SQL, Cursor, Database Backups and Restore, SQL Jobs, snapshot, SQL CLR (Day 10)

Cursors in SQL Server

Cursors in SQL Server are used to process the result set row-by-row, similar to a loop in programming. Cursors allow you to perform operations on each row individually rather than processing the entire set at once. Here's a breakdown of each step with an example.

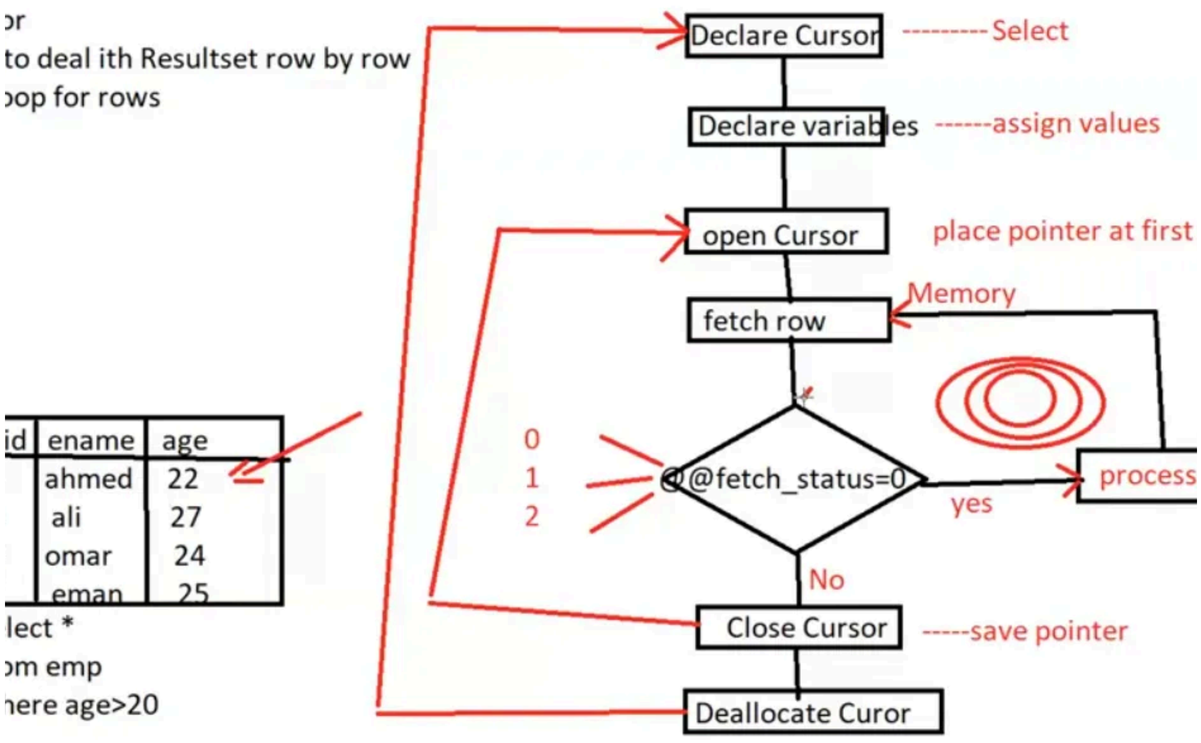
Example Scenario

Assume we have a table called `Employees`:

```sql
CREATE TABLE Employees ( EmployeeID INT, FirstName VARCHAR(50), LastName VARCHAR(50), Department VARCHAR(50) ); INSERT INTO Employees (EmployeeID, FirstName, LastName, Department) VALUES (1, 'John', 'Doe', 'Sales'), (2, 'Jane', 'Smith', 'HR'), (3, 'Mike', 'Johnson', 'IT');
```

Steps for Using a Cursor

1. **Declare the Cursor**: Define the cursor with a `SELECT` statement to specify which rows the cursor will iterate through.
2. **Declare Variables**: Define variables to store each row's values when fetched.
3. **Open the Cursor**: Move the cursor pointer to the first row of the result set.
4. **Fetch the Row**: Retrieve the current row's data into the variables.
5. **Check `@@FETCH_STATUS`**: A built-in variable that indicates the fetch status.
   - `0`: Row was successfully fetched.
   - `1`: Row could not be fetched (end of rows or an error).
   - `2`: Cursor is at a position that cannot be fetched.
6. **Close the Cursor**: Close the cursor when done (it can be re-opened if needed).
7. **Deallocate the Cursor**: Release the memory associated with the cursor.



id	ename	age
	ahmed	22
	ali	27
	omar	24
	eman	25

```
lect *
om emp
here age>20
```

Example of a Cursor Implementation

```
-- Step 1: Declare the cursor DECLARE employee_cursor CURSOR FOR SELECT Emplo
yeeID, FirstName, LastName FROM Employees; -- Step 2: Declare variables for t
he fetched values DECLARE @EmployeeID INT; DECLARE @FirstName VARCHAR(50); DE
CLARE @LastName VARCHAR(50); -- Step 3: Open the cursor OPEN employee_cursor;
-- Step 4: Fetch the first row FETCH NEXT FROM employee_cursor INTO @Employee
ID, @FirstName, @LastName; --or this query -->FETCH employee_cursor INTO @Emp
loyeeID, @FirstName, @LastName; -- Step 5: Loop through the rows using @@FETC
H_STATUS WHILE @@FETCH_STATUS = 0 BEGIN -- Perform operations with the fetche
d data (for example, printing) PRINT 'EmployeeID: ' + CAST(@EmployeeID AS VAR
CHAR) + ', Name: ' + @FirstName + ' ' + @LastName; -- Fetch the next row FETC
H NEXT FROM employee_cursor INTO @EmployeeID, @FirstName, @LastName; END; --
Step 6: Close the cursor CLOSE employee_cursor; -- Step 7: Deallocate the cur
sor DEALLOCATE employee_cursor;
```

Explanation

- `DECLARE employee_cursor CURSOR FOR`: Defines the cursor with the query.
- `FETCH NEXT FROM employee_cursor INTO @EmployeeID, @FirstName, @LastName;`: Fetches the next row from the cursor into the variables.
- `WHILE @@FETCH_STATUS = 0`: Continues looping as long as there are rows to fetch.
- `CLOSE` and `DEALLOCATE`: Free up resources when finished with the cursor.

Step-by-Step Cursor Processing (You can skip this)

1. **Declare the Cursor**

   Define the cursor and specify the `SELECT` query that determines the rows it will iterate over.

2. **Declare Variables**

   Create variables to store the values from each fetched row.

3. **Open the Cursor**

   Move the cursor to the first row in the result set and prepare it for fetching.

4. **Fetch the First Row**

   Retrieve the first row from the cursor and assign its values to the declared variables.

5. **Check `@@FETCH_STATUS`**
   - If `@@FETCH_STATUS = 0`, proceed to process the fetched row.
   - If not, end the loop.
6. **Process Row Data**

   Perform the desired operation on the fetched data, such as printing or updating the variables.
7. **Fetch the Next Row**

   Retrieve the next row from the cursor.
8. **Repeat Steps 5–7**

   Continue fetching and processing rows while `@@FETCH_STATUS = 0`.
9. **Close the Cursor**

   When all rows are processed, close the cursor to save the pointer position (allowing it to be re-opened if needed).
10. **Deallocate the Cursor**

    Free the memory associated with the cursor to release resources.

Example 1: Simple Cursor for Reading or Updating

```
DECLARE EmployeeCursor CURSOR FOR SELECT EmployeeID, FirstName FROM Employees
FOR READ ONLY; DECLARE @EmployeeID INT, @EmployeeName VARCHAR(50); OPEN Emplo
yeeCursor; FETCH NEXT FROM EmployeeCursor INTO @EmployeeID, @EmployeeName; WH
ILE @@FETCH_STATUS = 0 BEGIN PRINT CONCAT('EmployeeID: ', @EmployeeID, ', Nam
e: ', @EmployeeName); FETCH NEXT FROM EmployeeCursor INTO @EmployeeID, @Emplo
yeeName; END; -- You can use PRINT or SET CLOSE EmployeeCursor; DEALLOCATE Em
ployeeCursor;
```

Example 2: Concatenate Employee Names with Comma Separation

```
DECLARE EmployeeNameCursor CURSOR FOR SELECT Name FROM Employees WHERE Name I
S NOT NULL FOR READ ONLY; DECLARE @EmployeeName VARCHAR(50), @ConcatenatedNam
es VARCHAR(MAX) = ''; OPEN EmployeeNameCursor; FETCH NEXT FROM EmployeeNameCu
rsor INTO @EmployeeName; WHILE @@FETCH_STATUS = 0 BEGIN SET @ConcatenatedName
s = CASE WHEN @ConcatenatedNames = '' THEN @EmployeeName ELSE CONCAT(@Concate
natedNames, ', ', @EmployeeName) END; FETCH NEXT FROM EmployeeNameCursor INTO
@EmployeeName; END; CLOSE EmployeeNameCursor; DEALLOCATE EmployeeNameCursor;
SELECT @ConcatenatedNames AS CommaSeparatedNames;
```

Example 3: Update Salary Based on Conditions

```
DECLARE SalaryUpdateCursor CURSOR FOR SELECT Salary FROM Employees FOR UPDAT
E; DECLARE @Salary DECIMAL(18, 2); OPEN SalaryUpdateCursor; FETCH NEXT FROM S
alaryUpdateCursor INTO @Salary; WHILE @@FETCH_STATUS = 0 BEGIN IF @Salary < 2
000 UPDATE Employees SET Salary = @Salary * 1.20 WHERE CURRENT OF SalaryUpdat
eCursor; ELSE UPDATE Employees SET Salary = @Salary * 1.10 WHERE CURRENT OF S
alaryUpdateCursor; FETCH NEXT FROM SalaryUpdateCursor INTO @Salary; END; CLOS
E SalaryUpdateCursor; DEALLOCATE SalaryUpdateCursor;
```

Example 4: Count Occurrences of "mostafa" after "omar"

```
DECLARE NameSequenceCursor CURSOR FOR SELECT Name FROM Employees FOR READ ONL
Y; DECLARE @CurrentName VARCHAR(50), @PreviousName VARCHAR(50) = '', @Mostafa
AfterOmarCount INT = 0; OPEN NameSequenceCursor; FETCH NEXT FROM NameSequence
Cursor INTO @CurrentName; WHILE @@FETCH_STATUS = 0 BEGIN IF @PreviousName =
'omar' AND @CurrentName = 'mostafa' SET @MostafaAfterOmarCount += 1; SET @Pre
viousName = @CurrentName; FETCH NEXT FROM NameSequenceCursor INTO @CurrentNam
e; END; CLOSE NameSequenceCursor; DEALLOCATE NameSequenceCursor; SELECT @Most
afaAfterOmarCount AS MostafaAfterOmarCount;
```

Types of Backups

1. **Full Backup**
    - Captures a complete copy of the database, including all data from its creation up to the current moment.
    - Backs up the data from the primary data file (`.mdf`).
2. **Differential Backup**
    - Saves only the changes made since the last full backup, up to the current moment.
    - Requires at least one prior full backup to function correctly.
    - Also backs up data from the primary data file (`.mdf`), but only the changes since the last full backup.
3. **Transaction Log Backup**
    - Captures all transactions that occurred from the last backup, regardless of type (full, differential, or another transaction log), up to the current moment.
    - Uses the transaction log file (`.ldf`), enabling point-in-time recovery by keeping a record of each transaction and its timestamp.
4. **File Group Backup**
    - Allows backing up specific file groups within a database. This can be useful for large databases where only certain portions are frequently updated.
    - Enables selective backup and restoration of parts of the database rather than the entire database.
    - Data is backed up from specified file groups rather than the entire database.

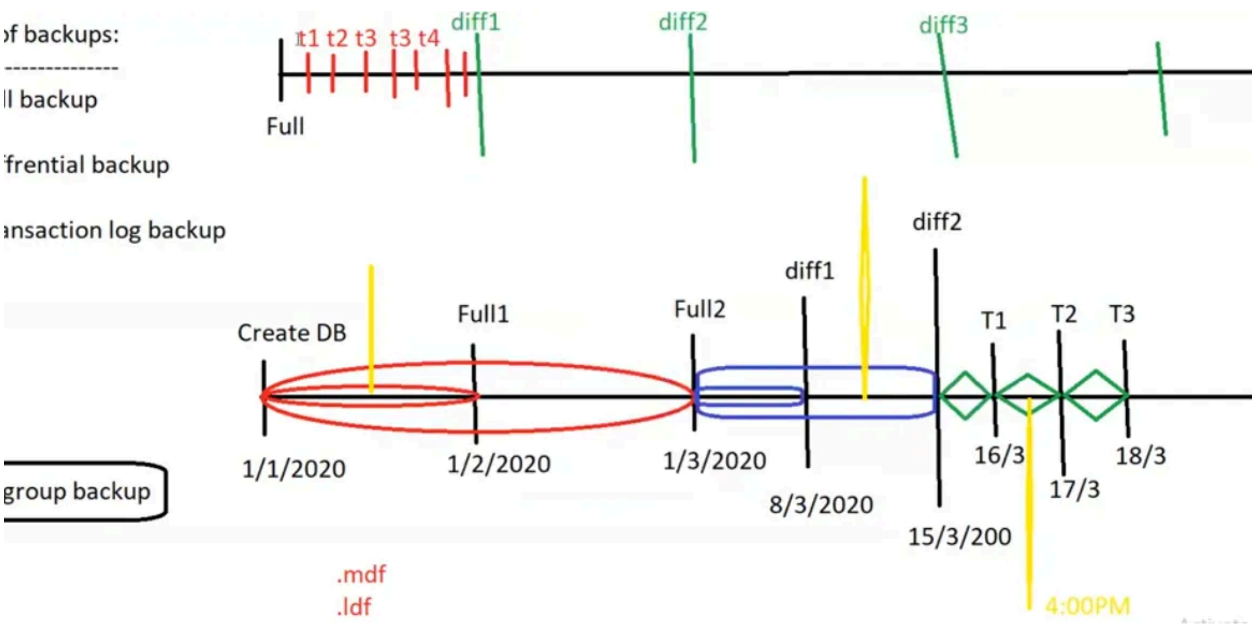
Backup Example and Sequence

1. **Database Creation Date:** `1/1/2020`

2. **Backup Sequence:**
   - **Full1**: `1/2/2020` (12 PM)
   - **Full2**: `1/3/2020` (12 PM)
   - **Differential1**: `8/3/2020` (12 PM)
   - **Differential2**: `15/3/2020` (12 PM)
   - **TransactionLog1**: `16/3/2020` (12 PM)
   - **TransactionLog2**: `17/3/2020` (12 PM)
   - **TransactionLog3**: `18/3/2020` (12 PM)

Examples of Restoring Data

1. **Restoring up to 8/3/2020:**
   - You would use **Full2** (1/3/2020) and **Diff1** (8/3/2020).
   - This restores all data up to 8/3/2020.
2. **Restoring up to 17/3/2020:**
   - You would need **Full2** (1/3/2020), **Diff2** (15/3/2020), **T1** (16/3/2020), and **T2** (17/3/2020).
   - This sequence allows for the restoration of all data up to 17/3/2020.
3. **Restoring on an unavailable date:**
   - **Example:** Restoring on `15/1/2020` or `10/3/2020` is not possible because no backups are available for these dates.
   - **Explanation:** Since the first full backup was made on `1/2/2020`, and the next differential on `8/3/2020`, these dates cannot be restored because they fall outside of available backup intervals.
4. **Restoring at a specific time on a date:**
   - **Example:** To restore data up to `17/3/2020` at `4:00 PM`, use **Full2** (1/3/2020), **Diff2** (15/3/2020), **T1** (16/3/2020), and **T2** (up to 4:00 PM).
   - **Explanation:** Transaction log backups record the exact time of each transaction, allowing recovery to a specific time if needed.



Example of Performing a Backup and Restore

Step-by-Step Example:

1. **Create Table and Insert Data**
   - Create a table in your database and insert initial data:

```
CREATE TABLE TestTable (ID INT PRIMARY KEY, Data VARCHAR(50)); INSERT
INTO TestTable VALUES (1, 'Row 1'), (2, 'Row 2'), (3, 'Row 3'), (4,
'Row 4'), (5, 'Row 5');
```

2. **Take a Full Backup**
   - Right-click on your database, select **Tasks** > **Back Up**
   - Choose **Full** as the backup type and complete the backup.
   - This captures all data in the database up to this point (5 rows).

3. **Insert More Data and Take a Differential Backup**
    - Add more rows to the table:

      ```
      INSERT INTO TestTable VALUES (6, 'Row 6'), (7, 'Row 7'), (8, 'Row
      8'), (9, 'Row 9');
      ```

    - Now, take a **Differential Backup** by following the same steps but selecting **Differential** as the backup type.
    - This captures changes since the last full backup, which includes the new 4 rows (up to 9 rows in total).
4. **Add More Data and Take a Transaction Log Backup**
    - Insert one more row to the table:

      ```
      INSERT INTO TestTable VALUES (10, 'Row 10');
      ```

    - Take a **Transaction Log Backup**, capturing only the changes since the last backup (the addition of Row 10).
5. **Delete Database**
    - For simulation purposes, delete the database (optional, depending on your test setup).
6. **Restore the Database**
    - Right-click on **Databases**, select **Restore Database...** and choose the appropriate backup file (e.g., `mytest.bak`).
    - Select **Full**, **Differential**, and **Transaction Log** backups from the available list.
7. **Restore to a Specific Point in Time**
    - Click on the **Timeline** button.
    - Set the timeline to a point in time just before the last transaction (for instance, 3 minutes before the current time).
    - This should restore the database with only the first 9 rows, excluding Row 10 added in the final transaction log backup.

Performing Backups via SQL Query and Automating Backups

1. Backup as a Query

You can back up your database using a T-SQL query:

```
BACKUP DATABASE CompanyDB TO DISK = 'G:\\Data\\CompanyDB.bak';
```

2. Schedule a Recurring Job in SQL Server Agent

SQL Server Agent provides a way to automate tasks, such as regular backups. Here's how to set up a daily backup job:

1. **Open SQL Server Management Studio (SSMS)** and expand **SQL Server Agent** in the Object Explorer.
2. **Right-click on Jobs** and select **New Job...** to create a new job.
3. **In the New Job window**, give your job a name (e.g., `Daily Backup Job`).
4. **Define the Backup Step:**
    - Go to the **Steps** page and click **New...** to add a new step.
    - Name the step (e.g., `Full Database Backup`).
    - Set **Type** to **Transact-SQL script (T-SQL)**.
    - In the **Command** field, enter your backup query:

      ```
      BACKUP DATABASE mytest TO DISK = 'G:\\Data Engineering\\mytest.bak' W
      ITH INIT;
      ```

    - Click **OK** to save the step.

5. **Schedule the Job:**
    - Go to the **Schedules** page and click **New...** to create a new schedule.
    - Name the schedule (e.g., `Daily Backup at 12 PM`).
    - Set **Schedule Type** to **Recurring**.
    - Configure the schedule to run **Daily** at **12:00 PM**.
    - Click **OK** to save the schedule.
6. **Save and Start the Job:**
    - Click **OK** to complete and save the job setup.
    - You can also start the job immediately by right-clicking it and selecting **Start Job at Step...**.

This job will now automatically back up the `mytest` database to the specified path (`G:\\Data Engineering\\mytest.bak`) every day at 12:00 PM.

Identity Column in SQL Server

1. **Definition**: An **IDENTITY** column is a column in a database table that automatically generates a unique value for each row inserted into the table. This is typically used for primary keys.
2. **Automatic Generation**:
    - When you insert rows into a table with an IDENTITY column, SQL Server automatically generates the next value in the sequence.
3. **Behavior on Deletion**:
    - If you insert rows with values from 1 to 10 and then delete rows 5 to 9, the remaining IDs will be 1, 2, 3, 4, 10. This creates gaps in the sequence.
4. **Inserting Specific Identity Values**:
    - If you need to insert a specific value into an IDENTITY column (for instance, to fill in gaps), you must first enable the ability to insert explicit values.

Syntax for Enabling Identity Insert

- **Enable Identity Insert**:

  ```
  SET IDENTITY_INSERT table_name ON;
  ```

- **Insert Statement**:
  - Now you can explicitly insert values into the identity column.

  ```
  INSERT INTO table_name (identity_column, other_column1, other_column2) VA
  LUES (value_for_identity, value1, value2);
  ```

- **Disable Identity Insert**:

  ```
  SET IDENTITY_INSERT table_name OFF;
  ```

Example

```
-- Enable identity insert SET IDENTITY_INSERT Employees ON; -- Insert specifi
c identity value INSERT INTO Employees (EmployeeID, Name) VALUES (5, 'John Do
e'); -- This inserts '5' even if it is in the gap. -- Disable identity insert
SET IDENTITY_INSERT Employees OFF;
```

Notes

- **Use Caution**: When enabling identity insert, ensure you do not create conflicts with existing values in the identity column.
- **Only One Table at a Time**: You can only set `IDENTITY_INSERT` to `ON` for one table in a session at a time.

Types of INSERT Operations in SQL Server

1. **Simple Insert**:
    - This is used to insert a single row into a table.

```sql
INSERT INTO Employees (EmployeeID, Name, Age) VALUES (1, 'Ahmed', 30);
```

2. **Insert Constructor**:
    - This allows you to insert multiple rows into a table in a single statement.

```sql
INSERT INTO Employees (EmployeeID, Name, Age) VALUES (2, 'Omar', 25), (3,
'Sara', 28), (4, 'Ali', 32);
```

3. **Insert Based on SELECT**:
    - You can insert rows into a table by selecting them from another table.

```sql
INSERT INTO NewEmployees (EmployeeID, Name, Age) SELECT EmployeeID, Name,
Age FROM Employees WHERE Age > 30;
```

4. **Insert Based on EXECUTE (with Stored Procedure)**:
    - This allows you to insert data using a stored procedure.

```sql
EXEC InsertEmployee @EmployeeID = 5, @Name = 'Mona', @Age = 29;
```

5. **BULK INSERT**:
    - This method is used to insert a large amount of data from a file into a table.
    - **Steps**:
        1. Create a text file with the data formatted as needed. For example:

           ```
           222,ahmed,21 333,omar,23 444,Amr,24
           ```

        2. Use the following SQL command to perform a BULK INSERT:

    ```
    BULK INSERT Employees FROM 'G:\\DataEngineering\\Amr.txt' WITH ( FIELDTER
    MINATOR = ',', -- Specify the field delimiter ROWTERMINATOR = '\\n', -- S
    pecify the row delimiter (optional, default is '\\n') FIRSTROW = 2 -- Ski
    p the header row if necessary (optional) );
    ```

Notes

- Ensure that the data types in the text file match the corresponding columns in the table.
- The `FIELDTERMINATOR` specifies how the fields in each row are separated, while the `ROWTERMINATOR` specifies how rows are separated.
- Consider permissions and file path accessibility when using BULK INSERT.

SQL Server Snapshots

Definition:

A snapshot in SQL Server is a read-only, static view of a database (similar to taking a screenshot) at a specific point in time. It maintains a pointer to the original data and captures the state of the database when the snapshot is created.

Creating a Database Snapshot

To create a database snapshot, you can use the following SQL command:

```sql
CREATE DATABASE itisnap ON ( NAME = 'CompanyDB', FILENAME = 'G:\\DataEngineer
ing\\CompanyDB_Snapshot.ss' ) AS SNAPSHOT OF CompanyDB;
```

Key Features of Database Snapshots

- **Read-Only**: Once a snapshot is created, it is read-only. You cannot modify the snapshot itself.
- **Data Integrity**: The snapshot captures the state of the data at the time it was taken. If changes are made to the original database, the snapshot retains the original data for read access.
- **Restoration**: If you need to restore data to its previous state, you can revert to the snapshot.

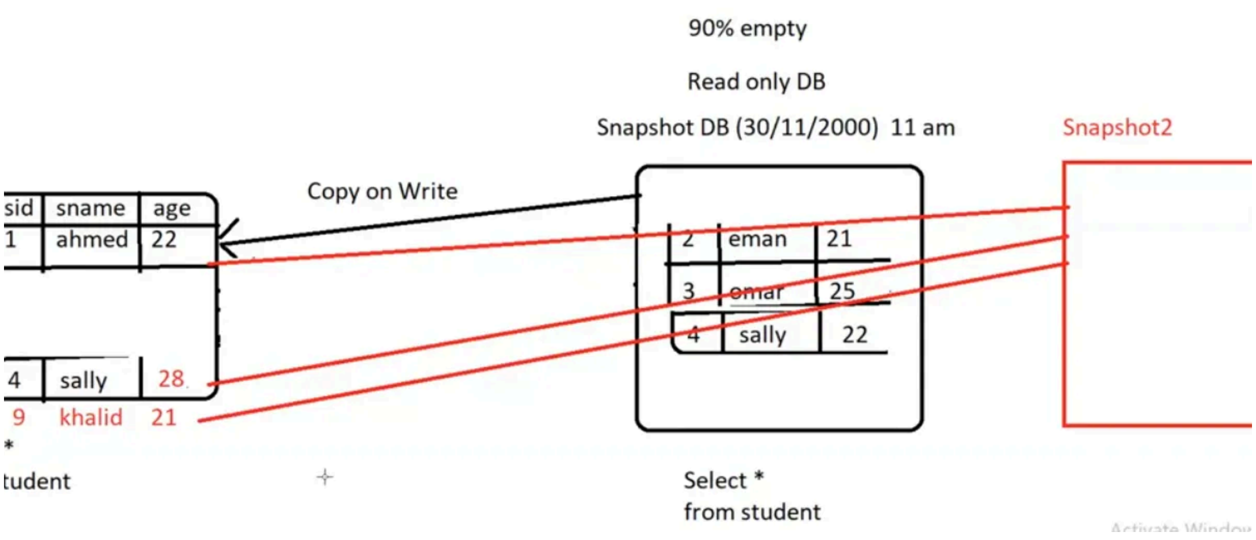
Restoring from a Snapshot

To restore a database from a snapshot, you can use the following command:

```sql
RESTORE DATABASE CompanyDB FROM DATABASE_SNAPSHOT = 'itisnap';
```

Benefits of Using Snapshots

- **Data Recovery**: Snapshots can be used to recover data after accidental deletions or updates.
- **Reporting**: You can generate reports based on the snapshot data without affecting the performance of the live database.
- **Testing**: Snapshots are useful for testing purposes, as they allow you to explore the state of the database at a specific point without altering the actual data.



SQLCLR (SQL Server Common Language Runtime)

Definition:

SQLCLR allows you to create functions or new data types in SQL Server using .NET languages such as C# or VB.NET This feature enables you to extend SQL Server's capabilities by leveraging the power of .NET Framework.

Enabling SQLCLR

Before creating SQLCLR functions or types, you need to enable CLR integration in SQL Server:

```
EXEC sp_configure 'clr_enable', 1; RECONFIGURE;
```

Creating a User-Defined Function (UDF)

1. **Open Visual Studio**: Start a new SQL Server Database Project.
2. **Add a New Item**:
   - Right-click on the Database project.
   - Select **Add** > **New Item**.
   - Choose **SQL CLR C# User Defined Function**.
3. **Implement the Function**: Write the following C# code in the generated class:

```
using System; using System.Data.SqlTypes; using Microsoft.SqlServer.Server; p
ublic partial class UserDefinedFunctions { [SqlFunction] public static SqlInt
16 SumTwoIntegers(SqlInt16 x, SqlInt16 y) { return x + y; } }
```

1. **Publish the Function**:
   - Right-click on the database project and select **Publish**.
   - Fill in the connection details:
     - **Server Name**: `DESKTOP-DLB8G1R\\SQL2022`
     - **Database Name**: `CompanyDB`
   - Click **OK**, then **Publish**.
2. **Verify the Function**:
   - Go to the `CompanyDB` database.
   - Navigate to **Programmability** > **Scalar Functions** to see the published function.

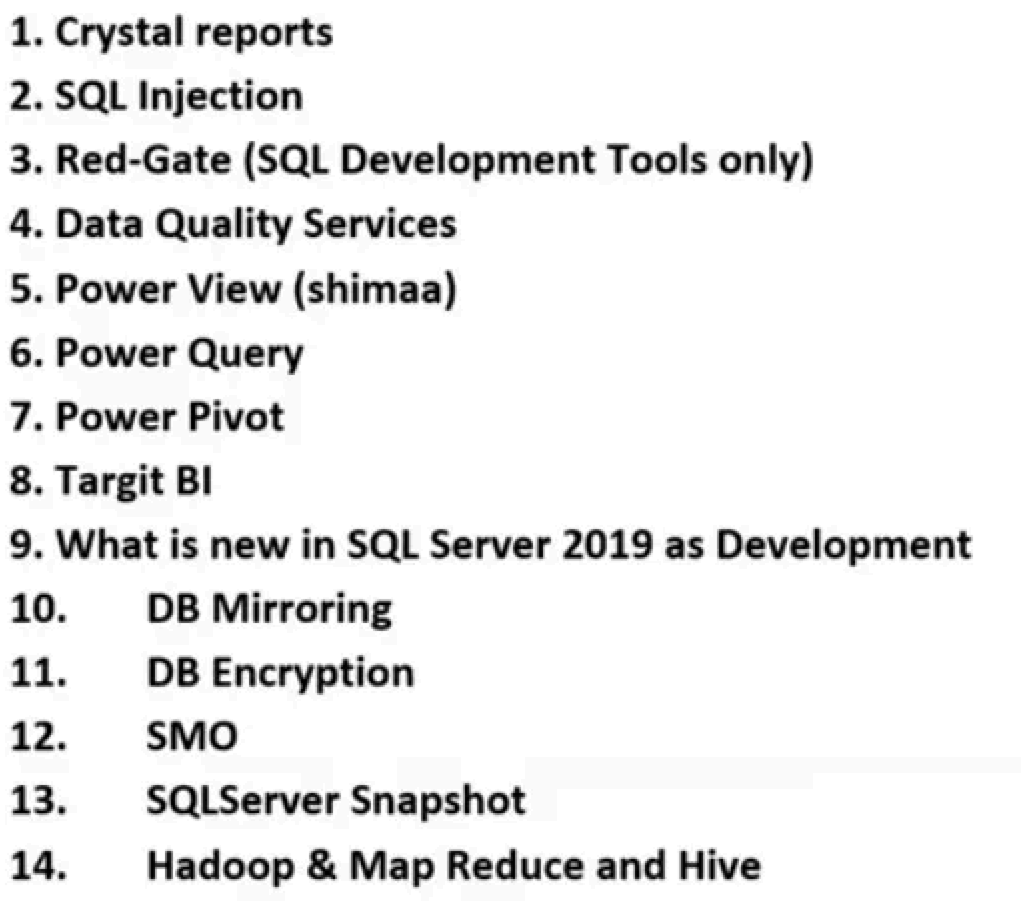
Creating a User-Defined Data Type (UDT)

1. **Open Visual Studio**: Start a new SQL Server Database Project.
2. **Add a New Item**:
   - Right-click on the Database project.
   - Select **Add** > **New Item**.
   - Choose **SQL CLR C# User Defined Type**.
3. **Implement the Data Type**: Write the necessary code for your custom data type.

Benefits of Using SQLCLR

- **Enhanced Performance**: Complex calculations and operations can be performed in .NET, which can be more efficient than T-SQL.
- **Access to .NET Libraries**: Leverage the extensive .NET libraries for advanced functionality not available in T-SQL.
- **Improved Code Reusability**: Write reusable code in .NET languages and utilize it across different SQL Server databases.

Search for those topics



1. Crystal reports
2. SQL Injection
3. Red-Gate (SQL Development Tools only)
4. Data Quality Services
5. Power View (shimaa)
6. Power Query
7. Power Pivot
8. Targit BI
9. What is new in SQL Server 2019 as Development
10. DB Mirroring
11. DB Encryption
12. SMO
13. SQLServer Snapshot
14. Hadoop & Map Reduce and Hive