules or policies and maintain data consistency and integrity.. *Here are some examples of triggers:*

Audit Trail Trigger

This trigger is used to create an audit trail of all the changes made to a specific table. The trigger is fired whenever an insert, update, or delete operation is performed on the table.

Cascade Trigger

This trigger is used to implement cascading updates or deletes on related tables. For example, if a record in the parent table is deleted, all related records in the child table(s) are also deleted.

Validation Trigger

This trigger is used to validate the data being inserted or updated in a table. The trigger is fired before the insert or update operation is performed and can be used to enforce complex business rules or data validation rules.

Security Trigger

This trigger is used to implement security measures on a database. For example, a trigger can be created to prevent unauthorized access to a specific table or to log all attempts to access the database.

Author: Muhammad Jahanzaib

Stored Procedures

Definition:

A stored procedure is a pre-written program that is stored in a database and can be executed on demand. It is a set of SQL statements that are compiled and stored in the database, and it can be executed by invoking the procedure name.

Key Points:

- **They are reusable:** Once created, stored procedures can be called and executed multiple times, reducing code redundancy and improving application performance.
- **They are efficient:** Stored procedures are pre-compiled and stored in memory, which can help to reduce database overhead and increase performance.
- **They are secure:** Stored procedures can be executed with limited permissions, reducing the risk of unauthorized access to sensitive data.
- **They are modular:** Stored procedures can be broken down into smaller, more manageable chunks of code, making it easier to develop and maintain complex applications.

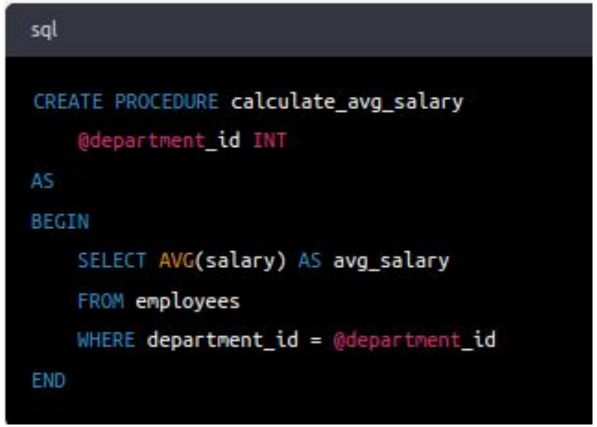
Best Practices:

- **Keep it simple:** Avoid creating overly complex stored procedures, as they can be difficult to understand and maintain.
- **Use input and output parameters:** Parameters can help to make stored procedures more flexible and reusable.
- **Avoid using dynamic SQL:** Dynamic SQL can be less secure and less efficient than static SQL.
- **Document your stored procedures:** Make sure to document your stored procedures so that other developers can understand how they work and how to use them.
- **Test your stored procedures:** Make sure to thoroughly test your stored procedures to ensure that they are working correctly and efficiently.

Stored Procedures - Example

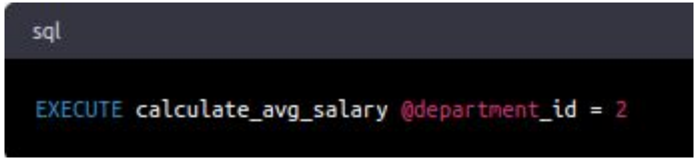
An example of a stored procedure might be a procedure that calculates the average salary of all employees in a given department. The steps to create and execute this stored procedure might look like this:

1. **Create the stored procedure:**



```sql
CREATE PROCEDURE calculate_avg_salary
    @department_id INT
AS
BEGIN
    SELECT AVG(salary) AS avg_salary
    FROM employees
    WHERE department_id = @department_id
END
```

2. **Execute the stored procedure:**



```sql
EXECUTE calculate_avg_salary @department_id = 2
```

This would calculate and return the average salary of all employees in department 2

Author: Muhammad Jahanzaib

Triggers & Stored Procedures - Interview Questions

1. What is the difference between a trigger and a stored procedure? Can you give an example of when you would use one over the other?
2. What is a cascading trigger? Can you give an example of how it works?
3. How do you create a trigger in SQL? What are the syntax and the required permissions?
4. How do you handle errors in a trigger or a stored procedure? Can you give an example of how you would do this?
5. What is the difference between an AFTER trigger and a BEFORE trigger? Can you give an example of when you would use each type of trigger?
6. What are the advantages of using stored procedures in a database? Can you give an example of how using a stored procedure can improve performance?
7. Can you explain the concept of a parameterized stored procedure? How is it different from a regular stored procedure?
8. Can you update or delete data from a stored procedure? Why or why not? What are the limitations?
9. How do you debug a stored procedure or a trigger in SQL? Can you give an example of how you would do this?
10. Can you explain the concept of a nested stored procedure? How is it different from a regular stored procedure, and when would you use it?

Views

Definition:

Views in databases are virtual tables that allow users to query data from one or more existing tables in a simplified manner. Instead of writing complex SQL queries every time, views allow users to define a set of commonly used joins and filters as a single view, which can then be queried as if it were a real table.

Some of the key benefits of using views include:

Simplified data access

Views provide a simplified way of accessing data from multiple tables without the need to write complex SQL queries every time

Enhanced security:

Views can be used to limit access to specific columns or rows of data, providing an additional layer of security to the database.

Improved performance:

Views can be used to precompute the results of complex queries, which can improve the performance of frequently executed queries.

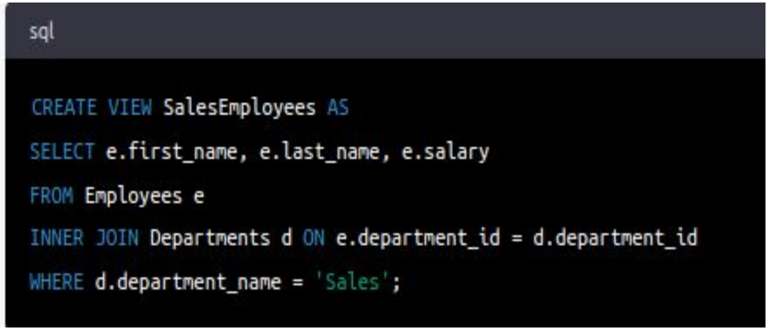
Improved maintainability:

Views encapsulate complex logic or frequently used queries, making database maintenance and updates easier. View definitions can be modified without affecting the underlying tables or applications, reducing the time and effort required for development and maintenance.

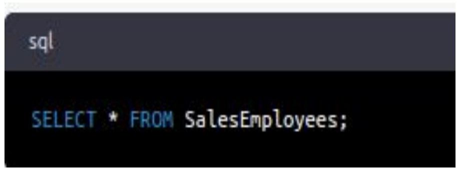
Author: Muhammad Jahanzaib

Views - Example

Consider a database with two tables: ***Employees*** and ***Departments***. The ***Employees*** table contains information about all the employees in the company, while the ***Departments*** table contains information about all the departments in the company.
To create a view that shows the names and salaries of all employees in the ***Sales*** department, we can use the following SQL query:



```sql
CREATE VIEW SalesEmployees AS
SELECT e.first_name, e.last_name, e.salary
FROM Employees e
INNER JOIN Departments d ON e.department_id = d.department_id
WHERE d.department_name = 'Sales';
```



```sql
SELECT * FROM SalesEmployees;
```

This query will return a result set containing the first name, last name, and salary of all employees in the Sales department.

*This view will create a virtual table called **SalesEmployees** that contains the first name, last name, and salary of all employees in the **Sales** department. The view can then be queried using a simple **SELECT** statement:*

Author: Muhammad Jahanzaib

Views - Interview Questions

1. What is a database view, and how is it different from a table? Can you give an example of when you would use a view?
2. Can you update or delete data from a view? Why or why not? What are the limitations?
3. What are the advantages of using views in a database? Can you give an example of how using a view can simplify a query?
4. How do you create a view in SQL? What are the syntax and the required permissions?
5. Can you explain the concept of a materialized view? How is it different from a regular view, and when would you use it?

Author: Muhammad Jahanzaib

Aggregate Functions

Aggregate functions are SQL functions that operate on a group of rows and return a single value as a result. They are commonly used in SQL queries to perform calculations and summaries on data.
Here are some common aggregate functions:

1. **COUNT**: Returns the number of rows in a specified column.
2. **SUM:** Returns the sum of values in a specified column.
3. **AVG:** Returns the average value of values in a specified column.
4. **MAX:** Returns the maximum value in a specified column.
5. **MIN:** Returns the minimum value in a specified column.

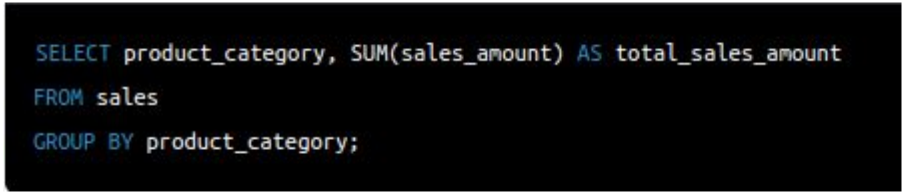
Applications:

Aggregate functions are useful for data analysis and reporting as they allow us to perform calculations and summarize data quickly and easily.

Aggregate Functions with GROUP BY Clause

Aggregate functions can also be used in combination with the GROUP BY clause to group the data based on certain criteria and perform calculations on the grouped data. For example, to find the total sales amount by product category, we can use the SUM function with the GROUP BY clause on the product category column.

An example of using the SUM function to calculate the total sales amount for each product category:



```
SELECT product_category, SUM(sales_amount) AS total_sales_amount
FROM sales
GROUP BY product_category;
```

*This query selects the **product_category** column and the sum of **sales_amount** column for each distinct **product_category** value in the **sales** table. The **GROUP BY** clause groups the data based on the **product_category** column, and the **SUM** function calculates the total sales amount for each group. The **AS** keyword is used to alias the result column as **total_sales_amount**.*

Author: Muhammad Jahanzaib

Aggregate Functions - Interview Questions

1. What is the difference between COUNT(*) and COUNT(column_name)? When would you use one over the other?
2. What is the difference between SUM() and AVG()? Can you give an example of when you would use each function?
3. How do you handle NULL values when using aggregate functions like MIN(), MAX(), AVG() and SUM()?
4. What is the difference between GROUP BY and HAVING clauses in a SQL statement? Can you give an example of when you would use each clause?
5. How do you combine aggregate functions with non-aggregate functions in a SQL statement? Can you give an example?

Author: Muhammad Jahanzaib

Sub Queries

Definition:

Subqueries are queries that are embedded inside another query and can be used to retrieve data that will be used as a part of the main query. The subquery is enclosed in parentheses and can be used in various parts of the main query such as the SELECT, FROM, WHERE, and HAVING clauses. Subqueries can be used to perform complex queries and are particularly useful when the result of one query is used as a filter for another query.

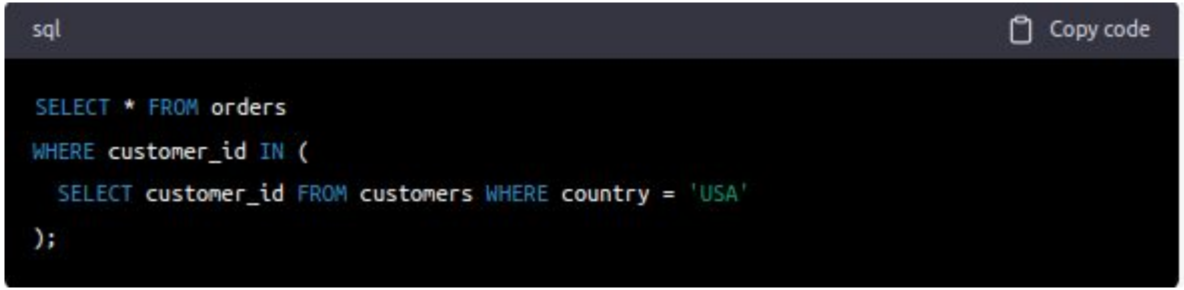
Key Points:

- A subquery can return a single value or a set of values to the main query.
- A subquery can be nested inside another subquery to form a more complex query.
- Subqueries can be used with comparison operators such as =, >, <, >=, <=, IN, NOT IN, EXISTS, and NOT EXISTS.
- Subqueries can also be used to retrieve data from multiple tables using joins.

Author: Muhammad Jahanzaib

Sub Queries - Example

Retrieving data using a subquery in the WHERE clause:



sql Copy code

```sql
SELECT * FROM orders
WHERE customer_id IN (
  SELECT customer_id FROM customers WHERE country = 'USA'
);
```

This query retrieves all orders made by customers from the USA. The subquery in the WHERE clause retrieves the IDs of customers from the USA, which are then used to filter the orders table.