**SQL INTERVIEW QUESTION AND ANSWER**

1. Different ways to autoincrement ID value
2. **Using AUTO\_INCREMENT Attribute**:
   * The most common and recommended way in MySQL is to use the **AUTO\_INCREMENT** attribute when defining the ID column. This attribute automatically generates a unique sequential integer value for each new row inserted into the table.
3. **Using Triggers**:
   * Triggers can be used to autoincrement the ID value by setting the value of the ID column based on the maximum value of the ID column in the table plus one. However, triggers can add complexity and overhead to the database.
4. **Custom Functions**:
   * Custom functions can be defined to generate unique ID values based on specific criteria or rules. While this approach offers flexibility, it may require additional development effort and may not be as efficient as other methods.
5. **Using UUID() Function**:
   * The **UUID()** function can generate Universally Unique Identifier (UUID) values, which are globally unique identifiers. However, UUIDs are not sequential and can result in larger storage requirements compared to integer IDs.
6. **Custom Sequence Table**:
   * A separate sequence table can be created to store a sequence of numbers, and these numbers can be used to generate unique ID values for the ID column. This approach provides more control over the ID generation process but may introduce additional complexity.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. primary key vs foreign key

**Primary Key**:

* A primary key is a column or a set of columns in a table that uniquely identifies each row in the table.
* It ensures that each row has a unique identifier and enforces entity integrity, preventing duplicate or null values in the key columns.
* Primary keys are typically defined when creating tables using the **PRIMARY KEY** constraint.
* CREATE TABLE students (
* student\_id INT PRIMARY KEY,
* name VARCHAR(50),
* ...
* );

**Foreign Key**:

* A foreign key is a column or a set of columns in a table that establishes a relationship with a primary key or a unique key in another table.
* It represents a reference from one table (the child table) to another table (the parent table), enforcing referential integrity.
* Foreign keys ensure that values in the referencing column(s) (the child table) must exist in the referenced column(s) (the parent table) or be null.
* Foreign keys are typically defined using the **FOREIGN KEY** constraint.
* CREATE TABLE enrollments (
* enrollment\_id INT PRIMARY KEY,
* student\_id INT,
* course\_id INT,
* FOREIGN KEY (student\_id) REFERENCES students(student\_id),
* FOREIGN KEY (course\_id) REFERENCES courses(course\_id)
* );
* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**.3. Data consistency**

Data consistency in a MySQL database refers to the accuracy, reliability, and integrity of the data stored within it. It ensures that the data remains valid and reliable throughout its lifecycle, from insertion to retrieval and modification.

Here are several aspects of maintaining data consistency in MySQL:

1. **Primary Keys**: Primary keys uniquely identify each record in a table. They ensure that each row has a unique identifier, preventing duplicate records and ensuring data uniqueness.
2. **Foreign Keys and Referential Integrity**: Foreign keys establish relationships between tables, ensuring referential integrity. They enforce constraints that prevent actions that would leave orphaned records or violate relationships between tables.
3. **Constraints**: Constraints such as **NOT NULL**, **UNIQUE**, and **CHECK** ensure that data adheres to predefined rules, preventing invalid data from being inserted into the database.
4. **Transactions**: Transactions in MySQL ensure that a series of database operations are performed as a single unit of work. They guarantee that either all operations within the transaction are executed successfully (commit) or none of them are (rollback), maintaining data consistency.
5. **Isolation Levels**: MySQL provides different isolation levels for transactions, such as **READ COMMITTED**, **REPEATABLE READ**, and **SERIALIZABLE**. These isolation levels control how transactions interact with each other, ensuring data consistency while balancing concurrency and performance.
6. **Indexing**: Proper indexing of tables can improve query performance and maintain data consistency by ensuring efficient data retrieval.
7. **Regular Backups and Disaster Recovery**: Regular backups of the database ensure that data can be restored in the event of data loss or corruption. Implementing a robust disaster recovery plan helps maintain data consistency and availability.
8. **Data Validation and Sanitization**: Perform data validation and sanitization to ensure that only valid and sanitized data is stored in the database, reducing the risk of data corruption or inconsistency.
9. **Monitoring and Maintenance**: Regularly monitor database performance, integrity, and consistency. Perform routine maintenance tasks such as optimizing queries, updating statistics, and performing database maintenance operations to ensure data consistency.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

1. What is the difference between view and Materialized View?
2. **View**:
   * A view is a virtual table created from a SELECT query. It does not store any data on its own but instead fetches data dynamically from the underlying tables whenever it is queried.
   * Views are primarily used for simplifying complex queries, providing a layer of abstraction over the underlying tables, and controlling access to the data.
   * Whenever a query is executed against a view, MySQL executes the underlying SELECT statement and returns the result set to the user.
   * Views are suitable for scenarios where real-time access to up-to-date data is required, and the underlying data does not change frequently.
3. **Materialized View**:
   * A materialized view is a physical copy of the result set of a query stored as a table-like structure in the database. Unlike views, materialized views store data persistently.
   * Materialized views are updated periodically, either based on a defined schedule or in response to changes in the underlying data, to keep the data fresh and up-to-date.
   * Querying a materialized view does not involve executing the underlying SELECT statement every time. Instead, it retrieves data directly from the precomputed result set stored in the materialized view.
   * Materialized views are useful for improving query performance by precomputing and caching the results of expensive or frequently executed queries. They are particularly beneficial in scenarios where query performance is critical, and the underlying data is relatively static or changes infrequently.
4. what is view

In MySQL, a view is a virtual table that is derived from the result set of a SELECT query. Unlike a physical table, a view does not store data on its own; instead, it represents the data stored in one or more underlying tables. Views provide a way to simplify complex queries, encapsulate logic, and provide a layer of abstraction over the underlying data.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. How query is different from View

In MySQL, a query and a view serve different purposes, although they both involve retrieving data from one or more tables. Here are the key differences between a query and a view:

1. **Definition**:
   * **Query**: A query is a statement written in SQL that retrieves data from one or more tables based on specified criteria. It is executed each time it is invoked.
   * **View**: A view is a virtual table derived from the result set of a SELECT query. It encapsulates the logic of the query and is stored in the database as a named object.
2. **Execution**:
   * **Query**: A query is executed whenever it is invoked. Each time you run a query, MySQL re-evaluates the query against the database and returns the result set based on the current data.
   * **View**: A view is precomputed and stored in the database. When you query a view, MySQL retrieves the result set from the stored view definition rather than re-executing the underlying SELECT statement.
3. **Storage**:
   * **Query**: A query is not stored in the database. It exists only as a statement that is executed dynamically when invoked.
   * **View**: The definition of a view, including the underlying SELECT statement, is stored in the database. However, the data itself is not stored separately; it is derived dynamically from the underlying tables when the view is queried.
4. **Abstraction**:
   * **Query**: A query provides a way to retrieve specific data from the database at a given point in time. It is typically used for ad-hoc data retrieval or for executing specific operations.
   * **View**: A view provides a layer of abstraction over the underlying data. It allows you to define complex queries, encapsulate logic, and provide a simplified interface for accessing data.
5. **Reusability**:
   * **Query**: A query can be written and executed as needed, but it does not persist beyond the execution scope.
   * **View**: A view can be created once and reused multiple times. Once defined, a view can be queried like a regular table, making it a reusable and convenient way to access data.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

1. what is Outer & inner join

**Inner Join**:

* An inner join returns only the rows from both tables that satisfy the join condition. In other words, it returns the intersection of the two tables.

**Outer Join**:

* Outer joins return all rows from one or both tables, along with matching rows where available. If there are no matches, NULL values are filled in for columns from the other table.
* There are three types of outer joins:
  + **LEFT JOIN**: Returns all rows from the left table and matching rows from the right table.
  + **RIGHT JOIN**: Returns all rows from the right table and matching rows from the left table.
  + **FULL JOIN**: Returns all rows from both tables, including unmatched rows from both tables.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

1. what is normalization in SQL

Normalization in MySQL is the process of organizing data in a database efficiently. It involves breaking down a large table into smaller, related tables and defining relationships between them. The main goals of normalization are to minimize redundancy, prevent data anomalies, and ensure data integrity.

There are several normal forms (1NF, 2NF, 3NF, BCNF, 4NF, 5NF, and others) that define specific rules to achieve a normalized database schema. Here's a brief overview of some common normal forms:

1. **First Normal Form (1NF)**:
   * Ensures that each column in a table contains atomic values (indivisible values).
   * Eliminates repeating groups and ensures that each column has a single value for each row.
2. **Second Normal Form (2NF)**:
   * Builds on 1NF by ensuring that all non-key attributes are fully dependent on the entire primary key, not just part of it.
   * Requires breaking down tables with composite primary keys and moving non-key attributes to separate tables.
3. **Third Normal Form (3NF)**:
   * Builds on 2NF by ensuring that there are no transitive dependencies between non-key attributes.
   * Requires moving attributes that depend on other non-key attributes to separate tables.
4. **Boyce-Codd Normal Form (BCNF)**:
   * A stricter form of 3NF that applies when there are multiple candidate keys in a table.
   * Ensures that every determinant is a candidate key.
5. **Fourth Normal Form (4NF)**:
   * Deals with multi-valued dependencies and requires breaking down tables with such dependencies.
6. **Fifth Normal Form (5NF)**:
   * Addresses cases where there are join dependencies between multiple candidate keys.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

1. what is procedure ,function, anonymous function and cursor?

**Procedure**:

* A procedure is a named block of SQL statements that can be stored and executed in the database. It can accept input parameters, perform actions, and return output parameters or result sets.
* Procedures are typically used to encapsulate frequently performed tasks or business logic, making it easier to manage and reuse code.
* CREATE PROCEDURE sp\_GetEmployee(IN employee\_id INT)
* BEGIN
* SELECT \* FROM employees WHERE employee\_id = employee\_id;
* END;

**Function**:

* A function is similar to a procedure, but it returns a single value or result set. Functions can be used in SQL queries, expressions, or other functions.
* Functions can be either built-in functions provided by MySQL or user-defined functions created by users.
* CREATE FUNCTION fn\_GetFullName(first\_name VARCHAR(50), last\_name VARCHAR(50))
* RETURNS VARCHAR(100)
* BEGIN
* RETURN CONCAT(first\_name, ' ', last\_name);
* END;

**Anonymous Function (Anonymous Block)**:

* An anonymous function, or anonymous block, is a set of SQL statements enclosed within a BEGIN...END block. Unlike procedures or functions, it is not given a name and cannot be stored in the database.
* Anonymous blocks are often used for ad-hoc or one-time tasks that don't require persistence.
* BEGIN
* DECLARE total\_salary DECIMAL(10,2);
* SELECT SUM(salary) INTO total\_salary FROM employees;
* SELECT total\_salary;
* END;

**Cursor**:

* A cursor is a database object used to retrieve and process individual rows returned by a SELECT query. Cursors are useful when you need to iterate over a result set row by row.
* Cursors are typically used within stored procedures or functions to perform row-level processing.
* DECLARE cursor\_name CURSOR FOR SELECT \* FROM employees;
* OPEN cursor\_name;
* FETCH cursor\_name INTO employee\_id, first\_name, last\_name;
* -- Process fetched data
* CLOSE cursor\_name;

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

1. How to create stored procedures, function calls and explain the code in my sql.

**Creating Stored Procedures**:

* Stored procedures are named blocks of SQL statements that can be called and executed within MySQL.

**Creating Functions**:

* Functions are similar to stored procedures but return a single value.
* CALL my\_stored\_procedure();
* SELECT my\_function(10);

**Anonymous Functions**:

* Anonymous functions, also known as inline functions, are not supported in MySQL. All functions must be explicitly named.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

1. Explain me the scenarios where joins will be used and where subquery can we use

**Scenarios for Using Joins**:

* **Retrieving Related Data**: Joins are ideal for retrieving data from multiple tables that have a relationship between them. For example, if you have a **customers** table and an **orders** table, you can use a join to retrieve all orders for each customer.
* **Combining Data**: Joins are used to combine columns from different tables into a single result set. This is useful when you need to display data from multiple tables in a single query result.
* **Performance**: Joins can sometimes be more efficient than subqueries, especially when dealing with large datasets. Using joins allows the database to optimize the query execution plan for better performance.

**Scenarios for Using Subqueries**:

* **Filtering Data**: Subqueries are useful for filtering data based on the result of another query. For example, you can use a subquery to retrieve orders that exceed a certain amount or meet specific criteria.
* **Conditional Logic**: Subqueries can be used to perform conditional logic within a SQL statement. For instance, you can use a subquery in the WHERE clause to filter rows based on a condition derived from another query's result.
* **Calculating Aggregates**: Subqueries can be used to calculate aggregates such as counts, sums, averages, etc., based on the result of a nested query.
* **Inline View**: Subqueries can be used as inline views to create temporary result sets that can be used as tables in the main query.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

1. what is the difference between SQL and NoSQL dB?
2. **Data Model**:
   * **SQL**: SQL databases use a structured data model based on tables with rows and columns. They enforce a schema that defines the structure of the data, including data types and relationships between tables.
   * **NoSQL**: NoSQL databases support a variety of data models, including document-oriented, key-value, wide-column, and graph databases. They are schema-less or have flexible schemas, allowing for dynamic and unstructured data.
3. **Query Language**:
   * **SQL**: SQL databases use the SQL query language for data manipulation, retrieval, and management. SQL provides a standard syntax for performing operations such as SELECT, INSERT, UPDATE, DELETE, and JOIN.
   * **NoSQL**: NoSQL databases may use different query languages or APIs depending on the data model. Some NoSQL databases have their own query languages optimized for specific use cases, while others support query APIs or DSLs (Domain-Specific Languages).
4. **Scalability**:
   * **SQL**: Traditional SQL databases are typically scaled vertically, meaning that they are scaled up by adding more resources (CPU, memory, storage) to a single server. Scaling out (adding more servers) can be challenging and may require complex replication or sharding strategies.
   * **NoSQL**: NoSQL databases are designed for horizontal scalability, allowing them to scale out easily by adding more nodes to a cluster. They can handle large volumes of data and high write throughput by distributing data across multiple nodes in a cluster.
5. **Consistency and Transactions**:
   * **SQL**: SQL databases typically provide strong consistency and support ACID (Atomicity, Consistency, Isolation, Durability) transactions. They ensure data integrity and guarantee that transactions are executed reliably.
   * **NoSQL**: NoSQL databases offer varying levels of consistency, including eventual consistency, strong consistency, and eventual consistency. Some NoSQL databases sacrifice consistency for scalability and availability, offering eventual consistency models that prioritize availability and partition tolerance over consistency.
6. **Use Cases**:
   * **SQL**: SQL databases are well-suited for applications that require complex queries, transactional integrity, and relational data modeling. They are commonly used in traditional relational database applications such as ERP systems, CRM systems, and financial applications.
   * **NoSQL**: NoSQL databases are ideal for applications with large-scale data requirements, high write throughput, and flexible data models. They are commonly used in web applications, real-time analytics, IoT (Internet of Things), and large-scale distributed systems.
7. **Examples**:
   * **SQL**: Examples of SQL databases include MySQL, PostgreSQL, Oracle Database, Microsoft SQL Server, and SQLite.
   * **NoSQL**: Examples of NoSQL databases include MongoDB, Cassandra, Redis, Couchbase, Amazon DynamoDB, and Neo4j.