SQL Interview Questions

SQL Server

Server=localhost\SQLEXPRESS;Database=master;Trusted\_Connection=True;

C:\Program Files\Microsoft SQL Server\160\Setup Bootstrap\Log\20241013\_102330

C:\SQL2022\Express\_ENU

C:\Program Files\Microsoft SQL Server\160\SSEI\Resources

-- run below commmand from command prompt

-- sqlcmd -S DESKTOP-JDQBCCU\SQLEXPRESS01 -E

select @@Version

go

**Here are some SQL interview questions suitable for an experienced developer:**

**1. Performance Optimization**

* How do you optimize slow-running queries in SQL?
* Explain the difference between clustered and non- clustered indexes. When would you use one over the other?
* What is a query execution plan, and how would you use it to optimize a query?
* How do you handle deadlocks in SQL Server?

**2. Database Design**

* How do you design a relational database for scalability and performance?
* What is database normalization, and when should you denormalize a database?
* Explain ACID properties in the context of a transactional database.
* How would you model a many-to-many relationship in a database?

**3. Advanced SQL Queries**

* Write a SQL query to find the second highest salary in a table.
* How do you perform a recursive query in SQL (e.g., for hierarchical data)?
* What is the difference between RANK(), DENSE\_RANK(), and ROW\_NUMBER() in SQL?

**4. Data Integrity & Constraints**

* What are the different types of constraints in SQL? Explain with examples.
* How do you ensure referential integrity in a database?
* How do foreign keys work, and what are cascading actions?

**5. Transactions & Locking**

* Explain how transactions work in SQL and the different isolation levels.
* What is the difference between pessimistic and optimistic locking?
* How would you implement a retry mechanism in case of transaction failures?

**6. Database Security**

* How do you implement database security to control user access?
* Explain the difference between authentication and authorization in SQL databases.
* How would you secure sensitive data in a database?

**7. Stored Procedures, Functions, and Triggers**

* What is the difference between a stored procedure and a function?
* How do you debug a stored procedure?
* Explain the use of triggers. When would you prefer to use them?

**8. Data Warehousing and ETL**

* How would you design a data warehouse for reporting and analytics?
* Explain the difference between OLTP and OLAP databases.
* What are the best practices for designing an ETL process?

**9. Handling Large Data Sets**

* How do you handle large datasets in SQL for better performance?
* What is partitioning in SQL, and when would you use it?
* Explain the use of window functions for working with large datasets.

**10. Joins and Subqueries**

* Explain the different types of joins in SQL and when to use each.
* What is the difference between a correlated subquery and a non-correlated subquery?
* How do you handle the performance of join operations in large databases?

Would you like to explore answers or detailed explanations for any of these?

**How do you optimize slow-running queries in SQL? explain**

Optimizing slow-running SQL queries is crucial for improving database performance and reducing resource usage. Below are several strategies to identify and optimize slow queries:

**1. Analyze the Query Execution Plan**

* **What it is:** The execution plan shows how the SQL engine executes the query, including the steps it takes, the use of indexes, table scans, and joins.
* **How it helps:** It helps identify bottlenecks like full table scans, improper join orders, or missing indexes.
* **How to use:** Most databases (e.g., MySQL, SQL Server, PostgreSQL) provide commands like EXPLAIN, EXPLAIN PLAN, or SHOW PLAN to display the execution plan. Look for operations like full table scans, and large sorts, or joins that can be optimized.

**2. Optimize Index Usage**

* **Add indexes where necessary:** Indexes can dramatically speed up query performance, especially for large tables.
  + **Single column indexes:** For queries filtering by one column, a simple index on that column helps.
  + **Composite indexes:** If queries filter or join on multiple columns, consider using composite (multi-column) indexes. Make sure the columns in the index are in the right order (based on the most restrictive filters).
* **Avoid over-indexing:** Too many indexes can slow down INSERT, UPDATE, and DELETE operations, so it’s essential to index only where needed.
* **Covering indexes:** Design indexes that can cover all the columns used in SELECT, WHERE, and JOIN clauses, so the query doesn’t need to access the table directly.

**3. Rewrite the Query**

* **Simplify complex queries:** Break down complex queries into smaller ones, especially if they contain multiple subqueries or nested selects.
* *Avoid \* in SELECT :* Always specify the columns you need, as selecting unnecessary columns increases the amount of data processed and returned.
* **Use joins effectively:** Prefer INNER JOIN over LEFT JOIN or RIGHT JOIN when possible since they usually involve fewer data comparisons.
* **Use WHERE instead of HAVING:** The WHERE clause filters rows before grouping, while HAVING filters after aggregation. In most cases, using WHERE improves performance.
* **Use EXISTS instead of IN:** EXISTS generally performs better than IN when dealing with subqueries that return a large dataset.

SELECT \* FROM orders WHERE customer\_id IN (SELECT customer\_id FROM customers);

vs.

SELECT \* FROM orders WHERE EXISTS (SELECT 1 FROM customers WHERE customers.customer\_id = orders.customer\_id);

**4. Optimize Joins**

* **Ensure indexed columns in joins:** Make sure the columns used in join conditions are indexed, especially on large tables.
* **Join order matters:** In some databases, the order in which tables are joined can impact performance. Smaller result sets should be processed first.
* **Avoid unnecessary joins:** Remove joins that aren't required or add complexity without adding value to the final result.

**5. Use Query Caching**

* **What it is:** Query caching stores the results of frequently run queries to avoid recalculating them every time.
* **How it helps:** It reduces the database load and response time for repeated queries.
* **Database options:** Many databases, like MySQL and PostgreSQL, offer built-in query caching or allow caching in application layers like Redis or Memcached.

**6. Optimize Subqueries**

* **Replace subqueries with joins:** Subqueries (especially correlated subqueries) can be slower because they may need to be executed multiple times. Rewriting them as joins often improves performance.

SELECT \* FROM employees WHERE employee\_id = (SELECT manager\_id FROM managers WHERE department = 'Sales');

vs.

SELECT e.\* FROM employees e JOIN managers m ON e.employee\_id = m.manager\_id WHERE m.department = 'Sales';

* **Optimize correlated subqueries:** If a correlated subquery must be used, make sure it returns a small dataset or is indexed appropriately.

**7. Limit the Data Returned**

* **Use LIMIT or TOP:** Restrict the number of rows returned, especially for queries where you only need a subset of the data.

SELECT \* FROM employees ORDER BY hire\_date DESC LIMIT 10;

* **Pagination:** For large datasets, use pagination techniques (OFFSET and LIMIT) to fetch data in chunks.

**8. Partition Large Tables**

* **What it is:** Partitioning splits a large table into smaller, more manageable parts, often based on ranges or specific column values (e.g., date ranges).
* **How it helps:** Queries can access only the relevant partitions instead of scanning the entire table, improving performance on large datasets.
* **Types:** Horizontal partitioning (splitting rows), vertical partitioning (splitting columns).

**9. Use Appropriate Data Types**

* **Correct data types:** Make sure that columns are using the most efficient data types. For example, using INT instead of BIGINT if the values don’t require a large range, or using VARCHAR with appropriate length rather than an excessively large one.
* **Avoid unnecessary conversions:** When filtering or joining, avoid casting or converting data types that prevent the use of indexes.

**10. Reduce Lock Contention**

* **Transaction management:** Keep transactions as short as possible. Long-running transactions can cause locks, which delay or block other queries.
* **Avoid unnecessary locks:** Use appropriate isolation levels and locking hints to minimize contention (e.g., using NOLOCK for read-only queries).

**11. Utilize Database-Specific Features**

* **Materialized views:** Some databases support materialized views, which store the result of a query and can be refreshed periodically. These are especially useful for expensive, frequently accessed queries.
* **Partitioning and sharding:** For very large datasets, partitioning tables across different physical disks or sharding them across multiple databases can improve performance.

**12. Batch Processing**

* **Batch updates and inserts:** Instead of performing multiple INSERT, UPDATE, or DELETE operations one by one, batch them together to reduce the number of database round trips.

INSERT INTO employees (id, name) VALUES (1, 'John'), (2, 'Jane');

Optimizing SQL queries requires analyzing specific use cases, understanding the database engine, and applying the right indexing, data structure, and optimization techniques.

**design a relational database for scalability and performance?**

Designing a relational database for scalability and performance involves making careful decisions about data modeling, indexing, partitioning, and system architecture. Here’s a step-by-step approach to creating a scalable and performant database:

**1. Data Model Design**

Start by defining an efficient data model that balances normalization and denormalization for performance and scalability.

**a. Normalization**

* **Goal:** Eliminate redundancy and ensure data integrity by organizing the database into normalized forms (e.g., 1NF, 2NF, 3NF).
* **Example:**
  + A normalized design might involve splitting customer and order details into separate tables (e.g., Customers, Orders, OrderDetails) to avoid repeating customer information across multiple rows.
* **Pros:** Reduces data duplication and ensures consistency.
* **Cons:** Excessive normalization may lead to complex joins and slow queries.

**b. Denormalization (Selective)**

* **Goal:** Reduce the number of joins for read-heavy operations by denormalizing some tables.
* **When to use:** Denormalization is appropriate when reads significantly outnumber writes and when performance is more important than storage efficiency.
* **Example:** Storing a total\_order\_value in the Orders table rather than recalculating it every time by summing individual OrderDetails.

**2. Indexing Strategy**

Proper indexing is essential for query performance, especially as the database grows.

**a. Primary Indexes**

* Ensure each table has a **primary key** that uniquely identifies records. This key should be indexed to enable fast lookups.
* **Example:** A CustomerID in the Customers table can serve as a primary index.

**b. Secondary (Non-Clustered) Indexes**

* Use **secondary indexes** on frequently queried columns, such as foreign keys or columns used in filtering, sorting, and joining operations.
* **Composite indexes** can be used for queries filtering on multiple columns. Ensure that the index order matches the query filter order for optimal performance.
* **Example:** Index the OrderDate and CustomerID in the Orders table to speed up queries like:

SELECT \* FROM Orders WHERE CustomerID = 1 AND OrderDate >= '2024-01-01';

**c. Avoid Over-Indexing**

* While indexes speed up reads, they slow down writes (INSERT, UPDATE, DELETE operations), so avoid indexing columns unnecessarily.
* Consider dropping indexes that aren't frequently used or relevant to current query patterns.

**3. Partitioning for Scalability**

Partitioning helps scale databases by splitting large tables into smaller, more manageable parts based on specific criteria, improving query performance on large datasets.

**a. Horizontal Partitioning (Sharding)**

* **What it is:** Splitting a table’s rows across multiple physical databases based on a shard key (e.g., customer ID ranges, geographic regions).
* **When to use:** Use horizontal partitioning for very large datasets that span multiple servers (e.g., 10+ million rows).
* **Example:** Customer data could be split by region, where customers in the US are stored in one shard, and customers in Europe are stored in another.

**b. Vertical Partitioning**

* **What it is:** Splitting a table’s columns into separate physical tables to reduce the amount of data read for certain queries.
* **When to use:** For wide tables with many columns, vertical partitioning can improve performance by grouping frequently queried columns together.
* **Example:** Split a Customers table into CustomersBasic (with basic info like CustomerID, Name, and Email) and CustomersDetails (with additional info like Address, Phone, etc.).

**c. Range Partitioning**

* Split large tables based on a range of values, often used for date-based data.
* **Example:** Split an Orders table into partitions by year, where all orders from 2023 go into one partition and orders from 2024 go into another. This improves performance when querying a specific range of data (e.g., all orders from 2024).

**4. Replication for High Availability and Scalability**

Database replication involves maintaining multiple copies of your data across different servers to ensure high availability, load distribution, and fault tolerance.

**a. Master-Slave (Primary-Replica) Replication**

* **What it is:** A master database handles all writes, while read queries can be distributed across multiple read replicas.
* **Use case:** If the application has high read traffic, replication can be used to scale horizontally by distributing reads to replicas.
* **Example:** A system with a single master for INSERT, UPDATE, DELETE queries and multiple replicas for SELECT queries.

**b. Multi-Master Replication**

* **What it is:** Multiple databases can handle both reads and writes, and data is synchronized between them.
* **Use case:** Useful in distributed systems where you need low-latency writes in geographically dispersed locations.

**5. Caching Strategy**

Reduce the load on the database by caching frequently accessed data.

**a. In-Memory Caching**

* Use in-memory caching systems like **Redis** or **Memcached** to store results of frequent or expensive queries.
* **Example:** Cache the results of a query that retrieves product details or user session information, reducing the number of database hits.

**b. Query Caching**

* Some relational databases (like MySQL) support query caching, where the results of a query are stored and returned when the same query is executed again without needing to hit the database.

**6. Optimize Data Types**

Choose appropriate data types for each column to minimize storage and improve performance.

* **Use fixed-width data types when possible:** For example, use CHAR(10) for a fixed-length field like a phone number instead of VARCHAR(255).
* **Avoid unnecessarily large data types:** Use INT instead of BIGINT when the range of values allows it.
* **Example:** In a table tracking orders, instead of using DATETIME for a date column when only the date is necessary, use DATE to save space and improve performance.

**7. Handling Large Data Loads**

Use bulk operations and batching to handle large data inserts and updates efficiently.

* **Bulk Inserts:** Use bulk insert operations to insert large amounts of data efficiently. Avoid inserting rows one by one.

INSERT INTO Orders (OrderID, CustomerID, OrderDate) VALUES (1, 1, '2024-01-01'), (2, 2, '2024-01-02');

* **Batch Processing:** For very large operations, process data in smaller batches to avoid locking tables for too long.

**8. Concurrency Control and Isolation Levels**

Design your database to handle concurrent operations while maintaining data consistency.

**a. Isolation Levels**

* Choose the right **transaction isolation level** (Read Uncommitted, Read Committed, Repeatable Read, Serializable) based on the system’s consistency and performance needs.
* **Example:** For high-throughput systems with low contention, Read Committed or Read Uncommitted may offer better performance, while highly sensitive transactions may require Serializable isolation.

**b. Optimistic vs. Pessimistic Locking**

* Use **optimistic locking** in low-contention environments where you expect fewer conflicts, reducing the overhead of locking.
* **Pessimistic locking** is used in high-contention environments where data integrity is more important than performance.

**9. Use Materialized Views (When Needed)**

Materialized views store the result of a query physically and can be refreshed periodically. They are especially useful for complex, frequently executed queries.

* **Example:** If you're running an analytics system, create a materialized view to aggregate monthly sales data rather than recalculating it every time a query runs.

**10. Regular Database Maintenance**

* **Rebuild or reorganize indexes:** Regularly defragment and reorganize indexes to maintain performance, especially in high-write environments.
* **Update statistics:** Ensure that your database’s query optimizer has the most up-to-date statistics to generate efficient execution plans.
* **Database backups:** Plan for backups without locking the database for long periods (e.g., through online backups, snapshotting).

**Example Design:**

CREATE TABLE Customers (

CustomerID INT PRIMARY KEY,

Name VARCHAR(100),

Email VARCHAR(100) UNIQUE,

Region CHAR(2),

CreatedAt TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

CREATE TABLE Orders (

OrderID INT PRIMARY KEY,

CustomerID INT,

OrderDate DATE,

TotalAmount DECIMAL(10, 2),

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)

) PARTITION BY RANGE (OrderDate) (

PARTITION p2024 VALUES LESS THAN (TO\_DATE('2024-01-01', 'YYYY-MM-DD')),

PARTITION p2025 VALUES LESS THAN (TO\_DATE('2025-01-01', 'YYYY-MM-DD'))

);

CREATE INDEX idx\_orders\_customerid ON Orders (CustomerID);

This design includes normalized tables, range partitioning on Orders, indexing on frequently queried fields (CustomerID), and constraints to ensure data integrity, providing a foundation for scalability and performance.

**How do you handle deadlocks in SQL Server**

Deadlocks in SQL Server occur when two or more transactions are waiting for each other to release locks, causing a situation where none of them can proceed. SQL Server has a deadlock detection mechanism that automatically detects deadlocks and terminates one of the transactions, allowing the other to continue. The terminated transaction is rolled back, and the application must handle it.

To handle deadlocks, you can take several proactive and reactive steps to prevent, detect, and recover from deadlocks. Here’s how you can manage deadlocks in SQL Server:

**1. Minimize Deadlock Risk with Best Practices**

**a) Ensure Consistent Locking Order**

* Always access objects in the same order in all transactions. For example, if two transactions access table A first and then table B, they should follow this order consistently to prevent deadlocks.
* Example:

-- Consistent Locking Order

BEGIN TRANSACTION

UPDATE TableA SET ... WHERE ...

UPDATE TableB SET ... WHERE ...

COMMIT TRANSACTION

**b) Keep Transactions Short and Fast**

* Avoid long-running transactions that hold locks for a long time. The longer a transaction runs, the higher the chance it will block other transactions and cause deadlocks.
* Commit transactions as quickly as possible after the necessary work is done.

**c) Use Lower Isolation Levels**

* Consider using lower isolation levels like READ COMMITTED or READ COMMITTED SNAPSHOT to reduce locking contention. However, balance this with the risk of allowing dirty or non-repeatable reads.
* Example of using READ COMMITTED SNAPSHOT:

ALTER DATABASE YourDatabase SET READ\_COMMITTED\_SNAPSHOT ON;

**d) Use NOLOCK or READ UNCOMMITTED (With Caution)**

* In some cases, you can use the NOLOCK hint (or READ UNCOMMITTED isolation level) to avoid locking issues, but this allows dirty reads, so it should be used with caution.
* Example:

sql

Copy code

SELECT \* FROM Employees WITH (NOLOCK);

**e) Optimize Indexing**

* Create appropriate indexes to reduce the number of rows affected by queries. Fewer rows involved in a transaction reduce the chance of deadlocks.
* Index optimization helps ensure that SQL Server can lock fewer rows or pages instead of locking entire tables.

**2. Handling Deadlock Errors Programmatically**

SQL Server automatically selects a "deadlock victim" (one of the transactions) to terminate when a deadlock occurs. If your transaction is chosen as the victim, you need to handle this in your application by retrying the transaction.

You can handle this by catching deadlock exceptions and retrying the transaction a few times before failing.

**Example in T-SQL (Retry Logic):**

DECLARE @retry INT = 3; -- Number of retry attempts

WHILE @retry > 0

BEGIN

BEGIN TRY

BEGIN TRANSACTION;

-- Your transactional code goes here

-- Example: UPDATE Employees SET Salary = Salary \* 1.1 WHERE Department = 'Sales';

COMMIT TRANSACTION;

BREAK; -- Exit loop if successful

END TRY

BEGIN CATCH

IF ERROR\_NUMBER() = 1205 -- Deadlock error code

BEGIN

-- Rollback if deadlock occurred

ROLLBACK TRANSACTION;

SET @retry = @retry - 1; -- Decrease retry counter

END

ELSE

BEGIN

-- If it's not a deadlock, raise the error

THROW;

END

END CATCH;

END

* **Explanation**: The script retries the transaction up to three times if a deadlock error occurs (ERROR\_NUMBER() = 1205). Each time a deadlock happens, it rolls back the transaction and retries. If it fails after three attempts, it stops retrying.

**3. Deadlock Monitoring and Detection**

SQL Server provides tools to monitor and detect deadlocks, which can help you identify which transactions or queries are causing the deadlocks.

**a) Enable Deadlock Trace Flags**

* Enable trace flag 1222 or 1204 to log detailed deadlock information in the SQL Server error log.
* **Trace Flag 1222**: Provides more detailed information on the deadlock graph, including the processes and resources involved.

DBCC TRACEON (1222, -1); -- Enables trace flag 1222 globally

**b) Use Extended Events or SQL Profiler**

* SQL Server’s Extended Events and SQL Profiler tools can be used to capture deadlock information and generate deadlock graphs.
* Example using Extended Events to monitor deadlocks:

CREATE EVENT SESSION MonitorDeadlocks ON SERVER

ADD EVENT sqlserver.deadlock\_graph

ADD TARGET package0.event\_file (SET filename = 'C:\deadlocks.xel')

WITH (MAX\_MEMORY = 4096 KB, EVENT\_RETENTION\_MODE = ALLOW\_SINGLE\_EVENT\_LOSS, MAX\_DISPATCH\_LATENCY = 30 SECONDS, MAX\_EVENT\_SIZE = 0 KB, MEMORY\_PARTITION\_MODE = NONE, TRACK\_CAUSALITY = OFF, STARTUP\_STATE = OFF);

ALTER EVENT SESSION MonitorDeadlocks ON SERVER STATE = START;

* You can then view the captured deadlock information using SQL Server Management Studio or query the .xel file.

**4. Use SET DEADLOCK\_PRIORITY to Influence Victim Selection**

SQL Server decides which transaction to kill as the deadlock victim based on the cost of rollback. You can influence this decision by setting the deadlock priority of a session to LOW, NORMAL, or HIGH.

sql

Copy code

SET DEADLOCK\_PRIORITY LOW; -- Lower priority, this transaction will be chosen as the deadlock victim

**Summary**

* **Prevention**: Use consistent locking order, minimize transaction duration, use appropriate isolation levels, and optimize queries and indexes.
* **Handling**: Implement retry logic for deadlock victims by catching deadlock errors (ERROR\_NUMBER() = 1205) and retrying the transaction.
* **Monitoring**: Use SQL Server tools like trace flags, Extended Events, or SQL Profiler to detect and troubleshoot deadlocks.

**what is Optimistic and Pessimistic Locking in database**

**Optimistic** and **Pessimistic** locking are two common strategies used in databases to manage concurrent access to data while ensuring consistency and integrity, especially in multi-user environments.

**1. Pessimistic Locking**

Pessimistic locking assumes that conflicts between transactions are likely to happen, so it locks the data immediately to prevent other transactions from making changes to the same data until the current transaction is complete.

**How It Works:**

* When a transaction reads or updates data, it acquires a lock (shared or exclusive) on the record or table.
* Other transactions are blocked from accessing or modifying the locked data until the lock is released (usually at the end of the transaction).
* There are two types of locks:
  + **Shared Lock:** Allows other transactions to read the data but not modify it.
  + **Exclusive Lock:** Blocks both read and write operations from other transactions.

**Use Case:**

* Pessimistic locking is useful in environments with **high contention**, where many users are trying to modify the same data simultaneously, and you expect **frequent conflicts**.
* It is common in **banking systems** or **inventory management** systems where data integrity is critical, and conflicting transactions must be avoided.

**Example:**

In an inventory system where two users are trying to modify the same product quantity:

1. **User 1** initiates a transaction to update the quantity of a product.
2. A pessimistic lock is placed on the product record.
3. **User 2** attempts to update the same product but is blocked until **User 1** finishes the transaction and releases the lock.
4. Once **User 1** commits or rolls back, **User 2's** transaction can proceed.

**Downsides:**

* **Performance impact:** Because locks block other transactions, it can lead to **deadlocks** or cause transactions to wait, reducing system throughput.
* **Reduced concurrency:** Other users are blocked from accessing the locked data, even for read operations (if an exclusive lock is in place).

**2.** **Optimistic Locking**

Optimistic locking assumes that conflicts between transactions are rare, so it doesn’t lock the data right away. Instead, it checks for conflicts at the end of the transaction when the update is about to be committed.

**How It Works:**

* The system allows multiple transactions to read the same data simultaneously.
* Before updating the data, the system checks if the data has been modified by another transaction during the read-to-update phase.
* If the data has changed, the transaction will fail, and you must retry or handle the conflict. If the data hasn't changed, the update proceeds.

**Use Case:**

* Optimistic locking is ideal in **low-contention** environments, where conflicts are unlikely and locking data upfront would unnecessarily hurt performance.
* It is often used in **web applications** where users are reading and editing data independently, and conflicts are rare.

**Example:**

In an online editing system where multiple users can edit the same document:

1. **User 1** reads a record (e.g., document version 1).
2. **User 2** reads the same record (version 1).
3. **User 1** makes changes and submits the update. The system checks if the version of the record is still 1. If it is, the update proceeds, and the version is incremented to 2.
4. **User 2** attempts to submit changes. Since the version is no longer 1 (it's now 2), the system detects a conflict and rejects **User 2’s** changes.

This approach can be implemented by adding a **version number** or **timestamp** to the data row. When a transaction updates a row, it checks that the version or timestamp hasn’t changed since the row was first read.

**Downsides:**

* **Retry logic:** If a conflict occurs, the transaction fails, and you must retry or handle the conflict, which can add complexity.
* **Potential wasted work:** Since the conflict is detected late (at commit), users or systems might spend time working on changes only to have them rejected.

**What are ACID properties;**

ACID properties are a set of four key principles that ensure reliable processing of database transactions in a transactional database system. ACID stands for **Atomicity**, **Consistency**, **Isolation**, and **Durability**. These properties ensure data integrity, reliability, and fault tolerance, especially in systems that require multiple, concurrent transactions.

Here’s a detailed explanation of each property in the context of a transactional database:

**1. Atomicity**

* **Definition**: Atomicity ensures that a transaction is **all or nothing**. Either all the operations within the transaction are executed successfully, or none of them are applied. If any part of the transaction fails, the entire transaction is rolled back, and the database is left unchanged.
* **Example**: Suppose you are transferring money between two bank accounts:

sql

Copy code

BEGIN TRANSACTION;

UPDATE Accounts SET Balance = Balance - 100 WHERE AccountID = 1; -- Withdraw from Account 1

UPDATE Accounts SET Balance = Balance + 100 WHERE AccountID = 2; -- Deposit into Account 2

COMMIT;

If the second update (deposit into Account 2) fails for some reason (e.g., a system crash or violation of constraints), **atomicity** ensures that the first update (withdrawal from Account 1) is rolled back, so no money is lost.

* **Real-life analogy**: Atomicity is like a checklist for an airplane takeoff—either all checks are successfully completed and the plane takes off, or none are completed and the plane remains grounded.

**2. Consistency**

* **Definition**: Consistency ensures that a transaction brings the database from one valid state to another, adhering to all defined rules, constraints, and triggers. The data must always remain in a valid state, both before and after the transaction.
* **Example**: Consider the rule that an account balance cannot go negative. If a transaction attempts to withdraw more money than available, consistency ensures the transaction fails, maintaining the rule:

sql

Copy code

UPDATE Accounts SET Balance = Balance - 100 WHERE AccountID = 1 AND Balance >= 100;

If the balance in AccountID = 1 is less than $100, the update will fail, ensuring that the balance never becomes negative, preserving consistency.

* **Real-life analogy**: Consistency is like maintaining proper rules in a sports game. If a rule violation occurs, the play is stopped, and the game is reset to its correct state.

**3. Isolation**

* **Definition**: Isolation ensures that the execution of a transaction is independent of other concurrent transactions. Intermediate states of a transaction should not be visible to other transactions until the transaction is committed. Depending on the isolation level, transactions are prevented from interfering with each other, reducing issues like dirty reads, non-repeatable reads, and phantom reads.
* **Example**: Consider two transactions:
  + **Transaction 1**: Transfers $100 from Account A to Account B.
  + **Transaction 2**: Reads the balance of Account A while the transfer is in progress. Depending on the isolation level, **Transaction 2** will either see the initial balance of Account A (before the transfer starts) or the final balance after the transfer is complete. It won’t see the intermediate state where $100 has been deducted from Account A but not yet added to Account B.
* **Isolation Levels**:
  + **Read Uncommitted**: Transactions can see uncommitted changes of other transactions (risk of dirty reads).
  + **Read Committed**: Transactions can only see committed changes.
  + **Repeatable Read**: Transactions see consistent data throughout their execution.
  + **Serializable**: Transactions are executed as if they are run one after another (highest level of isolation).
* **Real-life analogy**: Isolation is like waiting in line at a bank teller. Even though multiple people may be in line, each customer is served individually without interference from the others.

**4. Durability**

* **Definition**: Durability ensures that once a transaction is committed, its changes are permanently saved in the database, even in the event of a system crash or failure. The system must guarantee that committed data is stored reliably and can be recovered after a failure.
* **Example**: Suppose a transaction commits a payment of $500. Even if the server crashes immediately after the commit, the payment is guaranteed to be recorded in the database once the system is restored. This is achieved through mechanisms like write-ahead logging, where changes are first written to a durable log before being applied to the database.
* **Real-life analogy**: Durability is like writing with a permanent marker—once something is written, it can't be erased, even if the marker runs out of ink or the paper is lost temporarily.

**Summary of ACID Properties:**

| **Property** | **Description** | **Example** | **Benefit** |
| --- | --- | --- | --- |
| **Atomicity** | Ensures all operations in a transaction are completed or none are. | If transferring money between accounts fails mid-way, the entire transaction is rolled back. | Guarantees completeness of transactions. |
| **Consistency** | Ensures a transaction leaves the database in a valid state. | Enforcing rules like "account balance cannot be negative." | Preserves data integrity according to defined constraints. |
| **Isolation** | Ensures transactions don’t interfere with each other. | A bank transfer does not affect another concurrent query checking account balance. | Maintains correctness in concurrent environments. |
| **Durability** | Ensures committed transactions are permanently saved. | Data persists even after a system crash. | Guarantees data is safe once the transaction is complete. |

**ACID in Action:**

For example, let’s say you're processing an e-commerce order, which involves multiple operations like reducing the product quantity, charging the customer, and updating the order status:

1. **Atomicity**: If any of the steps fail (e.g., payment fails), the entire transaction is rolled back.
2. **Consistency**: After the transaction, the product inventory, payment system, and order status will be consistent with the business rules.
3. **Isolation**: Other users won’t see partial changes (e.g., reduced inventory but no payment yet).
4. **Durability**: Once the order is placed, the system guarantees that the changes (order details, payment confirmation) are safely stored.

Together, the ACID properties ensure that transactions are processed reliably, even in the presence of failures or concurrent activity, making them a cornerstone of robust database management systems.

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

The primary difference between a **stored procedure** and a **function** in a database lies in their purpose, behavior, and how they are used. While both are used to encapsulate a set of SQL statements for reuse, there are some important distinctions between them in terms of functionality, return types, side effects, and usage.

Here’s a detailed comparison of the two:

**1. Purpose**

* **Stored Procedure**: A stored procedure is typically used to perform a sequence of operations that may include querying or modifying the database, such as inserting, updating, or deleting records. It can encapsulate complex business logic and can perform multiple tasks within a single call.
* **Function**: A function is primarily used to compute a value and return it. It is typically used for calculations or data retrieval and is designed to be called within SQL queries, returning a result based on input parameters.

**2. Return Value**

* **Stored Procedure**: A stored procedure **may or may not return a value**. If it does, it usually returns the result using OUT or INOUT parameters or a result set (in some databases, procedures can return multiple result sets). However, procedures do not typically return a scalar value directly.
* **Function**: A function **must return a value** (either a scalar value, table, or a complex type). Functions are often used to return a single value (e.g., INT, VARCHAR) or a table and can be used in SELECT statements.

Example:

* + **Procedure**: No requirement to return a value, but may use output parameters.
  + **Function**: Must return a specific value (scalar, table, etc.).

**3. Usage in SQL Statements**

* **Stored Procedure**: Procedures **cannot** be directly used in SQL statements (e.g., in a SELECT, WHERE, or JOIN clause). They are called using the CALL or EXEC command.

Example:

CALL UpdateCustomerBalance(100, 'C001');

* **Function**: Functions **can** be used within SQL statements, such as SELECT, WHERE, or JOIN clauses, as part of expressions.

Example:

SELECT CalculateDiscount(100, 'Regular') FROM Orders;

**4. Side Effects**

* **Stored Procedure**: A stored procedure can have **side effects** such as modifying database state (inserting, updating, or deleting data). It can include multiple SQL statements and perform operations that change the data.
* **Function**: Functions are generally **side-effect free** in most databases. They are typically restricted to operations that do not modify database state. In some databases (like PostgreSQL), there are exceptions, but in most cases, functions are not allowed to perform INSERT, UPDATE, or DELETE operations.

**5. Transaction Control**

* **Stored Procedure**: Procedures can include **transaction control statements** such as BEGIN, COMMIT, and ROLLBACK. This allows a procedure to manage transactions explicitly and ensure that certain steps are rolled back in case of failure.
* **Function**: Functions typically **cannot** control transactions. You cannot commit or roll back within a function, as functions are expected to be part of a broader transaction.

**6. Input Parameters**

* **Stored Procedure**: Procedures can accept **input, output, and input-output (IN, OUT, INOUT)** parameters. This allows a procedure to return multiple values through its output parameters.

Example:

sql

Copy code

CREATE PROCEDURE GetEmployeeDetails(IN employeeID INT, OUT employeeName VARCHAR(100))

BEGIN

SELECT name INTO employeeName FROM Employees WHERE id = employeeID;

END;

* **Function**: Functions typically only accept **input (IN)** parameters and must return a value directly. Functions do not support output parameters.

Example:

sql

Copy code

CREATE FUNCTION GetEmployeeSalary(employeeID INT) RETURNS DECIMAL(10, 2)

BEGIN

RETURN (SELECT salary FROM Employees WHERE id = employeeID);

END;

**7. Return Type**

* **Stored Procedure**: Procedures do not need to return a value, but if they do, they use output parameters or may return a result set (like a table).
* **Function**: Functions must return a value and can return a scalar value, a table, or a complex data type, depending on the definition.

**8. Execution**

* **Stored Procedure**: Procedures are executed using the CALL or EXEC command, often outside of a query.

Example:

sql

Copy code

EXEC UpdateEmployeeSalary(1001, 50000);

* **Function**: Functions are executed as part of a SQL expression and can be used in various SQL clauses.

Example:

sql

Copy code

SELECT GetTotalSales('2024-01-01', '2024-12-31');

**9. Error Handling**

* **Stored Procedure**: Procedures often provide built-in error handling mechanisms, such as TRY-CATCH blocks, making it easier to handle exceptions during execution.
* **Function**: Error handling in functions is generally more limited. Functions are typically expected to handle computations rather than transactional logic, so they do not provide advanced error handling like procedures.

**10. Performance Considerations**

* **Stored Procedure**: Procedures are generally used for operations that involve complex business logic, batch processing, or data modification, and they may impact performance when they involve heavy I/O or multiple operations.
* **Function**: Functions are optimized for performance, especially when they are designed for computations or retrieving data, as they can be used inline within queries.

**Explain the use of triggers. When would you prefer to use them?**

**Triggers** in a database are special types of stored procedures that automatically execute or "fire" in response to certain events on a specific table or view. These events typically involve data modifications such as **INSERT**, **UPDATE**, or **DELETE** operations. Triggers are used to enforce business rules, maintain data integrity, and perform automated tasks within the database without requiring explicit application logic.

**Use of Triggers**

Triggers can be used for a variety of purposes, such as:

1. **Auditing**: Automatically log changes made to a table for auditing purposes. This could be used to track who modified data, when it was modified, and what the changes were.
   * Example: Keep a history of changes in an Employees table by logging every modification in an Employee\_Audit table.
2. **Enforcing Business Rules**: Ensure that certain business rules are adhered to before changes are committed to the database. For example, ensuring that a product cannot be ordered if it’s out of stock.
   * Example: Prevent an order from being placed if the stock level in the Inventory table is below a certain threshold.
3. **Data Validation**: Ensure that data modifications meet specific validation criteria before they are applied. For example, preventing updates to a column if certain conditions are not met.
   * Example: Automatically ensure that the salary of an employee doesn’t exceed a predefined limit.
4. **Cascading Changes**: Automatically propagate changes from one table to another. For example, if a record is deleted from a parent table, related records in child tables can be automatically deleted (cascade delete).
   * Example: When deleting a Customer, automatically delete all related records in the Orders table.
5. **Synchronizing Tables**: Automatically copy or synchronize data between tables when certain events occur. This could be useful for replicating data or maintaining denormalized data.
   * Example: Automatically update a TotalSales column in a Summary table whenever a new sale is recorded in the Sales table.
6. **Maintaining Derived or Aggregated Values**: Automatically update aggregated or derived values in a table when underlying data changes. This helps ensure that summary data is always up to date.
   * Example: Update a TotalOrderAmount field in the Customers table whenever a new order is placed or an existing order is updated.

**Example of a Trigger**

Let’s consider a trigger that logs all changes made to the Employees table into an audit table called Employee\_Audit:

CREATE TRIGGER trg\_employee\_update

AFTER UPDATE ON Employees

FOR EACH ROW

BEGIN

INSERT INTO Employee\_Audit (EmployeeID, OldSalary, NewSalary, ModifiedDate)

VALUES (OLD.EmployeeID, OLD.Salary, NEW.Salary, NOW());

END;

This trigger fires **after an update** on the Employees table and inserts the old and new salary values into the Employee\_Audit table.

**Types of Triggers**

1. **Before Trigger**:
   * Fires **before** the triggering event (such as an insert, update, or delete) takes place.
   * Typically used for validation or modifying data before it is committed.
   * Example: Ensure that a value being inserted into a column meets a certain condition.
2. **After Trigger**:
   * Fires **after** the triggering event has been completed successfully.
   * Typically used for actions such as logging or auditing.
   * Example: Log the changes after a record has been updated.
3. **Instead of Trigger**:
   * Used to replace the triggering action with a custom action. These are typically used on views where the underlying table cannot be modified directly.
   * Example: Perform a custom insert operation when a user tries to insert into a view.

**When to Prefer Triggers**

Triggers should be used in the following scenarios:

1. **Automatic Enforcement of Business Rules**: When you want to ensure that certain business rules are enforced consistently across the application, triggers can provide a centralized mechanism for doing so. For example, preventing salary increases above a specific threshold.
2. **Data Integrity Across Applications**: When multiple applications interact with the database, you might not have full control over the application logic. In such cases, using triggers to maintain integrity and enforce rules directly at the database level ensures consistency.
3. **Auditing Changes**: If there is a need to track changes in certain tables, triggers can be useful for automatically logging changes (such as updates or deletions) without relying on application logic.
4. **Cascading Actions**: When changes in one table must lead to automatic changes in related tables, such as cascading deletes or updates. This avoids the need for manually handling relationships in every part of the application.
5. **Maintaining Derived Values or Aggregates**: If you have calculated or derived values that depend on changes in other tables (e.g., total sales in a customer table), triggers can help keep those values in sync without requiring manual updates.
6. **Complex Validation**: When the validation logic involves multiple tables or complex conditions, triggers can be an effective way to ensure data consistency before changes are committed.

**When to Avoid Triggers**

While triggers can be useful, they should be avoided in certain scenarios due to their complexity and potential impact on performance:

1. **Performance Concerns**: Triggers can slow down data modification operations (INSERT, UPDATE, DELETE) because they introduce additional processing. This overhead can become significant when working with large datasets or highly transactional systems.
2. **Hard-to-Debug Logic**: Triggers are executed implicitly, which can make debugging and understanding the flow of logic in the database more difficult. This can lead to unexpected behavior, especially when multiple triggers interact or are not well documented.
3. **Application-Level Logic**: Some business rules and validations are better implemented in the application layer, especially when they require user-specific behavior or complex workflows that are easier to manage outside the database.
4. **Recursive Trigger Calls**: Triggers can sometimes lead to recursive calls (i.e., a trigger fires another trigger, leading to an infinite loop), which can be hard to detect and manage unless specific safeguards are in place.

**Index Types**

**5. Clustered Index**

* **Description**: A clustered index defines the physical order of data in the table. There can be only **one** clustered index per table because the data rows are stored in the same order as the index. In most cases, the primary key is a clustered index by default.
* **Use cases**: Efficient for range queries and queries that retrieve rows based on the ordering of the indexed column.
* **Drawback**: Since it dictates the table’s physical storage, inserting new records in between existing rows can be slow due to reordering.
* **Example**:

CREATE CLUSTERED INDEX idx\_employee\_id ON Employees(ID);

**6. Non-Clustered Index**

* **Description**: A non-clustered index does not affect the physical order of the data in the table. Instead, it creates a separate structure that stores the indexed columns along with pointers to the data rows. A table can have multiple non-clustered indexes.
* **Use cases**: Ideal for speeding up queries where exact lookups or ranges are needed, but the ordering of data doesn’t matter.
* **Example**:

CREATE INDEX idx\_employee\_salary ON Employees(Salary);

**7. Unique Index**

* **Description**: A unique index enforces uniqueness on the values in the indexed column(s). It ensures that no two rows can have the same value in the indexed columns. If a column is defined with a UNIQUE constraint, a unique index is automatically created.
* **Use cases**: Use when you need to enforce uniqueness on a column, such as email or username.
* **Example**:

CREATE UNIQUE INDEX idx\_employee\_email ON Employees(Email);

**8. Composite Index (Multi-column Index)**

* **Description**: A composite index is an index on two or more columns. It speeds up queries that filter based on multiple columns. However, the order of the columns in the index matters.
* **Use cases**: Useful when your queries filter on more than one column (e.g., searching by first\_name and last\_name together).
* **Drawback**: The order of columns is important; it is most effective when the queries use the columns in the same order as the index.
* **Example**:

CREATE INDEX idx\_employee\_name\_salary ON Employees(Name, Salary);

Oracle Sql Queries : -

<https://www.complexsql.com/complex-sql-queries-examples-with-answers/>

Q5. What are joins in SQL?

A JOIN clause is used to combine rows from two or more tables, based on a related column between them. It is used to merge two tables or retrieve data from there. There are 4 types of joins, as you can refer to below:

Chart, bubble chart

Description automatically generated

Inner join: [Inner Join in SQL](https://www.edureka.co/blog/sql-joins-types#_blank) is the most common type of join. It is used to return all the rows from multiple tables where the join condition is satisfied.

Left Join:  Left Join in SQL is used to return all the rows from the left table but only the matching rows from the right table where the join condition is fulfilled.

Right Join: Right Join in SQL is used to return all the rows from the right table but only the matching rows from the left table where the join condition is fulfilled.

Full Join: Full join returns all the records when there is a match in any of the tables. Therefore, it returns all the rows from the left-hand side table and all the rows from the right-hand side table.

Q6. What is the difference between CHAR and VARCHAR2 datatype in SQL?

Both Char and Varchar2 are used for characters datatype but varchar2 is used for character strings of variable length whereas Char is used for strings of fixed length. For example, char(10) can only store 10 characters and will not be able to store a string of any other length whereas varchar2(10) can store any length i.e 6,8,2 in this variable.

Q7. What is a Primary key?

A screenshot of a computer

Description automatically generated with medium confidenceA [Primary key in SQL](https://www.edureka.co/blog/primary-key-in-sql/#_blank) is a column (or collection of columns) or a set of columns that uniquely identifies each row in the table.

Uniquely identifies a single row in the table

Null values not allowed

Example- In the Student table, Stu\_ID is the primary key.

Q8. What are Constraints?

[Constraints in SQL](https://www.edureka.co/blog/sql-constraints/#_blank) are used to specify the limit on the data type of the table. It can be specified while creating or altering the table statement. The sample of constraints are:

NOT NULL

CHECK

DEFAULT

UNIQUE

PRIMARY KEY

FOREIGN KEY

Q9. What is the difference between DELETE and TRUNCATE statements?

|  |  |
| --- | --- |
| DELETE vs TRUNCATE | |
| DELETE | TRUNCATE |
| Delete command is used to delete a row in a table. | Truncate is used to delete all the rows from a table. |
| You can rollback data after using delete statement. | You cannot rollback data. |
| It is a DML command. | It is a DDL command. |
| It is slower than truncate statement. | It is faster. |

Q10. What is a Unique key?

Uniquely identifies a single row in the table.

Multiple values allowed per table.

Null values allowed.

Apart from this SQL Interview Questions blog, if you want to get trained from professionals on this technology, you can opt for [structured training from edureka!](https://www.edureka.co/sql-essentials-training#_blank)

Q11. What is a Foreign key in SQL?

Foreign key maintains referential integrity by enforcing a link between the data in two tables.

The foreign key in the child table references the primary key in the parent table.

The [foreign key constraint](https://www.edureka.co/blog/foreign-key-sql/#_blank) prevents actions that would destroy links between the child and parent tables.

**Q12.** **What do you mean by data integrity?**

Data Integrity defines the accuracy as well as the consistency of the data stored in a database. It also defines integrity constraints to enforce business rules on the data when it is entered into an application or a database.

**Q13.** **What is the difference between clustered and non-clustered index in SQL?**

The differences between the clustered and non clustered index in SQL are :

Clustered index is used for easy retrieval of data from the database and its faster whereas reading from non clustered index is relatively slower.

Clustered index alters the way records are stored in a database as it sorts out rows by the column which is set to be clustered index whereas in a non clustered index, it does not alter the way it was stored but it creates a separate object within a table which points back to the original table rows after searching.

One table can only have one clustered index whereas it can have many non clustered index.

**Q18. What is an Index?**

An index refers to a performance tuning method of allowing faster retrieval of records from the table. An index creates an entry for each value and hence it will be faster to retrieve data.

**Q19. Explain different types of index in SQL.**

There are three [types of index in SQL](https://www.edureka.co/blog/index-in-sql/#_blank) namely:

**Unique Index:**

This index does not allow the field to have duplicate values if the column is unique indexed. If a primary key is defined, a unique index can be applied automatically.

**Clustered Index:**

This index reorders the physical order of the table and searches based on the basis of key values. Each table can only have one clustered index.

**Non-Clustered Index:**

Non-Clustered Index does not alter the physical order of the table and maintains a logical order of the data. Each table can have many nonclustered indexes.

https://www.giantstride.gr/en/sql-indexing-part2/

**Q23. What is the ACID property in a database?**

ACID stands for Atomicity, Consistency, Isolation, Durability. It is used to ensure that the data transactions are processed reliably in a database system.

**Atomicity:** Atomicity refers to the transactions that are completely done or failed where transaction refers to a single logical operation of a data. It means if one part of any transaction fails, the entire transaction fails and the database state is left unchanged.

**Consistency:** Consistency ensures that the data must meet all the validation rules. In simple words,  you can say that your transaction never leaves the database without completing its state.

**Isolation:** The main goal of isolation is concurrency control.

**Durability:** Durability means that if a transaction has been committed, it will occur whatever may come in between such as power loss, crash or any sort of error.

https://www.geeksforgeeks.org/acid-properties-in-dbms/

**Q24. What do you mean by “Trigger” in SQL?**

[Trigger in SQL](https://www.edureka.co/blog/triggers-in-sql/#_blank) is a special type of stored procedures that are defined to execute automatically in place or after data modifications. It allows you to execute a batch of code when an insert, update or any other query is executed against a specific table.

**Q56.** **What is a Stored Procedure?**

A Stored Procedure is a function which consists of many SQL statements to access the database system. Several SQL statements are consolidated into a stored procedure and execute them whenever and wherever required which saves time and avoid writing code again and again.

**Q57. List some advantages and disadvantages of Stored Procedure?**

**Advantages:**

A Stored Procedure can be used as a modular programming which means create once, store and call for several times whenever it is required. This supports faster execution. It also reduces network traffic and provides better security to the data.

Disadvantage:

The only disadvantage of Stored Procedure is that it can be executed only in the database and utilizes more memory in the database server.

**Type of SQL Keys** :Multiple types of Keys are supported by the SQL Server.

The following are the list of SQL Keys: <https://www.guru99.com/dbms-keys.html>

**Primary Key ,**[**Unique Key**](https://www.educba.com/unique-key-in-sql/) **,Candidate Key ,Alternate Key ,Composite Key , Super Key ,Foreign Key**

For Example

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Customer Table | | | | |
| cust\_id | cust\_name | cust\_address | cust\_aadhaar\_number | cust\_pan\_number |
| 100001 | Sunil Kumar | Noida | 372464389211 | ADSFS3456K |
| 100002 | Ankit Gupta | Gr Noida | 442289458453 | CGHAD7583L |
| 100003 | Suresh Yadav | New Delhi | 878453444144 | NMKRT2278O |
| 100004 | Nilam Singh | Lucknow | 227643441123 | HFJFD3876U |
| 100005 | Amal Rawat | Ghaziabad | 932571156735 | CBMVA9734A |
| 100006 | Harsh Saxena | Kanpur | 1453534363319 | TRYUC2568H |

Below given the “Order” table having the related data corresponding to the “cust\_id” from the Customer Table.

|  |  |  |
| --- | --- | --- |
| Order Table | | |
| cust\_id | order\_month\_year | order\_amount |
| 100001 | 2019 – Jan | $100,000 |
| 100002 | 2019 – Jan | $120,000 |
| 100003 | 2019 – Jan | $100,000 |
| 100004 | 2019 – Jan | $110,000 |
| 100001 | 2019 – Feb | $105,000 |
| 100002 | 2019 – Feb | $125,000 |

Now, we will go through one by one on each of the Key:

1. Primary Key

Primary Key is a field that can be used to identify all the tuples uniquely in the database. Only one of the columns can be declared as a primary key. A Primary Key can not have a NULL value.

Example: In the above given relational table, “cust\_id” is the Primary Key as it can identify all the row uniquely from the table.

2. Unique Key

Unique Key can be a field or set of fields that can be used to uniquely identify the tuple from the database. One or more fields can be declared as a unique Key. The unique Key column can also hold the NULL value. Use of Unique Key improves the performance of data retrieval. It makes searching for records from the database much more faster & efficient.

Example: In the above given relational table, “cust\_aadhaar\_number”, “cust\_pan\_number” are the Unique Key as it can allow one value as a NULL in the column

3. Candidate Key

Candidate Key can be a column or group of columns that can qualify for the Unique Key. Every table has at least one Candidate Key. A table may have one or more Candidate Key. Each Candidate Key can work as a Primary Key if required in certain scenarios.

Example: In the above given relational table, “cust\_id”, “cust\_aadhaar\_number”, “cust\_pan\_number” are the Candidate Key as it can identify all the row uniquely from the table. These columns also qualify the criteria to be a Primary Key.

 4. Alternate Key

Alternate Key is that Key which can be used as a Primary Key if required. Alternate Key also qualifies to be a Primary Key but for the time being, It is not the Primary Key.

Example: In the above given relational table, “cust\_aadhaar\_number”, “cust\_pan\_number” are the Alternate Key as both of the columns can be a Primary Key but not yet selected for the Primary Key.

5. Composite Key

Composite Key is also known as Compound Key / Concatenated Key. Composite Key refers to a group of two or more columns that can be used to identify a tuple from the table uniquely. A group of the column in combination with each other can identify a row uniquely but a single column of that group doesn’t promise to identify the row uniquely.

Example: In the above given relational table i.e. Order Table, “cust\_id”, “order\_month\_year” group of these columns used in combination to identify the tuple uniquely in the Order Table. The individual column of this table is not able to identify the tuple uniquely from the Order table.

6. Super Key

Super Key is a combination of columns, each column of the table remains dependent on it. Super Key may have some more columns in the group which may or may not be necessary to identify the tuple uniquely from the table. Candidate Key is the subset of the Super Key. Candidate Key is also known as minimal Super Key.

Example: In the above given relational table, Primary Key, Candidate Key & Unique Key is the Super Key. As a single column of Customer Table i.e ‘cust\_id’ is sufficient to identify the tuples uniquely from the table. Any set of the column which contains ‘cust\_aadhaar\_number’, ‘cust\_pan\_number’ is a Super Key.

7. Foreign Key

A foreign key is a column which is known as Primary Key in the other table i.e. A Primary Key in a table can be referred to as a Foreign Key in another table. Foreign Key may have duplicate & NULL values if it is defined to accept NULL values.

Example: In the above given relational table, ‘cust\_id’ is Primary Key in the Customer table but ‘cust\_id’ in the Order table known as a ‘Foreign Key’. Foreign Key in a table always becomes the Primary Key on the other table.

select all row which has empty or null value in Column(COL\_NAME)

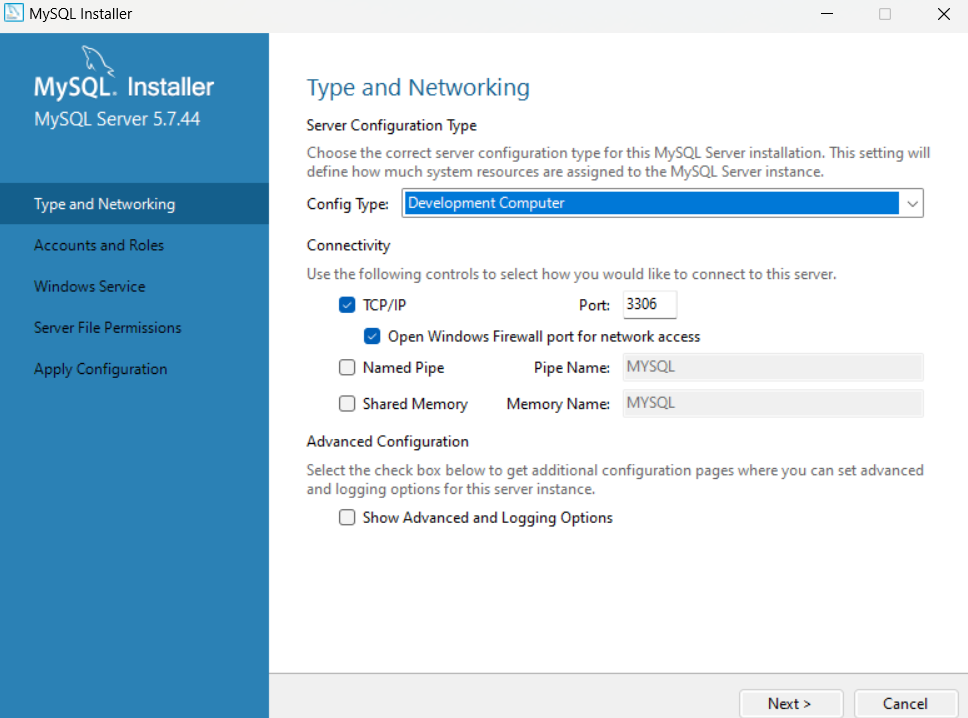
SELECT \* FROM TABLE WHERE COL\_NAME IS NULL OR LENGTH(TRIM (COL\_NAME)) = 0

**SQL Queries: -**

* Write a SQL query to find the second highest salary in a table.
* How do you perform a recursive query in SQL (e.g., for hierarchical data)?
* What is the difference between RANK(), DENSE\_RANK(), and ROW\_NUMBER() in SQL?

MySql

Password:- Rajiv@123



-- Create the Employees table

CREATE TABLE Employees (

ID INT PRIMARY KEY,

Name VARCHAR(100),

Designation VARCHAR(100),

Salary DECIMAL(10, 2),

ManagerID INT

);

-- Insert sample data

INSERT INTO Employees (ID, Name, Designation, Salary, ManagerID) VALUES

(1, 'Alice Johnson', 'CEO', 200000.00, NULL), -- Top-level, no manager

(2, 'Bob Smith', 'CTO', 180000.00, 1), -- Reports to CEO

(3, 'Carol White', 'CFO', 180000.00, 1), -- Reports to CEO

(4, 'David Brown', 'VP of Engineering', 150000.00, 2), -- Reports to CTO

(5, 'Eve Black', 'VP of Finance', 150000.00, 3), -- Reports to CFO

(6, 'Frank Green', 'Engineering Manager', 130000.00, 4), -- Reports to VP of Engineering

(7, 'Grace Lee', 'Software Engineer', 110000.00, 6), -- Reports to Engineering Manager

(8, 'Henry Scott', 'Software Engineer', 110000.00, 6), -- Reports to Engineering Manager

(9, 'Ivy Walker', 'Finance Manager', 120000.00, 5), -- Reports to VP of Finance

(10, 'Jack King', 'Accountant', 90000.00, 9); -- Reports to Finance Manager

**find the Nth highest salary in a table**

To find the **Nth highest salary** in a table, you can use a SQL query that ranks the distinct salary values and selects the salary at the desired position. There are several ways to do this, and one of the most common methods is using the **LIMIT/OFFSET** clause or **window functions** like ROW\_NUMBER() or DENSE\_RANK().

**Approach 1: Using DISTINCT, ORDER BY, and LIMIT/OFFSET (for MySQL)**

In databases like MySQL that support LIMIT with OFFSET, you can find the Nth highest salary using this approach.

Assume we have a table named Employees with a column Salary.

SELECT DISTINCT Salary

FROM Employees

ORDER BY Salary DESC

LIMIT 1 OFFSET N-1;

* **Explanation**:
  + DISTINCT Salary: Ensures that only distinct salary values are considered.
  + ORDER BY Salary DESC: Orders the salaries in descending order so that the highest salaries come first.
  + LIMIT 1 OFFSET N-1: Retrieves only one row, but skips the first N-1 rows to get the Nth highest salary.
*  Use **DISTINCT** to eliminate duplicates based on the selected columns.
*  Use **GROUP BY** if you want to group duplicates and aggregate data.
*  Use **ROW\_NUMBER()** (available in MySQL 8.0 and above) to uniquely rank rows and filter out duplicates.
*  Apply **subqueries** to first remove duplicates and then paginate the results.
*  Use **consistent sorting** (such as by a primary key) to prevent duplicates from appearing across different paginated pages.

**Approach 2: Using ROW\_NUMBER() (for SQL Server, PostgreSQL, Oracle)**

The ROW\_NUMBER() function works similarly to DENSE\_RANK() but without assigning the same rank to duplicate values.

SELECT Salary

FROM (

SELECT Salary, ROW\_NUMBER() OVER (ORDER BY Salary DESC) AS rn

FROM Employees

) AS ranked\_salaries

WHERE rn = N;

* **Explanation**:
  + ROW\_NUMBER() assigns a unique rank to each row, even if the salary values are the same.
  + You then filter the result to find the row with the desired rank (i.e., rn = N).

**Example - select salary from (select salary,ROW\_NUMBER() over (order by salary desc) rn From Employees ) as rank where rank.rn=3;**

**Approach 3: Using a Subquery (for simpler databases like MySQL)**

This method uses a correlated subquery to find the Nth highest salary.

SELECT Salary

FROM Employees e1

WHERE N - 1 = (SELECT COUNT(DISTINCT Salary)

FROM Employees e2

WHERE e1.Salary< e2.Salary);

* Explanation:
  + The subquery counts how many distinct salaries are higher than the salary in the current row (e1.Salary).
  + The outer query filters rows where the count of higher salaries equals N-1, which effectively selects the row with the Nth highest salary.

**What is the difference between RANK(), DENSE\_RANK(), and ROW\_NUMBER() in SQL**

The functions RANK(), DENSE\_RANK(), and ROW\_NUMBER() are all **window functions** in SQL used to assign a unique rank or position to rows in a result set. They are often used for ranking results based on the values in a particular column, typically within partitions or over ordered sets. Although they seem similar, they handle ties and ordering differently.

**1. ROW\_NUMBER()**

* **Purpose**: Assigns a unique sequential number to each row, starting from 1.
* **How it handles ties**: It does **not** account for ties. Even if two or more rows have the same value, ROW\_NUMBER() will assign different sequential numbers to them.
* **Use case**: When you want unique numbering for each row in the result set, regardless of the values in the columns.

**Example:**

SELECT Name, Salary, ROW\_NUMBER() OVER (ORDER BY Salary DESC) AS row\_num

FROM Employees;

| **Name** | **Salary** | **row\_num** |
| --- | --- | --- |
| Alice Johnson | 200000.00 | 1 |
| Bob Smith | 180000.00 | 2 |
| Carol White | 180000.00 | 3 |
| David Brown | 150000.00 | 4 |
| Eve Black | 150000.00 | 5 |

In this example, Bob Smith and Carol White have the same salary, but ROW\_NUMBER() gives them unique row numbers (2 and 3).

**2. RANK()**

* **Purpose**: Assigns a rank to each row based on the ordering of a specified column, starting from 1.
* **How it handles ties**: It **does** account for ties. Rows with the same value receive the **same rank**, but the next rank is **skipped** by the number of tied rows.
* **Use case**: When you want ranking but need to leave gaps in the ranks after handling ties.

**Example:**

SELECT Name, Salary, RANK() OVER (ORDER BY Salary DESC) AS rank

FROM Employees;

| **Name** | **Salary** | **rank** |
| --- | --- | --- |
| Alice Johnson | 200000.00 | 1 |
| Bob Smith | 180000.00 | 2 |
| Carol White | 180000.00 | 2 |
| David Brown | 150000.00 | 4 |
| Eve Black | 150000.00 | 4 |

In this case, Bob Smith and Carol White have the same salary and share rank 2. The next rank skips to 4, leaving a gap for rank 3.

**3. DENSE\_RANK()**

* **Purpose**: Similar to RANK(), but without gaps in the rank values.
* **How it handles ties**: Rows with the same value receive the same rank, but the next rank is **not skipped** (no gaps).
* **Use case**: When you want ranking but do **not** want gaps between rank values when there are ties.

**Example:**

SELECT Name, Salary, DENSE\_RANK() OVER (ORDER BY Salary DESC) AS dense\_rank

FROM Employees;

| **Name** | **Salary** | **dense\_rank** |
| --- | --- | --- |
| Alice Johnson | 200000.00 | 1 |
| Bob Smith | 180000.00 | 2 |
| Carol White | 180000.00 | 2 |
| David Brown | 150000.00 | 3 |
| Eve Black | 150000.00 | 3 |

Here, Bob Smith and Carol White have the same salary and share rank 2, but the next rank is 3 (no skipped ranks).

**SQL Query interview questions for experienced developer**

Here are some advanced SQL query interview questions for an experienced developer with 9 years of experience:

**1. Find Nth Highest Salary**

Write a query to find the Nth highest salary from the Employees table.

* **Answer**:

SELECT Salary

FROM (

SELECT Salary,

DENSE\_RANK() OVER (ORDER BY Salary DESC) AS rank

FROM Employees

) AS ranked\_salaries

WHERE rank = N;

**2. Find Duplicate Records in a Table**

Write a SQL query to find all records that have duplicate values in a specific column (e.g., Email in an Employees table).

* **Answer**:

SELECT Email, COUNT(\*)

FROM Employees

GROUP BY Email

HAVING COUNT(\*) > 1;

**3. Delete Duplicate Rows Without Primary Key**

How would you delete duplicate rows from a table that doesn't have a primary key, but all columns are duplicates?

* **Answer**:

DELETE FROM Employees

WHERE ID NOT IN (

SELECT MIN(ID)

FROM Employees

GROUP BY Name, Email, Designation

);

**Answer:**

WITH CTE AS (

SELECT ID, Name, Designation, Salary, ManagerID,

ROW\_NUMBER() OVER (PARTITION BY Name ORDER BY ID) AS rn

FROM Employees

WHERE Designation = 'Developer'

)

DELETE FROM Employees

WHERE ID IN (

SELECT ID

FROM CTE

WHERE rn > 1

);

**4. Find the Top 3 Salaries for Each Department**

Write a query to find the top 3 highest salaries in each department.

* **Answer**:

SELECT Department, Name, Salary

FROM (

SELECT Department, Name, Salary,

DENSE\_RANK() OVER (PARTITION BY Department ORDER BY Salary DESC) AS rank

FROM Employees

) AS ranked\_salaries

WHERE rank <= 3;

**5. Find Employees Who Do Not Report to Anyone**

Write a query to find all employees who do not have a manager (i.e., ManagerID is NULL).

* **Answer**:

SELECT Name, Designation

FROM Employees

WHERE ManagerID IS NULL;

**6. Find the Department with the Highest Total Salary**

Write a query to find the department with the highest total salary.

* **Answer**:

SELECT Department

FROM Employees

GROUP BY Department

ORDER BY SUM(Salary) DESC

LIMIT 1;

**7. Get Running Total of Salaries**

Write a query to calculate the running total of salaries for employees, ordered by their ID.

* **Answer**:

SELECT ID, Name, Salary,

SUM(Salary) OVER (ORDER BY ID) AS running\_total

FROM Employees;

**8. Find the Number of Employees in Each Department**

Write a query to get the number of employees working in each department.

* **Answer**:

SELECT Department, COUNT(\*) AS num\_employees

FROM Employees

GROUP BY Department;

**9. Query to Fetch Data in a Hierarchical Format**

Given an Employees table with ManagerID referencing the employee’s manager, write a query to fetch employee-manager hierarchical data.

* **Answer**:

SELECT e.Name AS Employee, m.Name AS Manager

FROM Employees e

LEFT JOIN Employees m ON e.ManagerID = m.ID;

or

select manager.Name as employee\_name, emp.Name as emp\_name from Employees emp , Employees manager where emp.Id =manager.ManagerID;

**10. Find Employees with the Highest Salary in Each Department**

Write a query to find employees who have the highest salary in each department.

* **Answer**:

SELECT Name, Salary, Department

FROM Employees

WHERE (Department, Salary) IN ( ---- two parameter is not supported in sql server

SELECT Department, MAX(Salary)

FROM Employees

GROUP BY Department

);

or

select e1.Designation,MAX(e1.Salary) from Employees e1 group by e1.Designation

**11. Get the Second Highest Salary Using Subquery**

Write a query to get the second highest salary using a subquery.

* **Answer**:

SELECT MAX(Salary)

FROM Employees

WHERE Salary < (SELECT MAX(Salary) FROM Employees);

**12. Update Employee Salary Based on Department Average**

Write a query to increase the salary of employees by 10% if their salary is below the department’s average salary.

* **Answer**:

UPDATE Employees e

SET Salary = Salary \* 1.10

WHERE Salary < (

SELECT AVG(Salary)

FROM Employees

WHERE Department = e.Department

);

**13. Find the Last Record from the Table**

Write a query to fetch the last record from the Employees table based on ID.

* **Answer**:

SELECT \*

FROM Employees

ORDER BY ID DESC

LIMIT 1;

**14. Using Recursive Query to Get Employee Hierarchy**

Write a recursive query to get the full reporting hierarchy for a specific employee.

* **Answer**:

WITH RECURSIVE EmployeeHierarchy AS (

SELECT ID, Name, ManagerID

FROM Employees

WHERE ID = <employee\_id> -- Start with a specific employee

UNION ALL

SELECT e.ID, e.Name, e.ManagerID

FROM Employees e

INNER JOIN EmployeeHierarchy eh ON e.ID = eh.ManagerID

)

SELECT \* FROM EmployeeHierarchy;

**15. Identify Gaps in a Sequence**

Given an Employees table with sequential IDs, write a query to find gaps in the ID sequence.

* **Answer**:

SELECT (ID + 1) AS missing\_id

FROM Employees e

WHERE NOT EXISTS (

SELECT 1 FROM Employees WHERE ID = e.ID + 1

);

These questions cover a wide range of advanced SQL topics like ranking, window functions, CTEs, recursive queries, and hierarchical data management, which are relevant for a developer with 9 years of experience.