**Introduction:**

My self Pavan,i have 2+ years of experiance as sql developer.

Coming to daily activities creating stored procedures and tables and views based on the requirement.

In my current proj daily we are getting data from in-bound and outbound insurance and claim related data.

Feeding the data to analytics teams and source is coming oracle db's.they placed into sql db which act as source for me.

I have worked on database objects like tables,views,sp,functions,cursors.

**Interview Topics:**

1)performance tunning

2)truncate load and full load

3)what is the use of incremental load and how to implement in sp?

4)joins

5)merge statement in sql

6)dynamic sql

7)how to delete duplicate record?

8)rank and dense rank diff?

9)sub qurries

10)Variable creation

11)Cursors

12)Data Types

13) Diff b/w stored procedure and functions SQL

14)data import /export mechanism from one DB to another

15)order by cannot be used with union know why

**1)Performance tunning:**

SQL tuning is the process of improving SQL queries to accelerate your servers performance.

It's general purpose is to reduce the amount of time it takes a user to receive a result after issuing a query, and to reduce the amount of resources used to process a query.

SQL performance tuning consists of making queries of a relation database run as fast as possible. SQL performance tuning is not a single tool or technique. Rather, it's a set of practices that makes use of a wide array of techniques, tools, and processes.

**Index Optimization:**

Create and maintain indexes strategically to enhance query performance.

Regularly analyze and optimize indexes to ensure they align with the most common query patterns.

**Query Optimization:**

Write efficient queries by selecting only the necessary columns and minimizing the use of functions in WHERE clauses.

Optimize join conditions and types, and consider de-normalization where applicable.

**Statistics Management:**

Keep database statistics up to date to assist the query optimizer in making informed decisions.

Regularly update statistics for tables and indexes, especially after significant data changes.

**Regular Maintenance and Monitoring:**

Implement routine maintenance tasks such as rebuilding indexes and updating statistics.

Monitor performance using tools like query execution plans and profiling tools, adjusting configurations as needed.

**SELECT DISTINCT job\_title**

**FROM employees;**

**This query would return a result set containing only distinct job titles from the employees table.**

While SELECT DISTINCT is useful, it's important to be cautious about its performance implications, especially on large datasets. It can be resource-intensive because the database has to scan the entire result set to identify and eliminate duplicates. In some cases, alternative approaches like using GROUP BY or finding other ways to structure the query might be more efficient.

Performance tuning in SQL involves optimizing the execution speed and efficiency of your SQL queries and database operations. Here are some general tips and techniques for SQL performance tuning:

**Use Indexes Wisely:**

Indexes can significantly improve query performance by allowing the database engine to quickly locate and retrieve the necessary data. However, over-indexing can lead to overhead, so use indexes judiciously.

Regularly analyze and optimize indexes based on query patterns.

**Optimize Your Queries:**

Write efficient queries by minimizing the use of SELECT \* and fetching only the required columns.

Use appropriate join types (INNER JOIN, LEFT JOIN, etc.) based on your data relationships.

Avoid using functions in WHERE clauses on indexed columns, as it can prevent the use of indexes.

**Update Statistics:**

Keep the database statistics up to date to help the query optimizer make better decisions.

Regularly update statistics for tables and indexes, especially after significant data changes.

**Limit the Use of Cursors:**

Cursors can be resource-intensive. Whenever possible, use set-based operations instead of cursors to manipulate data.

**Optimize Joins:**

Ensure that your joins are well-optimized. Use appropriate join conditions and indexes to speed up the join process.

Consider denormalization in cases where it can improve query performance.

**Avoid SELECT DISTINCT:**

Using SELECT DISTINCT can be resource-intensive. If possible, try to find alternative ways to achieve the desired result.

**Partitioning Tables:**

If your tables are large, consider partitioning them. This can improve query performance by allowing the database to only scan the relevant partitions.

**Use Stored Procedures:**

Stored procedures can be precompiled and optimized, reducing the overhead associated with query parsing and compilation.

**Limit the Use of Subqueries:**

Subqueries can be slow, especially if they are correlated. Where possible, use JOINs or other set-based operations instead.

**Optimize Server Configuration:**

Adjust server settings such as memory allocation, parallelism, and caching to suit your specific workload and hardware.

**Use Query Execution Plan:**

Analyze and understand the query execution plan generated by the database engine. Use tools like SQL Server Management Studio (SSMS) or EXPLAIN in PostgreSQL to identify potential bottlenecks.

**Monitor and Analyze Performance:**

Regularly monitor the performance of your database using tools like performance counters, query execution plans, and database profiling tools.

**Regular Database Maintenance:**

Perform regular maintenance tasks such as rebuilding indexes, updating statistics, and optimizing storage to keep the database in good health.

Remember that the effectiveness of these strategies may vary depending on the specific database system you are using (e.g., MySQL, PostgreSQL, SQL Server) and the nature of your application's workload. It's essential to analyze and understand your specific performance challenges to implement the most appropriate optimizations.

**2)Truncate load and full load:**

In SQL, the TRUNCATE TABLE statement is a Data Manipulation Language (DML) operation that deletes all rows of a table without causing a triggered action. The result of this operation quickly removes all data from a table, typically bypassing a number of integrity enforcing mechanisms.

A target table is truncated before loading everything from the source. That's why this technique is also known as Destructive Load. In full load first we truncate the destination table and then we load all the data from source to destination. It is the simplest method to load the data from source to destination.

Full load and truncate load are terms commonly used in the context of data warehousing and database management, particularly when dealing with data integration and ETL (Extract, Transform, Load) processes. Here's an explanation of each:

**Full Load:**

In a full load scenario, all the data from the source system is extracted and loaded into the target system or data warehouse.

This process involves transferring the entire dataset, regardless of whether the data has changed since the last load.

Full loads are typically used in the initial data population or when the entire dataset needs to be refreshed.

**Truncate Load:**

Truncate load is a strategy where the target table is first truncated (all existing data is deleted), and then the entire dataset is loaded into the target table.

This approach is more aggressive than a full load because it removes all existing data from the target table before inserting the new data.

Truncate load is often used when you want to ensure that the target table is an exact replica of the source data, and you don't want to deal with updates or deletions of individual records.

**Key Differences:**

**Data Handling:**

In a full load, all data is transferred from the source to the target, regardless of whether it has changed.

In a truncate load, the target table is cleared (truncated) before loading the entire dataset.

**Efficiency:**

Full loads can be less efficient, especially when dealing with large datasets, as they transfer unchanged data.

Truncate loads can be more efficient in terms of loading speed, as they remove the existing data and then insert the new data.

**Use Cases:**

Full loads are commonly used for initial data loads, periodic refreshes, or when dealing with small datasets.

Truncate loads are suitable when you want to ensure a clean slate in the target table and when dealing with scenarios where updates and deletions are less frequent.

It's important to note that the choice between full load and truncate load depends on the specific requirements of the data integration process and the characteristics of the data being transferred. In some cases, a combination of both strategies might be employed, depending on the nature of the data and the update patterns.

**3)Incremental load:**

Incremental Load means comparing the target table against the source data based on technical columns InsertionDate, UpdateDate, and DeletionDate. The logic is like below, If there are any New records in Source data, then we have to insert those records in the target table.

Incremental loading refers to the process of loading only the data that has changed since the last load. This is in contrast to a full load, which loads all of the data in the source system into the data warehouse. You can use control tables to do incremental loading in PostgreSQL.

Use a Script task or an Execute SQL task to assemble the SQL statement that will be used to query for changes. For an example of how to assemble the query, see Prepare to Query for the Change Data. Use multiple Data Flow tasks to load the change data from each source table and apply it to the destination.

**Using Incremantal load in SP (using merge):**

This is the stored procedure for the merge method. The stored procedure below loads data from source only when there are new records inserted, else it will discard. Updates only when there are updates in the record.

-- Assume staging\_data has columns id, name, age, and other\_columns

-- Assume target\_data has the same structure

-- Use the MERGE statement to perform an incremental load

MERGE INTO target\_data AS target

USING staging\_data AS source

ON target.id = source.id

-- Specify the conditions for matching rows

WHEN MATCHED THEN

-- If the row exists in both tables, update the existing data if needed

UPDATE SET

target.name = source.name,

target.age = source.age,

-- Update other columns as needed

-- Specify the conditions for non-matching rows

WHEN NOT MATCHED BY TARGET THEN

-- If the row exists in staging\_data but not in target\_data, insert the new row

INSERT (id, name, age, other\_columns)

VALUES (source.id, source.name, source.age, source.other\_columns);

-- Additional actions can be defined, such as deleting rows not present in the staging table

-- WHEN NOT MATCHED BY SOURCE THEN

-- DELETE;

**Merge in SP for incremental load:**

ALTER/CREATE PROCEDURE Merge\_Source\_Target\_Data

AS

BEGIN

-- Assume staging\_data has columns id, name, age, and other\_columns

-- Assume target\_data has the same structure

-- Use the MERGE statement to perform an incremental load

MERGE INTO target\_data AS target

USING staging\_data AS source

ON target.id = source.id

-- Specify the conditions for matching rows

WHEN MATCHED THEN

-- If the row exists in both tables, update the existing data if needed

UPDATE SET

target.name = source.name,

target.age = source.age,

-- Update other columns as needed

-- Specify the conditions for non-matching rows

WHEN NOT MATCHED BY TARGET THEN

-- If the row exists in staging\_data but not in target\_data, insert the new row

INSERT (id, name, age, other\_columns)

VALUES (source.id, source.name, source.age, source.other\_columns);

-- Additional actions can be defined, such as deleting rows not present in the staging table

-- WHEN NOT MATCHED BY SOURCE THEN

-- DELETE;

END

EXEC Merge\_Source\_Target\_Data; -- execution of SP

"Incremental Load" isn't some magic concept. There must be a way to identify new or modified rows. SSIS lookups, as the name says, do lookups, not INSERTs. If you try to force and INSERT or Ignore, you're doing it the slowest way possible, no matter what database is used. Even if you used MySQL and ON CONFLICT IGNORE you'd still get very very slow performance.

All the major commercial databases (DB2, Oracle, SQL Server) have some kind of replication or change data capture that can be used to find and copy modified data. You'll have to find the equivalent for DB2. Another option is to use a modified date column or indicator and copy only columns that have been modified since the last time. Another option is to copy all PK values to a staging table to identify which are new, then load data from DB2 only for the new PK values

**4)Joins:**

**SQL | Join (Inner, Left, Right and Full Joins):**

SQL Join statement is used to combine data or rows from two or more tables based on a common field between them.

**Different types of Joins are as follows:**

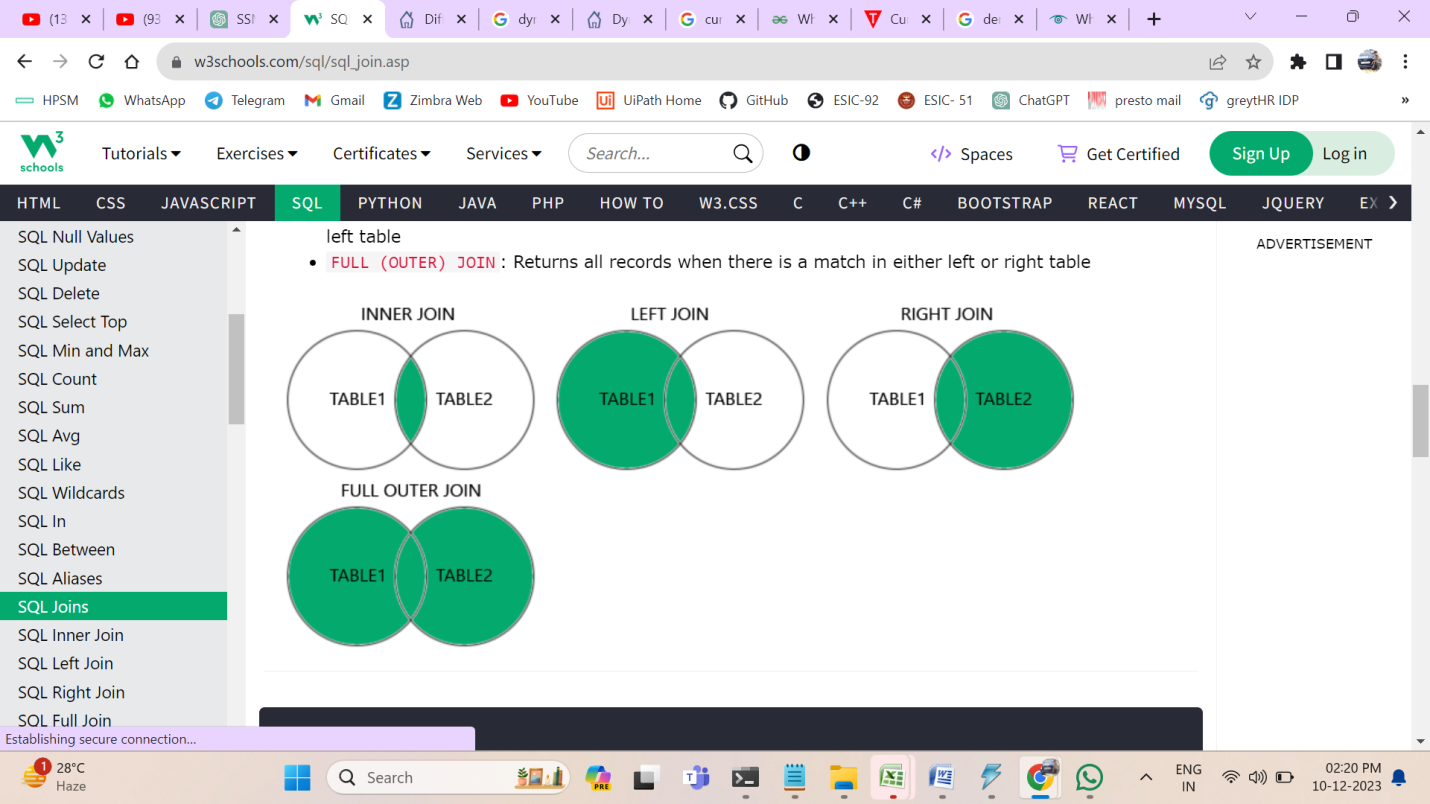
**INNER JOIN**

**LEFT JOIN**

**RIGHT JOIN**

**FULL JOIN**

**NATURAL JOIN**

****

**INNER JOIN:**

The INNER JOIN keyword selects all rows from both the tables as long as the condition is satisfied.create result-set by combining all rows from both the tables where the condition satisfies i.e value of the common field will be the same.

**Syntax:**

SELECT table1.column1,table1.column2,table2.column1,....

FROM table1

INNER JOIN table2

ON table1.matching\_column = table2.matching\_column;

**Note: We can also write JOIN instead of INNER JOIN. JOIN is same as INNER JOIN.**

**LEFT JOIN:**

This join returns all the rows of the table on the left side of the join and matches rows for the table on the right side of the join. For the rows for which there is no matching row on the right side, the result-set will contain null. LEFT JOIN is also known as LEFT OUTER JOIN.

**Syntax:**

SELECT table1.column1,table1.column2,table2.column1,....

FROM table1

LEFT JOIN table2

ON table1.matching\_column = table2.matching\_column;

**table1:** First table.

**table2:** Second table

matching\_column: Column common to both the tables.

**Note: We can also use LEFT OUTER JOIN instead of LEFT JOIN, both are the same.**

**RIGHT JOIN:**

RIGHT JOIN is similar to LEFT JOIN. This join returns all the rows of the table on the right side of the join and matching rows for the table on the left side of the join. For the rows for which there is no matching row on the left side, the result-set will contain null. RIGHT JOIN is also known as RIGHT OUTER JOIN.

**Syntax:**

SELECT table1.column1,table1.column2,table2.column1,....

FROM table1

RIGHT JOIN table2

ON table1.matching\_column = table2.matching\_column;

**table1**: First table.

**table2**: Second table

matching\_column: Column common to both the tables.

**Note: We can also use RIGHT OUTER JOIN instead of RIGHT JOIN, both are the same.**

**Example Queries(RIGHT JOIN):**

SELECT Student.NAME,StudentCourse.COURSE\_ID

FROM Student

RIGHT JOIN StudentCourse

ON StudentCourse.ROLL\_NO = Student.ROLL\_NO;

**FULL JOIN:**

FULL JOIN creates the result-set by combining results of both LEFT JOIN and RIGHT JOIN. The result-set will contain all the rows from both tables. For the rows for which there is no matching, the result-set will contain NULL values.

**Syntax:**

SELECT table1.column1,table1.column2,table2.column1,....

FROM table1

FULL JOIN table2

ON table1.matching\_column = table2.matching\_column;

**table1**: First table.

**table2**: Second table

matching\_column: Column common to both the tables.

**Example Queries(FULL JOIN):**

SELECT Student.NAME,StudentCourse.COURSE\_ID

FROM Student

FULL JOIN StudentCourse

ON StudentCourse.ROLL\_NO = Student.ROLL\_NO;

**NATURAL JOIN:**

In SQL, a natural join is a type of join that automatically matches columns with the same names in the tables being joined. The natural join keyword is "NATURAL JOIN." When you perform a natural join, the database engine automatically compares the column names in both tables and creates a join condition based on matching column names.

Here's a simple example:

SELECT \*

FROM employees NATURAL JOIN departments;

In this example, if both the employees and departments tables have a column named,

for instance, department\_id, the natural join would automatically use that column for the join condition.

The result would include all columns from both tables, with only one department\_id column present in the output.

**5)Merge statement in sql:**

**Purpose of MERGE:**

The MERGE statement in SQL is a powerful tool for performing conditional inserts, updates, or deletes in a single operation based on a specified condition.

**Syntax Overview:**

The basic syntax includes the keywords MERGE, USING, ON, WHEN MATCHED, and WHEN NOT MATCHED. It allows you to join a target table with a source table and define actions to be taken when a match is found (e.g., update) or not found (e.g., insert).

USE SqlShackMergeDemo

    GO

    MERGE TargetProducts AS Target

    USING SourceProducts AS Source

    ON Source.ProductID = Target.ProductID

    -- For Inserts

    WHEN NOT MATCHED BY Target THEN

        INSERT (ProductID,ProductName, Price)

        VALUES (Source.ProductID,Source.ProductName, Source.Price)

    -- For Updates

    WHEN MATCHED THEN UPDATE SET

        Target.ProductName = Source.ProductName,

        Target.Price = Source.Price;

**Benefits of MERGE:**

**Efficiency**:

MERGE minimizes the number of passes through the data, improving performance compared to separate INSERT, UPDATE, and DELETE statements.

**Atomic Operations**:

The entire operation is atomic, ensuring either all or none of the specified changes are applied.

**Conciseness**:

It provides a more concise and readable way to express conditional logic for data synchronization compared to multiple individual statements.

**Example Use Case:**

An example use case could be maintaining a target table of customer information. You can use MERGE to synchronize the target table with changes in the source table, updating existing records if they exist and inserting new records if they don't, all based on a specific condition like a matching identifier.

This answer provides a brief yet comprehensive overview of the MERGE statement, addressing its purpose, syntax, benefits, and a practical use case.

**6)Dynamic SQL:**

**Question: Can you explain what dynamic SQL is and provide a simple example?**

**Answer: Definition**

Dynamic SQL is a technique where SQL statements are constructed and executed dynamically at runtime, allowing for flexibility in query generation.

**Example:**

DECLARE @TableName NVARCHAR(50) = 'your\_table\_name';

DECLARE @DynamicQuery NVARCHAR(MAX);

SET @DynamicQuery = 'SELECT \* FROM ' + QUOTENAME(@TableName);

EXEC sp\_executesql @DynamicQuery;

**Question: What are the advantages of using dynamic SQL?**

**Answer: Flexibility**

Dynamic SQL provides flexibility by allowing the creation of dynamic queries based on changing conditions or user inputs.

**Example:**

A stored procedure with optional parameters that conditionally modifies the WHERE clause based on user inputs.

**Question: What are the potential risks or drawbacks of using dynamic SQL?**

**Answer: SQL Injection**

One significant risk is SQL injection if input is not properly validated or sanitized. Dynamic SQL should use parameterized queries to mitigate this risk.

**Example:**

DECLARE @InputParameter NVARCHAR(MAX)

SET @InputParameter = ''' OR 1=1; --'

DECLARE @SQLQuery NVARCHAR(MAX)

SET @SQLQuery = 'SELECT \* FROM Users WHERE UserName = ' + @InputParameter

**Question: How can you mitigate the risk of SQL injection when using dynamic SQL?**

**Answer: Parameterization**

Mitigate the risk by using parameterized queries, ensuring that user inputs are treated as parameters rather than concatenated directly into the SQL string.

**Example:**

DECLARE @InputParameter NVARCHAR(MAX)

SET @InputParameter = ''' OR 1=1; --'

DECLARE @SQLQuery NVARCHAR(MAX)

SET @SQLQuery = 'SELECT \* FROM Users WHERE UserName = @UserName'

EXEC sp\_executesql @SQLQuery, N'@UserName NVARCHAR(MAX)', @InputParameter

**8)Rank and dense rank diff?:**

**Question: What is the difference between RANK() and DENSE\_RANK() in SQL?**

**Answer:**

**Definition**

**RANK() and DENSE\_RANK()** are window functions in SQL used for ranking rows within a result set based on specified criteria.

**RANK():**

**Definition: RANK()** assigns a unique rank to each distinct row based on the specified column(s).

**Behavior:**

If two or more rows have the same values, they receive the same rank, and the next rank is skipped.

**DENSE\_RANK():**

**Definition: DENSE\_RANK()** also assigns a unique rank to each distinct row based on the specified column(s).

**Behavior**:

If two or more rows have the same values, they receive the same rank, but the next rank is not skipped. The ranks are consecutive without gaps.

**Example:**

Consider a table of sales data where multiple salespeople have the same sales amount. With RANK(), if two salespeople have the highest sales, they both get a rank of 1, and the next rank is 3. With DENSE\_RANK(), they both get a rank of 1, and the next rank is 2.

In summary, both RANK() and DENSE\_RANK() assign ranks to rows, but RANK() may have gaps in the ranking sequence, while DENSE\_RANK() produces consecutive ranks without gaps for rows with the same values.

**9)Sub queries:**

**Question: What are subqueries in SQL, and how are they used?**

**Answer: Definition**

**Subquery:**

A subquery is a query nested within another query, and it's enclosed in parentheses. It can be used to retrieve data that will be used in the main query's condition or expression.

**Example:**

SELECT ProductName

FROM Products

WHERE CategoryID IN (SELECT CategoryID FROM Categories WHERE CategoryName = 'Electronics');

In this example, the subquery retrieves the CategoryID for the category 'Electronics,' and the main query uses that result to find products in the specified category.

**Types of Subqueries:**

**Single-row Subquery:**

A subquery that returns only one row, often used with single-row comparison operators (e.g., =, >, <).

**Multiple-row Subquery:**

A subquery that returns multiple rows, often used with multiple-row comparison operators (e.g., IN, ANY, ALL).

**Correlated Subquery:**

A subquery that refers to columns from the outer query, making the subquery dependent on the outer query.

**Use Cases:**

**Filtering Data:**

Subqueries can be used to filter data based on conditions derived from another table.

**Data Modification:**

Subqueries can be used to provide values for INSERT, UPDATE, or DELETE statements.

**Aggregate Calculations:**

Subqueries can be used to perform aggregate calculations on subsets of data.

Understanding subqueries is crucial for writing complex and efficient SQL queries. They provide a way to break down problems into smaller, manageable pieces and allow for more flexible and expressive queries.

**10)Variable creation:**

In SQL, you can create variables to store and manipulate data within a script or a batch of statements. The syntax for creating variables varies slightly between different database management systems. Here are examples for two popular systems: Microsoft SQL Server and MySQL.

**Microsoft SQL Server:**

In SQL Server, you can use the DECLARE statement to create a variable. Here's an example:

DECLARE @VariableName DataType;

SET @VariableName = Value;

-- Example:

DECLARE @UserName NVARCHAR(50);

SET @UserName = 'JohnDoe';

-- You can also declare and set a variable in a single line:

DECLARE @UserID INT = 1;

**MySQL:**

In MySQL, you use the SET keyword to assign a value to a user-defined variable. Here's an example:

**SET @VariableName = Value;**

-- Example:

SET @ProductName = 'ExampleProduct';

-- You can also declare and set a variable in a single line:

SET @ProductID := 101;

**Usage in Queries:**

Once you've created a variable, you can use it in your SQL queries. For example:

-- Using variables in a SELECT statement (SQL Server):

DECLARE @ProductCount INT;

SET @ProductCount = (SELECT COUNT(\*) FROM Products);

SELECT @ProductCount AS TotalProducts;

-- Using variables in a SELECT statement (MySQL):

SET @ProductCount := (SELECT COUNT(\*) FROM Products);

SELECT @ProductCount AS TotalProducts;

Remember that the scope of a variable is typically limited to the batch or script in which it's declared. The usage of variables can make your SQL scripts more dynamic and easier to maintain, especially when dealing with dynamic queries or storing intermediate results.

Always check the documentation of your specific database system for any variations in syntax or behavior regarding variable declaration and usage.

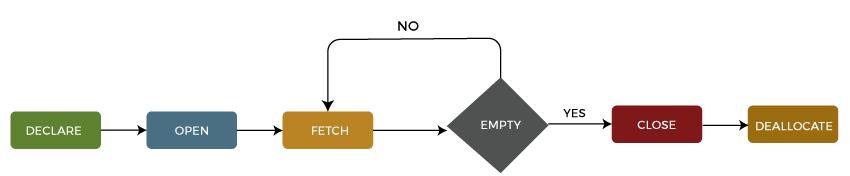
**11)Cursors:**

**Cursor in SQL Server**

A cursor in SQL Server is a database object that allows us to retrieve each row at a time and manipulate its data. A cursor is nothing more than a pointer to a row. It's always used in conjunction with a SELECT statement. It is usually a collection of SQL logic that loops through a predetermined number of rows one by one. A simple illustration of the cursor is when we have an extensive database of worker's records and want to calculate each worker's salary after deducting taxes and leaves.

The SQL Server cursor's purpose is to update the data row by row, change it, or perform calculations that are not possible when we retrieve all records at once. It's also useful for performing administrative tasks like SQL Server database backups in sequential order. Cursors are mainly used in the development, DBA, and ETL processes.

**Cursor Life Cycle:**



**Limitations of SQL Server Cursor:**

A cursor has some limitations so that it should always use only when there is no option except the cursor. These limitations are:

Cursor consumes network resources by requiring a network roundtrip each time it fetches a record.

A cursor is a memory resident set of pointers, which means it takes some memory that other processes could use on our machine.

It imposes locks on a portion of the table or the entire table when processing data.

The cursor's performance and speed are slower because they update table records one row at a time.

Cursors are quicker than while loops, but they do have more overhead.

The number of rows and columns brought into the cursor is another aspect that affects cursor speed. It refers to how much time it takes to open your cursor and execute a fetch statement.

-- Create a sample Employee table

CREATE TABLE Employee (

EmployeeID INT PRIMARY KEY,

EmployeeName VARCHAR(100)

);

-- Insert some sample data

INSERT INTO Employee (EmployeeID, EmployeeName) VALUES

(1, 'John Doe'),

(2, 'Jane Smith'),

(3, 'Bob Johnson');

-- Declare variables for cursor

DECLARE @EmployeeID INT;

DECLARE @EmployeeName VARCHAR(100);

-- Declare and set up the cursor

DECLARE employee\_cursor CURSOR FOR

SELECT EmployeeID, EmployeeName FROM Employee;

-- Open the cursor

OPEN employee\_cursor;

-- Fetch the first row into variables

FETCH NEXT FROM employee\_cursor INTO @EmployeeID, @EmployeeName;

-- Loop through the cursor

WHILE @@FETCH\_STATUS = 0

BEGIN

-- Process the current row (print in this example)

PRINT 'Employee ID: ' + CAST(@EmployeeID AS VARCHAR(10)) + ', Employee Name: ' + @EmployeeName;

-- Fetch the next row

FETCH NEXT FROM employee\_cursor INTO @EmployeeID, @EmployeeName;

END

-- Close and deallocate the cursor

CLOSE employee\_cursor;

DEALLOCATE employee\_cursor;

**12)SQL Data Types:**

The data type of a column defines what value the column can hold: integer, character, money, date and time, binary, and so on.

**SQL Data Types**

Each column in a database table is required to have a name and a data type.

An SQL developer must decide what type of data that will be stored inside each column when creating a table. The data type is a guideline for SQL to understand what type of data is expected inside of each column, and it also identifies how SQL will interact with the stored data.

**Normalization and denormalization:**

**Normalization:**

**Definition:**

Normalization is the process of organizing data in a relational database to reduce redundancy and dependency by dividing large tables into smaller, related tables.

**Objective**:

The main goal is to achieve data integrity and eliminate anomalies in the database by following a set of rules called normal forms.

**Example**:

If you have a combined table for customer and order information, normalization might involve breaking it into separate tables for customers and orders.

**Emphasis**:

Places emphasis on minimizing redundancy, maintaining data consistency, and reducing the risk of update anomalies.

**Denormalization:**

**Definition**:

Denormalization is the process of intentionally introducing redundancy into a database by combining tables or adding redundant data to improve read performance.

**Objective**:

Aims to enhance query performance by minimizing the need for joins and simplifying read operations, especially for reporting and analytics.

**Example**:

In a denormalized design, you might store redundant information, like customer addresses, directly in the customer table to avoid joins during read operations.

**Emphasis**: Emphasizes optimizing read operations, simplifying queries, and potentially increasing data redundancy for the sake of performance gains.

**13)Diff b/w stored procedure and fucntions sql:**

In SQL, stored procedures and functions are both database objects that contain a collection of SQL statements. However, there are some key differences between stored procedures and functions. Here are the main distinctions:

**Return Type:**

**Stored Procedure:**

Can return zero or more values. You typically use OUTPUT parameters to return values.

**Function:**

Must return a single value. This can be a scalar value or a table (in the case of table-valued functions).

**Usage in Queries:**

**Stored Procedure:**

Can be called using the EXECUTE statement or simply the procedure name. Can also be used in a SELECT statement.

**Function**:

Can be used in a SELECT statement or other T-SQL statements where an expression is allowed.

**Transaction Control:**

**Stored Procedure:**

Can contain transaction control statements such as COMMIT and ROLLBACK. You can control transactions explicitly within a stored procedure.

**Function:**

Generally, functions do not contain transaction control statements. They are designed to be deterministic and, as such, do not perform actions that would affect the state of the database.

**Error Handling:**

**Stored Procedure:**

Can use TRY...CATCH blocks for error handling. You can catch and handle errors within a stored procedure.

**Function:**

Limited error handling capabilities compared to stored procedures. Usually, errors propagate to the calling environment.

**Permissions:**

**Stored Procedure:**

Can be executed with explicit permission granted to users. The permissions can be granted on the procedure itself.

**Function:**

Generally requires permission to execute, but the permissions are granted on the function. Also, functions are often used in SELECT statements, so users need permission on the underlying tables.

**Location in SQL Statements:**

**Stored Procedure:**

Can be called from various places in a SQL script or application, including within other stored procedures, triggers, or directly in a SQL query.

**Function:**

Can be used directly in a SELECT, INSERT, UPDATE, or DELETE statement or within another function.

**Alteration and Execution:**

**Stored Procedure:**

Can be altered using the ALTER PROCEDURE statement. Can be executed using the EXECUTE statement.

**Function:**

Can be altered using the ALTER FUNCTION statement. Can be executed directly in a SQL query.

Remember that the specifics can vary depending on the database management system (DBMS) you are using, as different database systems have their own implementations and variations of stored procedures and functions.

**15)** **Data import /export mechanism from one DB to another:**

The process of importing and exporting data from one database to another involves transferring data between systems while ensuring data integrity and consistency. The specific steps and tools you use may depend on the type of databases involved, but I can provide you with a general guide that covers common scenarios.

**1. Identify Source and Target Databases:**

**Source Database:** This is the database from which you want to export data.

**Target Database**: This is the database where you want to import the data.

**2. Choose a Method of Data Transfer:**

**a. Manual Export/Import:**

**Export from Source Database:**

Use database-specific tools or SQL queries to export data into a file (e.g., CSV, SQL dump).

**Transfer the File:**

Move the exported file from the source environment to the target environment.

**Import to Target Database:**

Use database-specific tools or scripts to import data from the file into the target database.

**b. Database Replication:**

Set up database replication to automatically copy changes from the source to the target in real-time or at scheduled intervals.

**c. ETL (Extract, Transform, Load) Tools:**

Use ETL tools like Apache NiFi, Talend, or Apache Spark to automate the extraction, transformation, and loading of data from the source to the target.

**3. Database-specific Tools and Commands:**

a. SQL Databases (e.g., MySQL, PostgreSQL, SQL Server):

Export:

Use mysqldump for MySQL or pg\_dump for PostgreSQL.

Example (MySQL): mysqldump -u username -p dbname > dump.sql

**Import:**

Use mysql or psql.

**Example (MySQL):**

mysql -u username -p dbname < dump.sql

**-- Export data to a SQL dump file**

mysqldump -u username -p source\_db\_name > dump.sql

**-- Import data from a SQL dump file**

mysql -u username -p target\_db\_name < dump.sql

b. NoSQL Databases (e.g., MongoDB):

**Export:**

Use mongodump.

**Example:**

mongodump --db dbname --out /path/to/dump/directory

**Import:**

**Use mongorestore.**

**Example:**

**mongorestore --db dbname /path/to/dump/directory**

**4. Data Mapping and Transformation:**

Ensure that data types, structures, and constraints are compatible between the source and target databases.

Perform necessary data transformations during the export/import process if required.

**5. Test the Migration:**

Before performing a large-scale migration, test the process with a small dataset to identify and resolve any issues.

**6. Monitor and Verify:**

Monitor the data transfer process and verify the integrity of the data in the target database.

**7. Backup:**

Before making any changes, ensure that you have backups of both the source and target databases.

**8. Consider Security:**

Ensure that the data transfer process is secure, especially when moving sensitive or personal information.

**Note:**

Always refer to the documentation of the specific databases you are working with for detailed instructions and best practices.

Remember that the exact steps may vary based on the databases you're using and the specific requirements of your migration.