# MSSQL Notes

## step-by-step guide to installing SQL Server with explanations of each option/choice in the wizard

Download the SQL Server Installer: Before you can install SQL Server, you need to download the SQL Server installer from the Microsoft website.

Launch the SQL Server Installer: Double-click on the downloaded file to launch the SQL Server installer.

Choose Installation Type: In the SQL Server Installation Center, choose "Installation" from the left-hand menu, and then select "New SQL Server stand-alone installation or add features to an existing installation". This will start the installation wizard.

Enter Product Key: Enter your product key or select the evaluation edition and click "Next".

Accept the License Terms: Read and accept the license terms by selecting the checkbox and click "Next".

Choose Features: Select the features you want to install, such as the Database Engine Services, Analysis Services, Integration Services, and Reporting Services. You can also choose to install additional shared components, such as SQL Server Management Studio and SQL Server Data Tools. Click "Next".

Specify Instance Configuration: Choose the instance type, named or default, and specify the instance name if you choose named. Specify the instance root directory and the shared feature directory. Click "Next".

Server Configuration: Specify the account name and password for the SQL Server Database Engine and SQL Server Agent services. Choose the startup type and collation. Click "Next".

Database Engine Configuration: Specify authentication mode, either Windows Authentication or Mixed Mode. If you choose Mixed Mode, you need to provide a password for the SQL Server system administrator (sa) account. Click "Next".

Analysis Services Configuration: Choose the mode, either multidimensional and data mining mode or tabular mode. Specify the administrator account for the Analysis Services instance. Click "Next".

Reporting Services Configuration: Choose the mode, either Native mode or SharePoint integrated mode. Specify the instance ID and the report server database name. Click "Next".

Error Reporting and Usage: Specify if you want to enable Error Reporting and Usage data collection, and click "Next".

Ready to Install: Review your selections and click "Install" to start the installation process.

Installation Progress: The installation process will begin and show the progress for each feature being installed.

Complete the Installation: Once the installation is complete, click "Close" to exit the installer.

Congratulations, you have successfully installed SQL Server! You can now launch SQL Server Management Studio to connect to the database and start working with your data.

# Various configurations of a SQL Server

## Instance Configuration

An instance is a single installation of SQL Server on a machine. The instance configuration determines how the SQL Server instance is installed, named, and configured. During installation, you can choose the instance type, named or default, and specify the instance name if you choose named. You can also specify the instance root directory and the shared feature directory.

### Server Configuration:

The server configuration determines the settings for the SQL Server Database Engine and SQL Server Agent services. You can specify the account name and password for these services, choose the startup type, and set the collation. You can also specify the maximum memory usage for SQL Server, configure server-wide settings, and set up data directories for the SQL Server instance.

### Database Engine Configuration:

The database engine configuration determines how the SQL Server Database Engine operates. You can specify the authentication mode, either Windows Authentication or Mixed Mode. If you choose Mixed Mode, you need to provide a password for the SQL Server system administrator (sa) account. You can also configure database mail, network protocols, and server-level security settings.

### Analysis Services Configuration:

Analysis Services is a component of SQL Server that provides online analytical processing (OLAP) and data mining functionality. The Analysis Services configuration determines the mode, either multidimensional and data mining mode or tabular mode. You can also specify the administrator account for the Analysis Services instance and configure server-level settings.

### Reporting Services Configuration:

Reporting Services is a component of SQL Server that provides reporting functionality. The Reporting Services configuration determines the mode, either Native mode or SharePoint integrated mode. You can specify the instance ID and the report server database name, and configure other settings such as email settings, scale-out deployment, and authentication.

### Integration Services Configuration:

Integration Services is a component of SQL Server that provides data integration and transformation services. The Integration Services configuration allows you to configure the package store, setup logging, and configure package execution settings.

### SQL Server Management Studio Configuration:

SQL Server Management Studio is the main tool for managing and administering SQL Server. The SQL Server Management Studio configuration determines the settings for the tool, such as the font and color scheme, and allows you to set options for various features, such as Object Explorer, Query Editor, and IntelliSense.

These are the main configurations of a SQL Server. By configuring each of these components, you can set up a robust and secure SQL Server environment that meets your organization's needs.

## Database Fundamentals

*Tables*: The data in a relational database is organized into tables, which consist of rows and columns.

*Rows*: Each row represents a single record in the table and contains a unique identifier known as a primary key.

*Columns*: Each column represents a specific attribute or characteristic of the data being stored, such as a customer's name or address.

*Relationships*: Relationships are established between tables to create a more complex data model. The most common relationship is a one-to-many relationship, where each record in one table can be associated with many records in another table.

*Primary key*: A primary key is a unique identifier that is used to distinguish one record from another in a table.

*Foreign key:* A foreign key is a column in one table that refers to the primary key of another table, establishing a relationship between the two tables.

*Indexes*: Indexes are used to improve the performance of queries by creating a data structure that allows the database to quickly locate specific data.

*Constraints*: Constraints are rules that are enforced by the database to ensure data integrity. Examples of constraints include unique constraints, which ensure that a column contains only unique values, and foreign key constraints, which ensure that data in related tables remains consistent.

*Candidate key*: A candidate key is a column or set of columns in a table that could potentially be used as a primary key. It must be unique and cannot contain null values.

*Superkey*: A superkey is a set of one or more columns in a table that uniquely identifies each record. It may or may not be minimal, meaning it may include extraneous columns that are not strictly necessary to ensure uniqueness.

*Alternate key*: An alternate key is a candidate key that is not selected as the primary key for a table.

*Composite key*: A composite key is a primary key that consists of two or more columns in a table. It is used when no single column can uniquely identify a record, but the combination of columns can.

*Surrogate key*: A surrogate key is a primary key that is artificially generated by the database system rather than being based on a natural attribute of the data. It is often used when a natural key is too complex or volatile to use as a primary key.

# Transaction and ACID

- Transaction: A transaction is a sequence of one or more database operations that are treated as a single unit of work. A transaction is typically used to ensure data consistency and integrity by ensuring that all the operations in the sequence either succeed or fail as a group.
- Implicit Transaction: Not specified explicitly. Ex: Locking of row during update.
- Explicit Transaction

  ```
  BEGIN TRANSACTION;
  Select @@trancount –Gives number of nested transaction, here 1
  UPDATE my_table
  SET column1 = 'new_value'
  WHERE column2 = 'condition';
  COMMIT TRANSACTION;
  ```

- ACID properties: ACID is an acronym that stands for Atomicity, Consistency, Isolation, and Durability. These properties define the requirements that a database system must meet to ensure reliable transactions:
  - Atomicity: Atomicity requires that a transaction is treated as a single indivisible unit of work. Either all of the operations in the transaction are completed successfully, or none of them are. If a transaction fails, all changes made by the transaction must be rolled back to their previous state.
  - Consistency: Consistency requires that a transaction brings the database from one valid state to another valid state. In other words, the database must satisfy all the integrity constraints and rules defined for it.
  - Isolation: Isolation requires that multiple transactions can run concurrently without interfering with each other. Each transaction should appear to execute in isolation, meaning that each transaction should see the database as if it were the only one modifying it. Isolation can be achieved through locking and other concurrency control mechanisms.
  - Durability: Durability requires that once a transaction has been committed, its changes are permanent and will survive subsequent system failures, crashes, or other types of disruption. This is typically achieved by writing the transaction's changes to non-volatile storage such as disk.

## Locks

In Microsoft SQL Server (MSSQL), locks and lock types play a crucial role in concurrency control to ensure data integrity and prevent conflicts between concurrent transactions. Let's explore locks and the different lock types in MSSQL:

1. Locks:
   - A lock is a mechanism used to control access to a resource (e.g., a table, a row, or a page) to ensure data consistency.
   - When a transaction accesses a resource, it requests and acquires an appropriate lock on that resource.

- Locks can be held for a duration of a transaction or released earlier depending on the isolation level and locking hints used.

2. Lock Types:
    - Shared Lock (S): Also known as a "read lock," multiple transactions can acquire shared locks on a resource simultaneously. These locks are compatible with other shared locks but incompatible with exclusive locks.
    - Exclusive Lock (X): Also known as a "write lock," only one transaction can acquire an exclusive lock on a resource at a time. Exclusive locks are incompatible with shared and other exclusive locks.
    - Intent Lock: Indicates that a transaction intends to acquire a lock at a higher level of granularity (e.g., page or table) than the current lock. It helps coordinate locks at different levels and prevents deadlocks.
    - Schema Lock: A lock on a schema that prevents concurrent modification of the schema by multiple transactions.
    - Update Lock (U): A special type of lock used in certain scenarios to optimize concurrency when both read and write operations are expected. It is compatible with shared locks but incompatible with exclusive locks.
    - Key-Range Lock: A lock acquired on a range of keys in an index to ensure data consistency during operations like index scans or range updates.
3. Lock Compatibility Matrix: The following table illustrates the compatibility between different lock types in MSSQL:

| | **Shared (S)** | **Exclusive (X)** | **Intent** | **Schema** | **Update (U)** | **Key-Range** |
|---|---|---|---|---|---|---|
| Shared (S) | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ |
| Exclusive (X) | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| Intent | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ |
| Schema | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ |
| Update (U) | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ |
| Key-Range | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ |

4. ✓: Compatible ✗: Incompatible

## Isolation Levels

Isolation levels in Microsoft SQL Server (MSSQL) define the degree of concurrency and data consistency provided by the database system. By setting the isolation level, you can control how transactions interact with each other and ensure the desired balance between concurrency and data integrity. Here are the commonly used isolation levels in MSSQL:

1. Read Uncommitted:
    - Transactions at this isolation level are not required to wait for exclusive locks.
    - Dirty reads, non-repeatable reads, and phantom reads can occur.
    - It provides the highest level of concurrency but the lowest data integrity.
2. Read Committed (default):
    - Transactions at this isolation level wait for exclusive locks but allow reading committed data.
    - Dirty reads are prevented, but non-repeatable reads and phantom reads can occur.
    - It strikes a balance between concurrency and data integrity.
3. Repeatable Read:
    - Transactions at this isolation level hold shared locks on all accessed data until the transaction completes.
    - Dirty reads and non-repeatable reads are prevented, but phantom reads can occur.
    - It offers higher data integrity but may lead to increased blocking.
4. Serializable:
    - Transactions at this isolation level hold range locks on all accessed data until the transaction completes.
    - It provides the highest level of data integrity by preventing dirty reads, non-repeatable reads, and phantom reads.
    - It offers the least concurrency and may lead to increased blocking.
5. Snapshot Isolation:
    - Transactions at this isolation level use row versioning to provide a consistent snapshot of the database as of the start of the transaction.
    - It prevents dirty reads, non-repeatable reads, and phantom reads.
    - It allows for high concurrency by avoiding locks, but it may result in increased tempdb usage.

Each isolation level provides a trade-off between concurrency and data consistency. It is crucial to carefully select the appropriate isolation level based on your application's requirements and the nature of concurrent transactions. Additionally, MSSQL also supports setting isolation levels at the transaction level using the **SET TRANSACTION ISOLATION LEVEL** statement or at the connection level through database configuration options.

## Normalization

- Normalization: Normalization is the process of organizing data in a relational database to reduce redundancy and improve data integrity. It involves splitting large tables into smaller tables and defining relationships between them.
- Normal Forms: Normal Forms are a set of guidelines for ensuring that a database is well-designed and properly organized. There are several normal forms, each with increasing levels of normalization:
    - First Normal Form (1NF): A table is in 1NF if it has no repeating groups of data and all the columns contain atomic values (i.e., cannot be further subdivided).

- Second Normal Form (2NF): A table is in 2NF if it is in 1NF and all non-key attributes depend on the entire primary key (i.e., there are no partial dependencies).
- Third Normal Form (3NF): A table is in 3NF if it is in 2NF and all non-key attributes depend only on the primary key and not on any other non-key attributes (i.e., there are no transitive dependencies).
- Boyce-Codd Normal Form (BCNF): A table is in BCNF if it is in 3NF and every determinant (i.e., attribute that determines another attribute) is a candidate key (i.e., a minimal set of attributes that uniquely identifies each record).
- Fourth Normal Form (4NF): A table is in 4NF if it is in BCNF and has no multi-valued dependencies (i.e., attributes that have multiple values for a single record).
- Fifth Normal Form (5NF): A table is in 5NF if it is in 4NF and has no join dependencies (i.e., attributes that can be derived from other attributes in the same table).

## Creating a database

- Open Microsoft SQL Server Management Studio (SSMS) and connect to the SQL Server instance where you want to create the database.
- Right-click on the "Databases" folder and select "New Database..."
- In the "New Database" dialog box, enter a name for the database in the "Database name" field.
- Optionally, select a collation for the database. The collation determines how string data is sorted and compared. The default collation is usually sufficient.
- Optionally, set the initial size of the database and the growth options for the database files. This includes specifying the initial size of the data and log files, as well as the rate at which they should grow.
- Optionally, set the recovery model for the database. The recovery model determines how the database should handle transactions and backups. The three main recovery models are Simple, Full, and Bulk-Logged.
- Optionally, set other database options such as the compatibility level, containment type, and filestream options. These options affect how the database behaves and can be set based on your specific requirements.
- Click the "OK" button to create the database with the specified options.

## System Databases

In Microsoft SQL Server (MSSQL), there are two types of databases: user databases and system databases.

1. User Databases: These are the databases that users create to store their application data. User databases are created by users and can be customized to meet the specific needs of their applications. They can contain one or more tables, views, stored procedures, and other database objects.
2. System Databases: These are the databases that are created by the SQL Server installation process and are used to manage the system and provide services to users.

## Master Database

The Master database is the most important system database in MSSQL. It contains metadata information about all the databases, logins, and configurations on the SQL Server instance. The master database is the first database that is created when a SQL Server instance is installed, and it is constantly updated as new databases and objects are created on the server.

The Master database also stores information about system-level configuration settings such as server-level logins, endpoints, linked servers, and system objects. In addition, the Master database is critical for the proper functioning of SQL Server because it stores information about the SQL Server instance itself, such as startup parameters, collation settings, and database options.

Before the introduction of the Resource database, system objects were stored in individual system databases such as the Master, MSDB, and TempDB databases, which made upgrades and patches more complicated.

In summary, the Resource database is a read-only system database in MSSQL that contains all the system objects required for the proper functioning of SQL Server. It is a hidden system file that cannot be accessed or modified directly by users or administrators, and it separates the system objects from the user databases, making it easier to upgrade and patch SQL Server.

Here are some of the physical properties of the Master database in MSSQL:

1. File Extensions: The Master database consists of two physical files with the following extensions:
    - MDF (Master Data File): This is the primary data file of the Master database that stores the database schema, system objects, and metadata information.
    - LDF (Master Log File): This is the transaction log file of the Master database that stores all the transactional information, including rollback information, to ensure data consistency and integrity.
2. Database Size: The size of the Master database varies depending on the number of databases, objects, and configurations on the SQL Server instance. In general, the Master database is relatively small compared to the user databases on the server.
3. Autogrowth Settings: By default, the Master database is set to grow automatically when it reaches its maximum size. The autogrowth settings can be configured to specify how much the database should grow and when it should grow.
4. Recovery Model: The Master database uses the simple recovery model by default, which means that the transaction log is automatically truncated after each backup.
5. Compatibility Level: The compatibility level of the Master database depends on the version of SQL Server installed on the server. The compatibility level specifies which features and syntax are supported by the database engine.

Here are some of the restrictions on the Master database in MSSQL:

- Limited User Access: The Master database is a system database, and as such, only members of the sysadmin fixed server role have full access to it. Other users and roles may have some limited access to certain objects and metadata information in the Master database, but they cannot modify or update the database schema or system objects.

- No Backup/Restore: You cannot backup or restore the Master database directly using SQL Server Management Studio or T-SQL commands. Instead, you can only perform a full system backup that includes all the system and user databases on the server.
- No User Data Storage: The Master database is not designed for storing user data or user-specific configuration information. It only contains system objects, metadata, and server-level configurations.
- No User Modifications: Users cannot modify or update the system objects, schema, or configurations in the Master database directly. Any changes or updates to the system objects and configurations must be made using SQL Server Management Studio or T-SQL commands designed for that purpose.

Recommendations for Master Db:

1. Keep the database clean and avoid storing user data or user-specific configurations in it.
2. Backup and restore the system databases along with the user databases on a regular basis.
3. Limit access to the database to members of the sysadmin fixed server role.
4. Avoid modifying the system objects or schema directly, and use SQL Server Management Studio or T-SQL commands designed for that purpose.
5. Keep the database updated along with the SQL Server instance.

## Resource database

The Resource database is a read-only system database that was introduced in Microsoft SQL Server 2005. It contains all the system objects that are required for the proper functioning of SQL Server. The Resource database is located in the SQL Server instance's data folder, and it is a hidden system file that cannot be accessed or modified directly by users or administrators.

The Resource database contains all the system objects, such as system tables, stored procedures, functions, and views, that are used by SQL Server. These objects are used to provide fundamental functionalities to the SQL Server system, such as security, indexing, and data access.

The Resource database is read-only, meaning that users and administrators cannot modify or update the objects stored in the Resource database. Instead, any updates or modifications to the system objects are made through SQL Server updates and upgrades, service packs, or hotfixes.

One of the benefits of the Resource database is that it separates the system objects from the user databases, making it easier to upgrade and patch SQL Server. Before the introduction of the Resource database, system objects were stored in individual system databases such as the Master, MSDB, and TempDB databases, which made upgrades and patches more complicated.

## tempdb

- Tempdb is a system database in MSSQL that is used to store temporary user data, intermediate query results, and other temporary objects like temporary tables, indexes, and stored procedures.

- Tempdb is recreated every time SQL Server is started and is recreated whenever it is explicitly dropped or is corrupted. As a result, any data stored in Tempdb is temporary and is not persisted across server restarts.

- Tempdb is a shared resource that is used by all user databases on the SQL Server instance. This makes Tempdb a critical component of the SQL Server instance and can have a significant impact on performance.
- The size of Tempdb can be configured during installation, and it can be modified later using the ALTER DATABASE command. It is recommended to set the initial size of Tempdb to accommodate the expected workload.
- Tempdb is used for several operations such as sorting, hashing, creating temporary tables, storing table variables, and online index operations. These operations can consume a significant amount of disk space, especially during peak workloads, and can lead to performance degradation if not managed correctly.
- The performance of Tempdb can be improved by separating the data and log files onto different physical disks to reduce contention. It is also recommended to enable Instant File Initialization (IFI) for the Tempdb data and log files to improve performance during data file growth.
- Tempdb can also be monitored using several system views and performance counters to identify performance issues and optimize configuration.

Physical properties:

- Tempdb consists of one or more data files and one log file. The data files store temporary user data, intermediate query results, and temporary objects such as temporary tables, indexes, and stored procedures. The log file stores the transaction log information for Tempdb.
- By default, Tempdb has only one data file, but it is recommended to add additional data files to improve performance. The number of data files should be based on the number of logical processors on the SQL Server instance, with a recommended minimum of one data file per logical processor.
- The initial size of the data files for Tempdb can be set during installation, and it can be modified later using the ALTER DATABASE command. It is recommended to set the initial size of the data files to accommodate the expected workload.
- The Autogrowth setting for Tempdb data files can be configured to ensure that the database can grow as needed to accommodate workload spikes. However, it is recommended to pre-size the data files instead of relying solely on Autogrowth to avoid performance issues.
- The log file for Tempdb should be set to a reasonable size to ensure that it can handle the transaction log information for Tempdb. However, the log file for Tempdb is not typically a performance bottleneck.
- The physical location of the data and log files for Tempdb can be specified during installation or later using the ALTER DATABASE command. It is recommended to separate the data and log files onto different physical disks to reduce contention.

Restrictions:

- The database properties of Tempdb cannot be modified using the Database Properties dialog box in SQL Server Management Studio (SSMS). Instead, you must use the ALTER DATABASE command to modify the properties of Tempdb.
- The ownership of Tempdb cannot be changed, and it always belongs to the SQL Server system administrator (sa) login. This means that only the sa login and members of the sysadmin fixed server role have full control over Tempdb.
- Tempdb cannot be backed up or restored individually, and the backup and restore operations for Tempdb are included in the backup and restore operations for the entire SQL Server instance.
- You cannot create a database snapshot of Tempdb, and the database snapshot feature is not available for Tempdb.
- The Tempdb database cannot be dropped or renamed, and any attempt to do so will result in an error message.

Recommendations:

1. Increase the number of data files: Adding additional data files to Tempdb can improve performance by distributing the workload across multiple files and reducing contention for system allocation pages. It is recommended to have one data file per logical processor, with a minimum of four data files.
2. Pre-allocate space for data files: Pre-allocating space for data files can reduce contention for system allocation pages and improve performance. It is recommended to set the initial size of the data files to accommodate the expected workload.
3. Separate data and log files: Separating the data and log files onto different physical disks can reduce contention and improve performance.
4. Limit Autogrowth settings: Setting Autogrowth settings for Tempdb data files can ensure that the database can grow as needed to accommodate workload spikes. However, relying solely on Autogrowth can cause performance issues, so it is recommended to pre-size the data files instead.
5. Monitor disk I/O: Monitoring disk I/O can help identify performance issues and bottlenecks related to Tempdb. Tools like Performance Monitor and SQL Server Profiler can be used to monitor disk I/O.
6. Minimize usage of Tempdb: Minimizing the usage of Tempdb by avoiding the creation of unnecessary temporary objects, such as temporary tables and indexes, can help reduce contention and improve performance.

## Instant File Initialization

Instant File Initialization is a feature in Microsoft SQL Server that allows for faster creation and growth of data files by skipping the zeroing out of disk space. This feature can be particularly useful for Tempdb since it is recreated each time SQL Server is restarted, and can grow and shrink frequently based on workload.

By enabling Instant File Initialization for Tempdb, you can reduce the time it takes to create and grow the database files, which can help improve overall database performance. However, it is important to note that Instant File Initialization is only available on Windows Server editions, and requires the "Perform volume maintenance tasks" user right to be granted to the SQL Server service account.

To enable Instant File Initialization for Tempdb, follow these steps:

1. Grant the "Perform volume maintenance tasks" user right to the SQL Server service account.
2. Stop the SQL Server service.
3. Using Windows Explorer or the command line, navigate to the Tempdb files directory.
4. Right-click on each Tempdb data file, and select "Properties."
5. In the Properties dialog box, select the "Security" tab.
6. Click the "Edit" button, and add the SQL Server service account to the list of users and groups.
7. Check the "Allow" checkbox next to "Full control."
8. Click "OK" to close the Properties dialog box.
9. Start the SQL Server service.

## msdb database

MSDB is a system database in Microsoft SQL Server that stores information related to SQL Server Agent, Database Mail, and other components of SQL Server. Here are some quick notes on MSDB:

- SQL Server Agent: MSDB stores information about SQL Server Agent jobs, job steps, job schedules, and job history.
- Database Mail: MSDB stores configuration settings and metadata for the Database Mail feature, which allows you to send emails from within SQL Server.
- Backup and Restore: MSDB stores backup and restore history information, such as the backup or restore type, date, and time, as well as the backup or restore destination.
- Maintenance Plans: MSDB stores information about maintenance plans, including the schedule and the tasks that are included in each plan.
- Alerts: MSDB stores information about alerts that are configured in SQL Server, such as the alert name, severity level, and the email address or pager number to which the alert should be sent.

Physical properties:

- Filegroups: The MSDB database has two primary filegroups - PRIMARY and SECONDARY. The PRIMARY filegroup contains most of the system objects, while the SECONDARY filegroup contains user-defined objects like the Database Mail external files.
- Data files: MSDB has one primary data file and several secondary data files, which are used to store the database's data. The primary data file is named "MSDBData.mdf" and the secondary data files have the ".ndf" file extension. By default, these files are stored in the "C:\Program Files\Microsoft SQL Server\MSSQL<version>.<instance>\MSSQL\DATA" folder.

- Log files: MSDB has one transaction log file, which is used to record all transactions that occur in the database. The log file is named "MSDBLog.ldf" and is stored in the same folder as the data files.
- Database size: The size of the MSDB database depends on the number of jobs, schedules, and maintenance plans configured on the SQL Server instance. It is recommended to periodically clean up old job history and backup/restore history data to prevent the database from growing too large.

The restrictions on actions performed on the MSDB database in Microsoft SQL Server can be summarized as follows:

Ownership is set to the "sa" login and cannot be changed.

1. The MSDB database cannot be encrypted using Transparent Data Encryption (TDE).
2. Modifying the system tables in MSDB is not recommended.
3. Restoring backups of MSDB is generally not recommended.
4. Using the WITH MOVE option during restore of MSDB is not recommended.

Following these restrictions can help ensure the stability and security of the MSDB database and SQL Server. It is important to use supported methods for managing and maintaining the database.

Restoring the MSDB database from a different server in Microsoft SQL Server can be a complex process, and it is generally not recommended unless it is necessary to recover critical data. Here are some steps to restore MSDB from a different server:

1. Back up the MSDB database from the source server.
2. Copy the backup file to the destination server.
3. On the destination server, stop the SQL Server Agent service and any other services that depend on the MSDB database.
4. Restore the MSDB database backup file to the destination server using the WITH MOVE option to specify the destination file paths.
5. After the restore completes, restart the SQL Server Agent service and any other services that were stopped.

## Model db

A model database in Microsoft SQL Server (MSSQL) is a system database that serves as a template for creating new databases. It is used to define the physical properties and restrictions for the user-created databases. Here are some key points about the model system database in MSSQL:

1. Purpose: The model database is primarily used to provide a template for creating new databases. When a new database is created, it inherits the physical properties, objects, and settings defined in the model database.
2. Model db is also needed to create tempdb every time SQL server is started.
3. Physical Properties: The model database defines the initial physical properties for new databases. This includes settings such as file locations, file size and growth settings, collation settings, and recovery model.

4. Object Definition: Objects, such as tables, views, stored procedures, and other database schema objects, can be created in the model database. These objects are inherited by new databases created based on the model database.
5. Restrictions on Actions: There are certain restrictions and considerations to keep in mind when working with the model database in MSSQL. Some important points include:
    - Modifying the model database can impact all future databases created from it, so changes should be made with caution.
    - The system objects and system-defined schemas in the model database cannot be modified directly. Altering these objects can have unintended consequences on new databases.
    - Permissions and security settings defined in the model database are not inherited by new databases. Each new database has its own set of permissions that must be managed separately.
6. Customization: It is possible to customize the model database to suit specific requirements. This can include adding custom schema objects, setting specific database options, or configuring default settings.

Remember, making changes to the model database can have a wide-ranging impact on future databases. It is advisable to thoroughly plan and test any modifications before implementing them in the model database.

# Types of SQL Statements

## Data Definition Language (DDL) commands:

- CREATE: Used to create tables, views, indexes, and other database objects. Example: **CREATE TABLE Employees (ID int, Name varchar(255), Age int)**
- ALTER: Used to modify the structure of existing database objects. Example: **ALTER TABLE Employees ADD Gender varchar(10)**
- DROP: Used to remove tables, views, indexes, and other database objects. Example: **DROP TABLE Employees**
- Truncate Table

## Data Manipulation Language (DML) commands:

- SELECT: Used to retrieve data from one or more tables. Example: **SELECT * FROM Employees WHERE Age > 30**
- INSERT: Used to insert data into a table. Example: **INSERT INTO Employees (ID, Name, Age) VALUES (1, 'John', 35)**
- UPDATE: Used to modify data in a table. Example: **UPDATE Employees SET Age = 40 WHERE ID = 1**
- DELETE: Used to delete data from a table. Example: **DELETE FROM Employees WHERE Age > 60**

## Data Control Language (DCL) commands:

- GRANT: Used to grant permissions to users or roles. Example: **GRANT SELECT ON Employees TO User1**
- REVOKE: Used to revoke permissions from users or roles. Example: **REVOKE SELECT ON Employees FROM User1**

## Transaction Control Language (TCL) commands:

- COMMIT: Used to save the changes made in a transaction to the database. Example: **COMMIT**
- ROLLBACK: Used to undo the changes made in a transaction and restore the database to its previous state. Example: **ROLLBACK**
- A savepoint is a marker within a transaction that can be used to roll back a portion of the transaction, rather than the entire transaction. Here's an example:

```
BEGIN TRANSACTION;
-- Make some changes
UPDATE Customers SET Name = 'Alice' WHERE ID = 1;
-- Create a savepoint
SAVE TRANSACTION mySavepoint;
-- Make some more changes
UPDATE Customers SET Name = 'Bob' WHERE ID = 2;
-- Oops, something went wrong
ROLLBACK TRANSACTION mySavepoint;
-- The first update will still be committed
COMMIT TRANSACTION;
```

# CREATE Table

```
CREATE TABLE [schema_name].[table_name] (
   [column1] [data_type] [column1_constraint],
   [column2] [data_type] [column2_constraint],
   [column3] [data_type] [column3_constraint],
   ...
   [table_constraint1],
   [table_constraint2],
   ...
) ON [filegroup_name]
TEXTIMAGE_ON [filegroup_name]
[PARTITION BY RANGE ([column1])]
(
   [PARTITION partition_name VALUES LESS THAN (value)],
   [PARTITION partition_name VALUES LESS THAN (value)],
   ...
)
WITH ([table_option1] = value, [table_option2] = value, ...)
```

- **[schema_name]** is the name of the schema in which the table will be created. This is optional and can be omitted if you want to create the table in the default schema.
- **[table_name]** is the name of the table that you want to create.
- **[column1]**, **[column2]**, **[column3]**, and so on, are the names of the columns that you want to create.

- **[data_type]** is the data type of the column. You can use any valid data type, such as int, varchar, datetime, and so on.
- **[column1_constraint]**, **[column2_constraint]**, **[column3_constraint]**, and so on, are optional constraints that you can add to the columns. Examples of column constraints include PRIMARY KEY, UNIQUE, NOT NULL, and DEFAULT.
- **[table_constraint1]**, **[table_constraint2]**, and so on, are optional table-level constraints that you can add to the table. Examples of table constraints include PRIMARY KEY, FOREIGN KEY, CHECK, and UNIQUE.
- **[filegroup_name]** is the name of the filegroup in which the table will be stored.
- **TEXTIMAGE_ON [filegroup_name]** specifies the filegroup for storing large object data, such as text and image data.
- **[PARTITION BY RANGE ([column1])]** specifies that the table is partitioned by range on the specified column. You can use other partitioning methods, such as HASH, LIST, and COLUMNSTORE.
- **[partition_name VALUES LESS THAN (value)]** specifies the partition boundaries and the values that are stored in each partition.
- **WITH ([table_option1] = value, [table_option2] = value, ...)** specifies additional options for the table, such as FILESTREAM_ON, MEMORY_OPTIMIZED, and DATA_COMPRESSION.

## Constraints

### Primary Key Constraint:

CREATE TABLE Employees ( EmployeeID INT PRIMARY KEY, FirstName VARCHAR(50), LastName VARCHAR(50) );

Explanation: The primary key constraint is applied to the **EmployeeID** column, ensuring that each value is unique and not NULL.

### Foreign Key Constraint:

CREATE TABLE Orders ( OrderID INT PRIMARY KEY, CustomerID INT, OrderDate DATE, FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID) );

Explanation: The foreign key constraint establishes a relationship between the **Orders** and **Customers** tables based on the **CustomerID** column, ensuring data consistency.

### Unique Constraint:

CREATE TABLE Products ( ProductID INT PRIMARY KEY, ProductName VARCHAR(50), CategoryID INT, CONSTRAINT UQ_ProductName UNIQUE (ProductName) );

Explanation: The unique constraint ensures that the **ProductName** column contains unique values within the **Products** table.

### Check Constraint:

CREATE TABLE Students ( StudentID INT PRIMARY KEY, FirstName VARCHAR(50), LastName VARCHAR(50), Age INT CHECK (Age >= 18) );

Explanation: The check constraint ensures that the **Age** column in the **Students** table only accepts values that are 18 or greater.

## Default Constraint:

CREATE TABLE Orders ( OrderID INT PRIMARY KEY, OrderDate DATE DEFAULT GETDATE(), TotalAmount DECIMAL(10, 2) );

Explanation: The default constraint assigns the current date to the **OrderDate** column if no value is specified during the insertion of a new row.

## Identity Constraint:

CREATE TABLE Customers ( CustomerID INT PRIMARY KEY IDENTITY(1, 1), FirstName VARCHAR(50), LastName VARCHAR(50) );

Explanation: The identity constraint automatically generates unique values for the **CustomerID** column, incrementing by 1 for each new row.

To manually enter data in identity constraint:

To manually insert data into an identity column, you can use the **SET IDENTITY_INSERT** statement. Here's how you can do it:

1. Enable the **IDENTITY_INSERT** option for the table:

   SET IDENTITY_INSERT TableName ON/OFF;

Important Functions:

- **@@IDENTITY** returns the last identity value generated in the current session, irrespective of the scope or table.
- **SCOPE_IDENTITY()** returns the last identity value generated within the current scope (session, stored procedure, trigger, or function).
- **IDENT_CURRENT('TableName')** returns the last identity value generated for a specific table, regardless of the current session or scope.

Example:

```
SELECT @@IDENTITY AS LastGlobalIdentity;
SELECT SCOPE_IDENTITY() AS LastIdentityWithinScope;
SELECT IDENT_CURRENT('TableName') AS LastIdentityForTable;
```

## Checksum Constraint:

CREATE TABLE Products ( ProductID INT PRIMARY KEY, ProductName VARCHAR(50), Quantity INT, ChecksumValue AS BINARY_CHECKSUM(ProductName, Quantity) );

Explanation: The checksum constraint computes a checksum value based on the **ProductName** and **Quantity** columns, ensuring data integrity within the **Products** table.

## XML Constraint:

CREATE TABLE Orders ( OrderID INT PRIMARY KEY, OrderXML XML(SCHEMA Collection.OrdersSchema) );

Explanation: The XML constraint enforces that the **OrderXML** column adheres to the specified XML schema defined in the **Collection.OrdersSchema**.

Note:

The **WITH NOCHECK** option is used in the context of creating or re-enabling a foreign key constraint in SQL Server. When a foreign key constraint is created or re-enabled with the **WITH NOCHECK** option, SQL Server does not validate the existing data against the constraint.

Example:

ALTER TABLE [ChildTable] WITH NOCHECK ADD CONSTRAINT [FK_ConstraintName] FOREIGN KEY ([ChildColumn]) REFERENCES [ParentTable] ([ParentColumn]);

## Handling Operations during Alter

In Microsoft SQL Server (MSSQL), when altering a table to add or modify a foreign key constraint, you can specify the **ON UPDATE** and **ON DELETE** options to define the desired behavior when the referenced data is updated or deleted. Here's how you can use these options during an alter table operations:

### ON UPDATE:

Specifies the action to be taken when the referenced data is updated.

Available options:

- **NO ACTION** (default): The update is not allowed if it would violate the foreign key relationship.
- **CASCADE**: The update is propagated to the referencing table, modifying the corresponding foreign key values.
- **SET NULL**: The foreign key values in the referencing table are set to NULL when the referenced data is updated.
- **SET DEFAULT**: The foreign key values in the referencing table are set to their default values when the referenced data is updated.
- **SET {expression}**: The foreign key values in the referencing table are set to the specified expression when the referenced data is updated.

### ON DELETE:

Specifies the action to be taken when the referenced data is deleted.

- **NO ACTION** (default): The deletion is not allowed if it would violate the foreign key relationship.
- **CASCADE**: The deletion is propagated to the referencing table, deleting the corresponding rows.
- **SET NULL**: The foreign key values in the referencing table are set to NULL when the referenced data is deleted.
- **SET DEFAULT**: The foreign key values in the referencing table are set to their default values when the referenced data is deleted.
- **SET {expression}**: The foreign key values in the referencing table are set to the specified expression when the referenced data is deleted.

Here's an example of using the **ON UPDATE** and **ON DELETE** options during an alter table operation:

ALTER TABLE ChildTable ADD CONSTRAINT FK_ForeignKey FOREIGN KEY (ForeignKeyColumn) REFERENCES ParentTable (PrimaryKeyColumn) ON UPDATE CASCADE ON DELETE SET NULL;

## DataTypes in MSSQL

| Data type | Range |
|---|---|
| Numeric | **tinyint** (0 to 255) <br> **smallint** (-32,768 to 32,767) <br> **int** (-2,147,483,648 to 2,147,483,647) <br> **bigint** (-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807) <br> **decimal** (-10^38 +1 to 10^38 -1) <br> **numeric** (-10^38 +1 to 10^38 -1) <br> **float** (-1.79E+308 to 1.79E+308) <br> **real** (-3.40E+38 to 3.40E+38) |
| Character | **char** (0 to 8,000) <br> **varchar** (0 to 8,000) <br> **text** (0 to 2^31-1) |
| Unicode character | **nchar** (0 to 4,000) <br> **nvarchar** (0 to 4,000) <br> **ntext** (0 to 2^30-1) |
| Date and time | **date** (0001-01-01 to 9999-12-31) <br> **time** (00:00:00.0000000 to 23:59:59.9999999) <br> **datetime** (1753-01-01 to 9999-12-31 23:59:59.997) <br> **datetime2** (0001-01-01 to 9999-12-31 23:59:59.9999999) <br> **smalldatetime** (1900-01-01 to 2079-06-06 23:59:59) <br> **datetimeoffset** (-14:00 to +14:00) |
| Binary | **binary** (0 to 8,000) <br> **varbinary** (0 to 8,000) <br> **image** (0 to 2^31-1) |
| Other | **bit** (0 or 1) <br> **uniqueidentifier** <br> **sql_variant** <br> **xml** (0 to 2^31-1) |

### Strings

The main differences between varchar and nvarchar are:

varchar is used for non-Unicode character data, while nvarchar is used for Unicode character data.The storage size of varchar is based on the number of characters, while the storage size of nvarchar is based on the number of bytes.

nvarchar takes up twice as much space as varchar because it stores characters using two bytes per character.

In Microsoft SQL Server, the "varchar(max)" data type allows you to store variable-length character data up to a maximum size of 2^31-1 bytes (or 2,147,483,647 bytes). This is the largest amount of space that a single variable can occupy in SQL Server.

It's important to note that even though you can declare a "varchar(max)" column to hold such a large amount of data, the actual amount of data stored in the column will affect performance. If the data exceeds 8,000 bytes, SQL Server will store it in a special storage area called the "LOB" (Large Object) storage, which can have some performance implications.

If you're working with very large text data, you might also consider using the "nvarchar(max)" data type, which allows you to store Unicode character data with the same maximum size limit.

In summary, varchar is a more space-efficient option for non-Unicode data, while nvarchar is necessary for storing Unicode data.

### *Concatenate String*

1. CONCAT Function: The CONCAT function allows you to join multiple strings together. It automatically handles NULL values and converts non-string values to strings. For example:

   SELECT CONCAT('Hello', ' ', 'World') AS JoinedString;

2. Plus (+) Operator: You can also use the plus (+) operator to concatenate strings. For example:

   SELECT 'Hello ' + 'World' AS JoinedString;

3. COALESCE and ISNULL Functions: The COALESCE and ISNULL functions can be used to handle NULL values when joining strings. COALESCE returns the first non-null expression from the provided list, while ISNULL replaces a NULL value with a specified replacement value. For example:

   SELECT COALESCE(Column1, '') + COALESCE(Column2, '') AS JoinedString FROM TableName;
   SELECT ISNULL(Column1, '') + ISNULL(Column2, '') AS JoinedString FROM TableName;

## Decimal and Numeric

1. Decimal:
   - Used for precise numeric values with fixed precision and scale.
   - Syntax: **DECIMAL(p, s)** where **p** is the total number of digits (precision) and **s** is the number of digits after the decimal point (scale).
   - Example: **DECLARE @myDecimal DECIMAL(10, 2) = 1234.56;**
2. Numeric:
   - Similar to decimal, used for fixed-precision and scale numeric values.
   - Syntax: **NUMERIC(p, s)** with the same parameters as decimal.
   - Example: **DECLARE @myNumeric NUMERIC(8, 4) = 12.3456;**

Both data types provide precise control over decimal values. The choice between them is often based on personal preference or specific requirements.

## Money and Smallmoney

1. Money:
   - The **money** data type is used to store monetary values with a fixed precision.
   - It represents currency values ranging from -922,337,203,685,477.5808 to 922,337,203,685,477.5807.
   - The storage size is 8 bytes.
   - Example: **DECLARE @myMoney MONEY = 1234.56;**
2. Smallmoney:
   - The **smallmoney** data type is used to store smaller monetary values with a fixed precision.
   - It represents currency values ranging from -214,748.3648 to 214,748.3647.
   - The storage size is 4 bytes.
   - Example: **DECLARE @mySmallMoney SMALLMONEY = 12.34;**

Both data types are used to handle monetary values, with **money** having a larger range than **smallmoney**. The choice between them depends on the specific requirements and the range of values you need to work with.

## Float and Real

1. Float:
    - The **float** data type is used to store approximate numeric values with a floating decimal point.
    - It is typically used for scientific, engineering, or large decimal values.
    - The storage size is 8 bytes.
    - Example: **DECLARE @myFloat FLOAT = 3.14159;**

2. Real:
    - The **real** data type is used to store approximate numeric values with a floating decimal point, similar to **float**.
    - It is a smaller version of the **float** data type.
    - The storage size is 4 bytes.
    - Example: **DECLARE @myReal REAL = 2.71828;**

Both **float** and **real** are used for approximate numeric values, but **float** has a larger storage size and a higher precision compared to **real**. The choice between them depends on the precision requirements and the range of values you need to work with.

## Date

1. DATE: Stores dates without time. Format: 'YYYY-MM-DD'. Example: '2023-05-31'.
2. TIME: Stores time without dates. Format: 'HH:MI:SS[.FFFFFF]'. Example: '12:30:45.1234567'.
3. DATETIME: Stores both date and time. Format: 'YYYY-MM-DD HH:MI:SS[.FFFFFF]'. Example: '2023-05-31 12:30:45.1234567'.
4. SMALLDATETIME: Stores date and time with reduced range and precision. Format: 'YYYY-MM-DD HH:MI:SS'. Example: '2023-05-31 12:30:00'.
5. DATETIME2: Stores date and time with extended range and precision. Format: 'YYYY-MM-DD HH:MI:SS[.FFFFFF]'. Example: '2023-05-31 12:30:45.1234567'.

Useful date and time functions in SQL Server:

- GETDATE(): Returns the current date and time.
  Example: SELECT GETDATE()
- SYSDATETIME(): Returns the current system date and time with higher fractional precision.
  Example: SELECT SYSDATETIME()
- DATEADD(): Adds or subtracts a specified time interval from a date.
  Example: SELECT DATEADD(DAY, 7, '2023-05-31')
- DATEDIFF(): Calculates the difference between two dates in a specified time interval.
  Example: SELECT DATEDIFF(MONTH, '2022-01-01', '2023-05-31')
- DATEPART(): Extracts a specific part (year, month, day, etc.) from a date or time.
  Example: SELECT DATEPART(YEAR, '2023-05-31')
- CONVERT(): Converts a date or time value from one data type to another.
  Example: SELECT CONVERT(VARCHAR, GETDATE(), 106)
- FORMAT(): Formats a date or time value according to a specified format string.
  Example: SELECT FORMAT(GETDATE(), 'dd/MM/yyyy')

### Conversions between datatypes

In Microsoft SQL Server, you can perform data type conversions using various functions and operators. Here are some commonly used methods for data type conversions in MSSQL:

1. Implicit Conversion: SQL Server automatically performs implicit conversions when it can safely convert one data type to another without data loss. For example, converting an integer to a decimal. Implicit conversions are handled by the database engine without requiring any explicit conversion functions.
2. Explicit Conversion: You can explicitly convert data types using the CAST or CONVERT functions. These functions allow you to convert one data type to another explicitly. For example:
    - CAST function: **CAST(expression AS datatype)**
    - CONVERT function: **CONVERT(datatype, expression [,style])**
3. String Conversions: SQL Server provides functions like CAST, CONVERT, and PARSE for converting string values to other data types. For example:
    - **CAST('123' AS INT)** converts the string '123' to an integer.
    - **CONVERT(INT, '123')** converts the string '123' to an integer.
    - **PARSE('123' AS INT)** also converts the string '123' to an integer.
4. Type Precedence: SQL Server has a concept called "type precedence" that determines which data type takes precedence when performing operations involving different data types. The data type with higher precedence is implicitly converted to the data type with lower precedence. You can refer to Microsoft's documentation for the complete list of type precedence.
5. Handling NULL values: When converting NULL values, SQL Server generally returns NULL unless you use the ISNULL or COALESCE functions to handle them explicitly.

It's important to note that data type conversions may result in loss of precision or unexpected behavior, so it's essential to ensure the data types being converted are compatible and handle potential issues appropriately

## Temporary Tables

- Temporary tables are a type of table that exist only for the duration of a user's session and are automatically dropped when the session ends.
- Temporary tables can be used to store intermediate results during complex queries or procedures, or to share data between different parts of a script.
- There are two types of temporary tables in MSSQL: local temporary tables and global temporary tables. Local temporary tables are only visible to the current user's session, while global temporary tables are visible to all sessions.
- Temporary tables can have constraints and indexes just like regular tables.
- Temporary tables can have performance implications, especially if they are used excessively or if they are not indexed properly.
- Temporary tables are stored in the tempdb database, which is a system database that is used for storing temporary objects.

- Temporary tables can be dropped explicitly using the **DROP TABLE** statement, or they will be dropped automatically when the user's session ends.

- **Table variables** (DECLARE @t TABLE) are visible only to the connection that creates it, and are deleted when the batch or stored procedure ends.
- **Local temporary tables** (CREATE TABLE #t) are visible only to the connection that creates it, and are deleted when the connection is closed. Can also use select * into to create
- **Global temporary tables** (CREATE TABLE ##t) are visible to everyone, and are deleted when all connections that have referenced them have closed.
- **Tempdb permanent tables** (USE tempdb CREATE TABLE t) are visible to everyone, and are deleted when the server is restarted.

## Derived Tables

Derived tables in Microsoft SQL Server (MSSQL) are temporary result sets that are created within a query and used for further analysis or manipulation. Here are some concise notes on derived tables and their use:

1. Definition: A derived table, also known as an inline view or subquery, is a virtual table created within the scope of a query. It is not stored as a separate object in the database but exists only for the duration of the query execution.

2. Purpose: Derived tables are used to simplify complex queries by breaking them into smaller, more manageable parts. They allow you to perform intermediate calculations, filtering, and joining of data within a single query.

3. Syntax: To create a derived table, enclose the subquery within parentheses and assign it an alias. For example:

   SELECT * FROM (SELECT col1, col2 FROM table1) AS derived_table WHERE derived_table.col1 = 'value';

4. Usage scenarios:

- Filtering: You can use a derived table to filter records based on specific conditions before joining them with other tables in the main query.
- Aggregation: Derived tables can be employed to aggregate data using functions like SUM, COUNT, AVG, etc., and then utilize the aggregated result in the main query.
- Subqueries: When a subquery needs to be referenced multiple times within a query, using a derived table can improve performance by computing the subquery only once.

5. Benefits:

- Simplified query structure: Derived tables help organize complex queries by dividing them into logical steps.
- Improved readability: By giving a subquery a meaningful alias, the derived table makes the query more understandable.
- Optimized performance: In certain scenarios, using derived tables can enhance query performance by allowing the database optimizer to optimize and cache results.

CTE

In Microsoft SQL Server (MSSQL), a Common Table Expression (CTE) is a temporary named result set that you can reference within a SELECT, INSERT, UPDATE, or DELETE statement. CTEs provide a way

to break complex queries into smaller, more manageable parts, improving readability and maintainability of SQL code.

A CTE is defined using the WITH keyword, followed by a unique name for the CTE and a query that defines the result set. The CTE can then be referenced and treated like a table within the subsequent statement.

Example:

```
WITH SalesSummary AS (
  SELECT
    ProductCategory,
    SUM(Quantity) AS TotalQuantity,
    SUM(Revenue) AS TotalRevenue
  FROM
    Sales
  GROUP BY
    ProductCategory
)
SELECT
  ProductCategory,
  TotalQuantity,
  TotalRevenue
FROM
  SalesSummary
```

In the subsequent SELECT statement, the CTE "SalesSummary" is referenced and treated as a table. The columns from the CTE can be selected and used in the outer query.
CTEs are particularly useful when you need to reuse the same result set multiple times within a query or when you want to break down a complex query into smaller, more understandable parts. They can help improve query performance and simplify complex SQL logic.

## Synonym

In Microsoft SQL Server (MSSQL), a synonym is an alternative name or alias for an existing database object, such as a table, view, stored procedure, or function. Synonyms provide a way to simplify the naming and referencing of objects, especially when working with multiple databases or when you want to abstract the underlying object names.

```
CREATE SYNONYM MyTableAlias FOR MyDatabase.dbo.MyTable
```

By utilizing synonyms, you can achieve the following benefits:

- Simplify code: Synonyms can provide shorter or more intuitive names for objects, making the code more readable and easier to understand.
- Database abstraction: Synonyms can be used to abstract the underlying database structure, allowing you to switch between databases without modifying the code that references the objects.
- Security: Synonyms can be used to provide controlled access to objects, allowing users to interact with the synonym while restricting direct access to the original object.

# View

view is a virtual table that is based on the result set of a SELECT statement. Views do not store data themselves, but rather present data from one or more underlying tables in a specific way.

Views can be used to simplify complex queries, provide an additional level of security, or to hide sensitive or irrelevant data.

Views can be created using the CREATE VIEW statement, which includes a SELECT statement that defines the result set of the view.

Order by is invalid unless top/offset is defined.

CREATE VIEW vw_customer_orders AS SELECT c.customer_id, c.customer_name, o.order_id, o.order_date

FROM customers c INNER JOIN orders o ON c.customer_id = o.customer_id

There are several types of views in MSSQL, including:

- Simple views: These views are based on a single table or a SELECT statement that does not contain GROUP BY, HAVING, or aggregate functions.
- Complex views: These views are based on a SELECT statement that contains GROUP BY, HAVING, or aggregate functions.
- Indexed views: These views are created with a unique clustered index that is used to improve performance of frequently used queries.
- Partitioned views: These views are used to partition large tables into smaller subsets, which can improve performance and manageability.

Views can include various options, such as:

**WITH CHECK OPTION**, which prevents updates that do not meet the view's criteria,
**SCHEMABINDING**, which binds the view to the schema of the underlying tables,

**WITH ENCRYPTION,** which hides the select statement used to create view from other users.

## DML Commands on Views

It is possible to use DML (Data Manipulation Language) commands on views in MSSQL. However, the way in which you can use DML commands depends on the type of view you are using.

- Simple views: If the view is based on a single table or a SELECT statement that does not contain GROUP BY, HAVING, or aggregate functions, and has one table, you can use DML commands on the view as if it were a regular table. For example:

UPDATE vw_customers SET customer_name = 'John Smith' WHERE customer_id = 1

- Complex views: If the view is based on a SELECT statement that contains GROUP BY, HAVING, or aggregate functions, you cannot use DML commands directly on the view. However, you can use an **INSTEAD OF** trigger to intercept the DML command and perform the necessary actions on the underlying tables. For example:
- Indexed views: If the view is an indexed view, you can use DML commands on the view as if it were a regular table. However, you must ensure that the DML command meets certain

criteria to ensure that the view's index remains valid. For example, an UPDATE statement that affects the columns used in the view's index might fail. In this case, you can either update the underlying tables directly or modify the view's index definition to include the updated columns.

- Partitioned views: If the view is a partitioned view, you can use DML commands on the view as if it were a regular table. However, you must ensure that the DML command only affects the appropriate partition(s) of the underlying tables. For example, an UPDATE statement that affects all partitions might be slow and inefficient. In this case, you can specify a partitioning column in the WHERE clause of the UPDATE statement to limit the update to a specific partition.

Note that when you use DML commands on a view, the changes are made to the underlying tables, not to the view itself. The view only provides a way to present the data in a particular way.

### Materialized View

In MSSQL, there is no direct equivalent to a materialized view, but you can achieve similar functionality using indexed views. An indexed view is a view that has been persisted to disk and has an index associated with it, which allows for fast retrieval of data. When you query an indexed view, SQL Server uses the index to retrieve the data instead of executing the underlying query.

The benefit of using an indexed view is that it can improve query performance by reducing the number of reads and CPU usage required to retrieve data. However, creating an indexed view can also have an impact on the performance of data modification operations, as the index must be maintained whenever the underlying tables are updated.

To create an indexed view in MSSQL, you can use the CREATE VIEW statement with the WITH SCHEMABINDING option, which ensures that the underlying tables cannot be modified while the view is being created or modified. You can then create an index on the view using the CREATE UNIQUE CLUSTERED INDEX or CREATE NONCLUSTERED INDEX statement.

## Partitioned Tables

In MSSQL, a partitioned table is a table that has been split into multiple partitions, which are essentially smaller, more manageable pieces of data. Each partition can be stored on a separate filegroup or file system, allowing for more efficient data management and storage.

Partitioning a table can improve query performance by allowing SQL Server to read only the necessary partitions, rather than scanning the entire table. Partitioning can also make it easier to manage large tables and perform maintenance tasks, such as backups and index maintenance.

There are several types of partitioning available in MSSQL, including:

- Range partitioning: partitions data based on a range of values in a specified column.
- Hash partitioning: partitions data based on a hash function applied to a specified column.
- List partitioning: partitions data based on a list of values in a specified column.

To create a partitioned table in MSSQL, you can use the CREATE TABLE statement with the PARTITION BY clause to specify the partitioning scheme. You can also use the ALTER TABLE statement to add or remove partitions as needed.

Here's an example of creating a partitioned table in MSSQL using range partitioning:

```
CREATE TABLE sales (
   sale_id INT NOT NULL,
   sale_date DATE NOT NULL,
   region VARCHAR(50) NOT NULL,
   sales_amount DECIMAL(10,2) NOT NULL,
   CONSTRAINT pk_sales PRIMARY KEY (sale_id, sale_date)
) ON psales (sale_date)
PARTITION BY RANGE (sale_date)
(
   PARTITION p2019q1 VALUES LESS THAN ('2019-04-01'),
   PARTITION p2019q2 VALUES LESS THAN ('2019-07-01'),
   PARTITION p2019q3 VALUES LESS THAN ('2019-10-01'),
   PARTITION p2019q4 VALUES LESS THAN ('2020-01-01')
)
```

In this example, we're creating a partitioned table called "sales" and partitioning it by range on the "sale_date" column. We're also specifying four partitions based on the quarter of the year. The table is being created on a filegroup called "psales", and the primary key is defined on the "sale_id" and "sale_date" columns.

## SELECT

The SELECT statement in MSSQL is used to retrieve data from one or more tables in a database. It consists of the SELECT keyword followed by a list of columns or expressions to retrieve, and the FROM keyword followed by the table(s) to retrieve data from. Optional clauses can be used to filter and sort the data, such as WHERE, GROUP BY, HAVING, and ORDER BY.

Some important points to keep in mind when using the SELECT statement in MSSQL include:

- The column or expression list can include wildcards, functions, and aliases.
- Multiple tables can be joined using JOIN clauses, which specify how the tables are related.
- The WHERE clause is used to filter data based on specific conditions.
- The GROUP BY clause is used to group data based on one or more columns, and the HAVING clause is used to filter groups based on specific conditions.
- The ORDER BY clause is used to sort data based on one or more columns.
-

Here's an example of a basic SELECT statement in MSSQL:

```
SELECT column1, column2, column3 FROM table1 WHERE column1 = 'value';
```

Tip: SELECT TOP(n) PERCENT * from tbl_nm

--will return top 5% rows of a table.

## Operators, Expressions and Conditions

- Operators in MSSQL are symbols or keywords that are used to perform operations on one or more expressions. Some examples of operators include arithmetic operators (+, -, *, /), comparison operators (=, <>, >, <, >=, <=, !=), and logical operators (AND, OR, NOT).

- Expressions in MSSQL are combinations of operators, constants, functions, and column or table names that evaluate to a single value. They can be used in SELECT, WHERE, HAVING, and ORDER BY clauses.

- Conditions in MSSQL are used to filter data based on specific criteria. They are composed of one or more expressions, combined with comparison or logical operators. The WHERE clause is used to apply conditions to a SELECT statement, while the HAVING clause is used to apply conditions to a GROUP BY statement.
- Some important points to keep in mind when using operators, expressions, and conditions in MSSQL include:
    - Parentheses can be used to control the order of evaluation in expressions.
    - NULL values can be handled using the IS NULL and IS NOT NULL operators.
    - LIKE and IN operators can be used to perform pattern matching and list comparisons, respectively.
    - The EXISTS operator can be used to check for the existence of a subquery.
    - The BETWEEN operator can be used to specify a range of values.
- Here are some examples of using operators, expressions, and conditions in MSSQL:

SELECT column1 + column2 AS sum FROM table1 WHERE column3 > 10;

SELECT column1, COUNT(*) FROM table1 GROUP BY column1 HAVING COUNT(*) > 1;

SELECT column1, column2 FROM table1 WHERE column1 LIKE 'A%'; SELECT column1, column2 FROM table1 WHERE column3 IN (1, 2, 3);

SELECT column1, column2 FROM table1 WHERE EXISTS (SELECT * FROM table2 WHERE table2.column3 = table1.column3);

SELECT column1, column2 FROM table1 WHERE column3 BETWEEN 1 AND 10;

## Clause

### WHERE:

Filter data. Used in select, update, delete

### ORDER BY

Sort data in asc/desc. Can be on multiple cols. Will be consideredin order specified.

### GROUP BY

The GROUP BY clause in MSSQL is used to group rows of data based on one or more columns, and to apply aggregate functions to the grouped data.

Here are some important points to keep in mind when using GROUP BY in MSSQL:

- The GROUP BY clause must come after the WHERE clause and before the ORDER BY clause (if used).
- The SELECT statement can include both columns that are being grouped and columns that are not being grouped, but any columns that are not being grouped must be included in an aggregate function.
- Aggregate functions are used to perform calculations on the grouped data, such as SUM, AVG, COUNT, MIN, and MAX.
- The result set of a GROUP BY query contains one row for each unique combination of values in the grouped columns.

### HAVING CLAUSE

The HAVING clause in MSSQL is used in conjunction with the GROUP BY clause to filter the results of a query based on an aggregate function.

Here are some important points to keep in mind when using HAVING in MSSQL:

- The HAVING clause is used to filter the results of a query based on an aggregate function, such as SUM, AVG, COUNT, MIN, or MAX.
- The HAVING clause must come after the GROUP BY clause and before the ORDER BY clause (if used).
- The HAVING clause is used to filter groups of rows, rather than individual rows.
- The HAVING clause can include any valid expression that can be evaluated to true or false, such as comparison operators, logical operators, and subqueries.

```
SELECT category, SUM(sales) as total_sales
FROM sales_data
WHERE year = 2022
GROUP BY category
HAVING SUM(sales) > 1000
ORDER BY total_sales DESC;
```

## JOIN

Sure, here's a summary of SQL joins with their types in MSSQL:

- SQL joins are used to combine data from two or more tables based on a related column between them.
- There are four types of joins in SQL:
    1. Inner join: returns only the matching rows from both tables based on the specified condition.
    2. Left join (or Left outer join): returns all the rows from the left table and matching rows from the right table based on the specified condition. If there is no matching row in the right table, the result will contain NULL values for the right table columns.
    3. Right join (or Right outer join): returns all the rows from the right table and matching rows from the left table based on the specified condition. If there is no matching row in the left table, the result will contain NULL values for the left table columns.
    4. Full join (or Full outer join): returns all the rows from both tables and matching rows based on the specified condition. If there is no matching row in either table, the result will contain NULL values for the columns from the table that has no match.
- The syntax for joining tables in SQL is:

  ```
  SELECT * FROM table1 JOIN table2 ON table1.column_name = table2.column_name;
  ```

- You can use aliases for table names to make the query more readable and easier to write.
- You can join more than two tables by adding additional JOIN clauses and ON conditions to the query.

# SUBQUERIES

- A subquery is a query that is nested inside another query.
- The subquery is executed first, and its result is used by the outer query.
- Subqueries can be used in the SELECT, FROM, WHERE, and HAVING clauses of a query.
- Subqueries can return a single value, a single row, or multiple rows.

Example:

SELECT column_name FROM table_name WHERE column_name = (SELECT column_name FROM another_table WHERE column_name = 'value');

SELECT column_name FROM (SELECT column_name FROM table_name WHERE column_name = 'value') AS alias_name;

SELECT column_name FROM table_name WHERE column_name IN (SELECT column_name FROM another_table WHERE column_name LIKE '%value%');

SELECT column_name FROM table_name WHERE EXISTS (SELECT column_name FROM another_table WHERE column_name = 'value');

SELECT column_name FROM table_name WHERE column_name = ANY (SELECT column_name FROM another_table WHERE column_name > 10);

SELECT column_name, COUNT() AS count FROM table_name GROUP BY column_name HAVING COUNT() > (SELECT AVG(count) FROM (SELECT COUNT(*) AS count FROM table_name GROUP BY column_name) AS subquery);

This example uses a subquery in the HAVING clause to compare the count of each group to the average count of all groups. It only returns the rows where the count is greater than the average count

## Correlated Subquery

A correlated subquery is a type of subquery in SQL where the inner subquery references a column from the outer query. The subquery depends on values from the outer query to perform its operation. The subquery is executed for each row processed by the outer query, and the results are used in the evaluation of the outer query.

SELECT

CustomerName,

(SELECT COUNT(*) FROM Orders WHERE Orders.CustomerID = Customers.CustomerID) AS OrderCount FROM Customers

# Combine Datasets

## UNION

In Microsoft SQL Server (MS SQL), a union is a set operation that allows you to combine the result sets of two or more SELECT statements into a single result set. The UNION operator removes duplicate rows from the combined result set by default. The columns of tables we choose to combine must be of same number and compatible datatypes(char/varchar;date/datetime,etc). Between compatible types, larger type is chosen.

The syntax for using UNION in MS SQL is as follows:

```
SELECT column1, column2, ...
FROM table1
WHERE conditions
UNION [ALL]
SELECT column1, column2, ...
FROM table2
WHERE conditions;
```

**ALL** (optional): When **ALL** is used with **UNION**, it retains all rows, including duplicates, from the combined result set. By default, duplicate rows are removed.

## INTERSECT

In Microsoft SQL Server (MS SQL), the INTERSECT operator is used to retrieve the common rows between two or more SELECT statements. It returns only the distinct rows that are present in all the result sets of the SELECT statements involved.

Syntax:

```
SELECT column1, column2, ... FROM table1 WHERE conditions
INTERSECT
SELECT column1, column2, ... FROM table2 WHERE conditions;
```

## EXCEPT

In Microsoft SQL Server Management Studio (SSMS), the EXCEPT operator is used to retrieve the distinct rows from the result set of the first SELECT statement that are not present in the result set of the second SELECT statement. It essentially subtracts the rows of the second SELECT statement from the first SELECT statement.

Syntax:

```
SELECT column1, column2, ... FROM table1 WHERE conditions
EXCEPT
SELECT column1, column2, ... FROM table2 WHERE conditions;
```

## CASE Statement

In Microsoft SQL Server (MS SQL), the CASE statement is a conditional expression that allows you to perform branching logic within a query. It evaluates a set of conditions and returns a result based on the first matching condition.

```
CASE
  WHEN condition1 THEN result1
  WHEN condition2 THEN result2
  ...
  ELSE result
END
```

The results of each case must be of same datatype. Else is optional.

### IsNull

In Microsoft SQL Server (MS SQL), the ISNULL function is used to replace NULL values with a specified alternative value. It checks if a given expression is NULL and, if so, returns the alternative value; otherwise, it returns the original expression.

Ex: SELECT product_name, ISNULL(unit_price, 0) AS price FROM products;

### COALESCE

In Microsoft SQL Server (MS SQL), the COALESCE function is used to return the first non-null expression from a list of expressions. It evaluates the expressions in the specified order and returns the value of the first expression that is not NULL.

Ex: SELECT product_name, COALESCE(unit_price, 0) AS price FROM products;

## Merge

In Microsoft SQL Server (MS SQL), the MERGE statement is used to perform insert, update, or delete operations on a target table based on the condition specified. It combines the capabilities of INSERT, UPDATE, and DELETE statements into a single statement, making it useful for synchronizing data between source and target tables.

The syntax for using MERGE in MS SQL is as follows:

```
MERGE target_table AS target
USING source_table AS source
ON target.condition_column = source.condition_column
WHEN MATCHED THEN
   UPDATE SET target.column1 = source.column1, target.column2 = source.column2, ...
WHEN NOT MATCHED BY TARGET THEN
   INSERT (column1, column2, ...)
   VALUES (source.column1, source.column2, ...)
WHEN NOT MATCHED BY SOURCE THEN
   DELETE;
```

Here's a breakdown of the components:

- **target_table**: This is the table on which you want to perform the merge operation.
- **target**: This is an alias for the target table.
- **source_table**: This is the table that provides the data to be merged into the target table.
- **source**: This is an alias for the source table.
- **ON**: This specifies the join condition between the target and source tables.

- **WHEN MATCHED THEN**: This defines the action to be taken when a match is found between the target and source tables. Typically, an update operation is performed to modify the target table columns.
- **UPDATE SET**: This specifies the columns and their new values to be updated in the target table based on the matched condition.
- **WHEN NOT MATCHED BY TARGET THEN**: This defines the action to be taken when there is a row in the source table that does not have a corresponding match in the target table. Usually, an insert operation is performed to add new rows to the target table.
- **INSERT**: This specifies the columns and their values to be inserted into the target table from the source table.
- **WHEN NOT MATCHED BY SOURCE THEN**: This defines the action to be taken when there is a row in the target table that does not have a corresponding match in the source table. Typically, a delete operation is performed to remove the unmatched rows from the target table.

The MERGE statement allows you to perform complex data synchronization and manipulation operations in a single statement, making it efficient and convenient for handling data integration scenarios.

Source and destination tables must have same number of columns. One row must not join to multiple rows of other table(workaround: group by)

## INSERT STATEMENT IN DETAIL

- The INSERT statement is used to insert new data into a table.
- The syntax of the INSERT statement is as follows:

  INSERT INTO table_name (column1, column2, column3, ...) VALUES (value1, value2, value3, ...);

  - **table_name**: the name of the table where you want to insert data.
  - **(column1, column2, column3, ...)**: the names of the columns in the table where you want to insert data. This is optional if you're inserting values into all columns in the table.
  - **(value1, value2, value3, ...)**: the values that you want to insert into the table.
- You can insert multiple rows of data in a single INSERT statement by separating each row with a comma. The syntax is as follows:

  INSERT INTO table_name (column1, column2, column3, ...) VALUES (value1, value2, value3, ...), (value1, value2, value3, ...), (value1, value2, value3, ...);
- You can also insert data into a table using a SELECT statement. The syntax is as follows:

  INSERT INTO table_name (column1, column2, column3, ...) SELECT column1, column2, column3, ... FROM another_table WHERE condition;

  - **another_table**: the name of the table from which you want to select data.

- **condition**: the condition that determines which rows to select from the other table.

- If you're inserting data into an identity column, you can use the **SCOPE_IDENTITY()** function to retrieve the identity value assigned to the last row inserted.

Example:

- INSERT INTO students (first_name, last_name, age) VALUES ('John', 'Doe', 25);
- INSERT INTO students VALUES ('John', 'Doe', 25);
- INSERT INTO students (first_name, last_name, age) VALUES ('John', 'Doe', 25), ('Jane', 'Doe', 24), ('Jim', 'Smith', 30);
- INSERT INTO students (first_name, last_name, age) SELECT first_name, last_name, age FROM applicants WHERE status = 'accepted';

### SELECT INTO

- The SELECT INTO statement is used to create a new table based on the result set of a SELECT statement.
- The new table is created with the same columns and data types as the result set of the SELECT statement.
- The syntax of the SELECT INTO statement is as follows:
  SELECT column1, column2, column3, ... INTO new_table FROM old_table WHERE condition;
  - **column1, column2, column3, ...**: the columns that you want to select from the old table.
  - **new_table**: the name of the new table that you want to create.
  - **old_table**: the name of the old table from which you want to select data.
  - **condition**: the condition that determines which rows to select from the old table. This is optional.
- The new table is created in the same database as the old table, with the same column names and data types as the selected columns in the old table.
- The SELECT INTO statement automatically creates the new table and populates it with the selected data from the old table.
- If the new table already exists, the SELECT INTO statement will fail with an error.
- You can use the SELECT INTO statement to create a backup copy of a table or to create a temporary table to perform further analysis.

## UPDATE STATEMENT

1. Updating a table with values from another table using a JOIN:

UPDATE table1 SET table1.column1 = table2.column1, table1.column2 = table2.column2 FROM table1 INNER JOIN table2 ON table1.join_column = table2.join_column WHERE table1.condition_column = 'condition';

   - This statement updates **table1** with values from **table2** where **table1.join_column** matches **table2.join_column**, and **table1.condition_column** is equal to a specific value.

2. Updating a table with a subquery that returns multiple values:

UPDATE table1 SET column1 = ( SELECT SUM(column2) FROM table2 WHERE table2.join_column = table1.join_column GROUP BY table2.join_column ) WHERE condition_column = 'condition';

- This statement updates **table1** with the sum of **column2** from **table2** where **table2.join_column** matches **table1.join_column**, and **table1.condition_column** is equal to a specific value.

3. Updating a table with a case statement:

UPDATE table1 SET column1 = CASE WHEN column2 = 'value1' THEN 'new_value1' WHEN column2 = 'value2' THEN 'new_value2' ELSE column1 END WHERE condition_column = 'condition';

- This statement updates **table1** and sets **column1** to a new value based on the value of **column2**. If **column2** is equal to **value1**, **column1** is set to **'new_value1'**, if **column2** is equal to **value2**, **column1** is set to **'new_value2'**, and if **column2** has any other value, **column1** is not changed. This is applied only where **condition_column** is equal to a specific value.

## DELETE statement

1. Deleting rows based on a subquery:
2. DELETE FROM table1 WHERE column1 IN ( SELECT column1 FROM table2 WHERE column2 = 'value' );

    - This statement deletes all rows from **table1** where **column1** matches any value returned by the subquery, which selects **column1** from **table2** where **column2** is equal to **'value'**.

3. Deleting rows based on a join:DELETE t1 FROM table1 t1 INNER JOIN table2 t2 ON t1.join_column = t2.join_column WHERE t1.condition_column = 'condition';

    - This statement deletes all rows from **table1** where **join_column** matches **join_column** in **table2**, and **condition_column** in **table1** is equal to **'condition'**.

4. Deleting duplicate rows based on a subquery:

    DELETE FROM table1 WHERE id NOT IN ( SELECT MIN(id) FROM table1 GROUP BY column1, column2, column3 );

    - This statement deletes all duplicate rows from **table1** except for the one with the minimum **id** value, based on the grouping of columns **column1**, **column2**, and **column3**.

5. the DELETE TOP statement is used to delete a specified number of rows from a table. Here's the syntax:

DELETE TOP (N) FROM table_name WHERE condition;

### DELETE TRUNCATE DROP

1. **DELETE** statement: Removes one or more rows from a table based on a condition.

    DELETE FROM table_name WHERE condition;

    This statement removes rows from **table_name** that meet the specified **condition**.

2. **TRUNCATE** statement: Removes all rows from a table without logging the individual row deletions. This statement is generally faster than the **DELETE** statement because it doesn't

log each deletion. Truncate removes data by deallocating data pages used to store the data and records the page deallocation in transaction log. To use truncate ALTER permission is needed atleast. This is a DDL command. Incase of truncate, value of identity is also reset(it will restart from init value)

TRUNCATE TABLE table_name;

This statement removes all rows from **table_name** without logging each individual deletion.

3. **DROP** statement: Removes an entire table or database from the server.

   DROP TABLE table_name;

   DROP DATABASE database_name;

   These statements remove **table_name** or **database_name** from the server.

It's important to note that **TRUNCATE** and **DROP** statements are non-reversible and permanently delete data from the database. Therefore, they should be used with caution and only after verifying that the data being deleted is no longer needed.

# Dynamic SQL

```
Declare @cmd as varchar(250);
Set @cmd = 'select * from tblNme'
Execute(@cmd);
```

## SQL Injection

QL injection is a type of security vulnerability that occurs when an attacker is able to manipulate the SQL queries executed by an application, typically a web application, by injecting malicious SQL code into user input fields.

In the case of Microsoft SQL Server (MSSQL), SQL injection can occur when user-supplied input is concatenated directly into SQL statements without proper validation or sanitization.

Ex:

```
Set @cmd = 'select * from tblNme where id = ';
Declare @param as varchar(5) = '2';
Exec(@cmd+@param)
```

To prevent SQL injection in MSSQL, consider the following best practices:

1. Parameterized queries: Instead of concatenating user input directly into SQL statements, use parameterized queries or prepared statements. Parameterized queries use placeholders for user input, and the values are supplied separately, ensuring that they are treated as data and not executable code.

```
Set @cmd =N 'select * from tblNme where id =  @PrdId';
ser @param = N'2';
execute sys.sp_executesql
        @statement = @cmd,
        @params = N'@PrdId int',
        @PrdId = @param;
```

2. Stored procedures: Use stored procedures to encapsulate and control the database logic. By doing so, you can minimize the risk of SQL injection as the input is treated as parameters rather than direct SQL statements.
3. Input validation and sanitization: Validate and sanitize user input on the server-side. Ensure that input adheres to the expected format, type, and length. Consider using whitelisting or regular expressions to filter out potentially malicious characters or patterns.
4. Principle of least privilege: Ensure that the database user account used by the application has the least privileges necessary to perform its tasks. Avoid using an account with administrative privileges for regular application operations.
5. Avoid dynamic SQL: Minimize the use of dynamic SQL where user input is concatenated into the query string. If dynamic SQL is necessary, ensure proper validation and sanitization of the input before constructing the dynamic query.
6. Error handling: Implement appropriate error handling and avoid displaying detailed error messages to users, as they can reveal sensitive information about the database structure or the SQL statements being executed.
7. Regular updates and patches: Keep your MSSQL server up to date with the latest patches and security updates. This helps protect against known vulnerabilities.

## Stored Procedure

- A stored procedure is a precompiled block of code that is stored in the database and can be called from within an application or directly from the database.
- Stored procedures can be created using T-SQL in the SQL Server Management Studio (SSMS) or any other database development tool that supports T-SQL.
- Stored procedures can accept input parameters, which are used to customize the behavior of the procedure at runtime. Input parameters can be specified using the **IN** keyword and can be of any valid SQL Server data type.
- Input parameters can have default values, which are used if no value is provided when the stored procedure is called.

Syntax:

```
CREATE PROCEDURE [schema_name.]procedure_name
  [ @parameter1 datatype [ VARYING ] [ = default ] [ OUT | OUTPUT | READONLY ]
  , @parameter2 datatype [ VARYING ] [ = default ] [ OUT | OUTPUT | READONLY ]
  , ...
  ]
AS
  BEGIN
    -- Procedure body
  END;
```

- Stored procedures can return output parameters, which are used to pass data back to the calling application. Output parameters can be specified using the OUT keyword and must be declared before the AS keyword.

```
CREATE PROCEDURE getCustomerCount
  @count INT OUT
AS
BEGIN
  SELECT @count = COUNT(*) FROM Customers
END
```

- Stored procedures can also return result sets, which are used to return data to the calling application. Result sets are created using the **SELECT** statement and are returned using the **RETURN** keyword.

```
CREATE PROCEDURE getCustomersByCountry
  @country NVARCHAR(50) = 'New York' –Default value
AS
BEGIN
  SELECT * FROM Customers WHERE Country = @country
  RETURN
END
```

- SP can return both output parameter and result set

- Stored procedures can be used to perform error handling and raise custom error messages. This is typically done using the **TRY...CATCH** block, which allows for graceful handling of errors and provides more information about the error that occurred.

- Transactions can be managed within stored procedures using the **BEGIN TRANSACTION**, **COMMIT TRANSACTION**, and **ROLLBACK TRANSACTION** statements. This allows for multiple SQL statements to be executed as a single transaction, ensuring that all changes are made together or none at all.

```
CREATE PROCEDURE placeOrder
  @customerId INT,
  @productId INT,
  @quantity INT
AS
BEGIN
  BEGIN TRY
    BEGIN TRANSACTION
    INSERT INTO Orders (CustomerId, ProductId, Quantity) VALUES (@customerId,
@productId, @quantity)
    UPDATE Products SET UnitsInStock = UnitsInStock - @quantity WHERE ProductId =
@productId
    COMMIT TRANSACTION
  END TRY
  BEGIN CATCH
    ROLLBACK TRANSACTION
    DECLARE @errorMessage NVARCHAR(1000) = ERROR_MESSAGE()
```

RAISERROR(@errorMessage, 16,'message')

## VARIABLES

- Variables in MSSQL are used to store temporary data within a script or batch of SQL statements.
- To declare a variable, use the syntax: **DECLARE @variable_name data_type;**
- You can assign an initial value at declaration using: **DECLARE @variable_name data_type = initial_value;**
- To update the value of a variable, use: **SET @variable_name = new_value;**
- Variables can be used within SQL statements to hold and manipulate data.
- Remember that variables have scope and are typically limited to the script or batch where they are declared.

## RETURN Statement

- In MSSQL, the RETURN statement is used to return an integer value from a stored procedure. The RETURN statement is commonly used to indicate the status of the stored procedure execution, with 0 indicating success and non-zero values indicating various types of errors or warnings.

```
CREATE PROCEDURE myProcedure
AS
BEGIN
  -- Do some processing here
  IF (@someCondition = 1)
  BEGIN
    -- Return success
    RETURN 0
  END
  ELSE
  BEGIN
    -- Return an error
    RETURN 1
  END
END
```

- Example of EXEC:

```
DECLARE @returnValue INT

EXEC @returnValue = myProcedure @inputParameter1 = 'value1', @inputParameter2 = 42
```

## Output

In Microsoft SQL Server (MS SQL), you can use the OUTPUT clause within a stored procedure (SP) to return result sets or captured values from data manipulation operations. The OUTPUT clause allows you to retrieve and work with the modified data or metadata resulting from INSERT, UPDATE, DELETE, or MERGE statements.

INSERT|UPDATE|DELETE|MERGE INTO target_table OUTPUT [output_expression_list] [INTO output_table]

Example:

```
CREATE PROCEDURE UpdateEmployeeSalary
   @employeeID INT,
   @newSalary DECIMAL(10, 2) OUTPUT
AS
BEGIN
   UPDATE employees
   SET salary = @newSalary
   OUTPUT inserted.salary INTO @newSalary
   WHERE employee_id = @employeeID;
END
```

## Advantages of SP

1. Improved Performance: Stored procedures are compiled and optimized, which can result in improved performance over dynamic SQL statements. This is because stored procedures are cached in memory, which reduces the overhead of parsing and optimizing the SQL code every time it's executed.

2. Security: Stored procedures can be used to control access to the database by limiting access to the data itself. By granting permissions only to execute the stored procedures, you can ensure that unauthorized users can't access sensitive data.

3. Code Reusability: Stored procedures can be used as building blocks for larger applications. By breaking down complex logic into smaller, reusable pieces, developers can create applications more efficiently and with less duplication of code.

4. Maintenance: Stored procedures are easier to maintain because changes can be made to the procedure itself, rather than to every SQL statement that references it. This makes it easier to update applications and maintain consistency across multiple database objects.

5. Consistency: Stored procedures can enforce consistency across multiple applications that access the same database. By enforcing business rules within the stored procedures, you can ensure that data is entered and retrieved in a consistent manner.

6. Portability: Stored procedures can be easily moved between different database platforms, which can make it easier to migrate applications from one platform to another.

## Ways of Calling a SP

There are several ways to call stored procedures in Microsoft SQL Server (MSSQL), including:

1. EXECUTE or EXEC command: This is the most common way to call a stored procedure in MSSQL. You can use the EXECUTE or EXEC command followed by the name of the stored procedure and any parameters that need to be passed to the procedure. For example, if you have a stored procedure called "GetCustomers", you can call it using the following command:

```
EXEC GetCustomers;
```

If the stored procedure requires parameters, you can pass them like this:

```
EXEC GetCustomers @City = 'New York';
```

2. Stored Procedure with OUTPUT Parameters: Stored procedures that have OUTPUT parameters return values to the calling program or stored procedure.
3. Using sp_executesql system stored procedure: This stored procedure takes an SQL statement as input and can be used to execute dynamic SQL statements. It can be used to execute a stored procedure and pass parameters to it.
4. Using an application program: Stored procedures can also be called from application programs, such as a .NET application, using SQL Server Native Client or ODBC APIs.
5. Using SQL Server Management Studio (SSMS): In SSMS, you can execute a stored procedure by simply right-clicking on it and selecting "Execute Stored Procedure". You can also use the "Execute" button on the toolbar or press F5 to execute the procedure.
6. Using the SQLCMD utility: The SQLCMD utility is a command-line tool that can be used to execute SQL scripts and stored procedures. You can call a stored procedure using the following command:

```
sqlcmd -E -Q "EXEC GetCustomers"
```

These are some of the common ways to call stored procedures in MSSQL. The method you choose will depend on the specific requirements of your application or situation.

## While Loop

In Microsoft SQL Server (MS SQL), the WHILE loop is a control flow statement that allows you to repeatedly execute a block of code as long as a specified condition is true. It provides a way to create iterative loops within a T-SQL script or stored procedure.

The syntax for using the WHILE loop in MS SQL is as follows:

```
WHILE condition
BEGIN
 -- Code block to be executed
END
```

Here's a breakdown of the components:

- **condition**: This is the Boolean expression that determines whether the loop should continue executing or not. The loop will continue as long as the condition evaluates to true.
- **BEGIN** and **END**: These keywords define the start and end of the code block that will be executed repeatedly as long as the condition is true.

Within the code block, you can include any valid T-SQL statements, such as SELECT, INSERT, UPDATE, DELETE, or even nested IF statements and other loops.

Here's an example to illustrate the usage of the WHILE loop in MS SQL:

```
DECLARE @counter INT = 1;
 WHILE @counter <= 10
BEGIN
```

```
PRINT 'Current count: ' + CAST(@counter AS VARCHAR(10));
SET @counter = @counter + 1;
END
```

It's important to use caution when using a WHILE loop to avoid potential infinite loops. Make sure to include appropriate logic within the loop to ensure that the condition will eventually evaluate to false and the loop will terminate.

Other keywords that can be used: BREAK, CONTINUE, GOTO(avoid)

## Cursors

In Microsoft SQL Server, a cursor is a database object that allows you to retrieve and manipulate data row by row. Cursors provide a way to iterate over the result set of a query and perform operations on each row individually. They are often used when you need to perform complex processing or perform row-level operations that cannot be easily accomplished with set-based operations.

Here's a basic example of how to use a cursor in SQL Server:

Declare the cursor: You need to declare a cursor and specify the SELECT statement that defines the result set you want to iterate over.

```
DECLARE @cursor_name CURSOR;

SET @cursor_name = CURSOR FOR

  SELECT column1, column2, ...

  FROM your_table;
```

Open the cursor: Before you can start fetching rows from the cursor, you need to open it.

```
OPEN @cursor_name;
```

Fetch rows: Use the FETCH NEXT statement to retrieve the next row from the cursor.

```
DECLARE @column1 datatype, @column2 datatype, ...
FETCH NEXT FROM @cursor_name
INTO @column1, @column2, ...;
PRINT @column1 + ' ' + @column2;
```

Repeat fetching until no more rows: You can use a loop to continue fetching rows until there are no more rows available.

```
WHILE @@FETCH_STATUS = 0
BEGIN
  -- Process the row
  FETCH NEXT FROM @cursor_name
  INTO @column1, @column2, ...;
END;
```

Close and deallocate the cursor: After you have finished processing the rows, it's important to close and deallocate the cursor to free up resources.

```
CLOSE @cursor_name;
DEALLOCATE @cursor_name;
```

It's worth noting that cursors can have performance implications and may not be the most efficient solution for all scenarios. It's recommended to use set-based operations whenever possible, as they are generally faster and more optimized. Cursors should be used sparingly and when absolutely necessary.

## Try Catch

```
BEGIN TRY
   -- Statements that might cause an error go here
   -- For example, executing a query or performing some operation
   -- that could potentially raise an exception

   -- If an error occurs, the execution jumps to the CATCH block
END TRY
BEGIN CATCH
   PRINT 'Error Number: ' + CAST(ERROR_NUMBER() AS NVARCHAR(10));
   PRINT 'Error Message: ' + ERROR_MESSAGE();
   PRINT 'Error Severity: ' + CAST(ERROR_SEVERITY() AS NVARCHAR(10));
   PRINT 'Error State: ' + CAST(ERROR_STATE() AS NVARCHAR(10));
   PRINT 'Error Line: ' + CAST(ERROR_LINE() AS NVARCHAR(10));
   PRINT 'Error Procedure: ' + ERROR_PROCEDURE();
   -- You can take appropriate action based on this error information
END CATCH;
```

In the **TRY** block, you include the statements that you want to execute. If any errors occur during the execution of those statements, the execution will immediately jump to the **CATCH** block. In the **CATCH** block, you can handle the error by writing appropriate code, such as logging the error message or displaying a user-friendly message.

Additionally, you can access various system functions and variables in the **CATCH** block to retrieve information about the error. For example, the **ERROR_MESSAGE()** function returns the error message associated with the last error that occurred.

Note that the **TRY...CATCH** block is used for error handling within a single batch or transaction. It allows you to catch and handle exceptions that occur during the execution of that specific code block.

1. **ERROR_NUMBER()**: Returns the error number of the last error that occurred.
2. **ERROR_MESSAGE()**: Returns the error message text of the last error that occurred.
3. **ERROR_SEVERITY()**: Returns the severity level of the last error that occurred.
4. **ERROR_STATE()**: Returns the state number of the last error that occurred.
5. **ERROR_LINE()**: Returns the line number at which the error occurred.
6. **ERROR_PROCEDURE()**: Returns the name of the stored procedure or trigger where the error occurred.
7. **@@ERROR**: Returns the error number of the last statement executed.

## THROW

1. **THROW** statement: The **THROW** statement was introduced in SQL Server 2012 and is used to raise an exception and terminate the current batch, stored procedure, or transaction. It allows you to throw a custom error message along with the error number and severity. Throw can only be used in CATCH block and execution will stop after using throw.

   In

   ```
   THROW [ {error_number | @local_variable},
         {message | @local_variable},
         {state | @local_variable}
   Example:
   BEGIN TRY
     -- Statements that might cause an error go here
   END TRY
   BEGIN CATCH
     -- Exception handling code goes here
     THROW 50001, 'Custom error message', 1;
   END CATCH;
   ```

## RAISEERROR

The **RAISERROR** statement is available in older versions of SQL Server and is still supported in the latest versions. It is used to raise a custom error message and can be used both within and outside the **CATCH** block. The **RAISERROR** statement can specify the error message, severity, state, and other options.

```
RAISERROR ( {msg_id | msg_str | @local_variable}
        {,severity, state}
        {,argument [,...n]}
        {,WITH option [,...n]})


BEGIN TRY
  -- Statements that might cause an error go here
END TRY
BEGIN CATCH
  -- Exception handling code goes here
  RAISERROR ('Custom error message', 16, 1);
END CATCH;
```

In this example, the **RAISERROR** statement is used to raise an exception with a custom error message, severity level 16, and state 1.

# Functions

- Functions in MSSQL are database objects that return a single value or a table of values. They can be used to perform a specific task, such as performing a calculation, manipulating data, or returning a result set.
- There are two types of functions in MSSQL: scalar functions and table-valued functions.

- Functions can have input parameters, which are used to pass values into the function. Parameters can be of different types, such as INT, VARCHAR, or DATE.
- Functions can also have default parameter values, which are used if no value is passed for a parameter.
- User-defined functions (UDFs) are functions that are created by users and stored in the database. They can be created using the CREATE FUNCTION statement.
- Functions can be used in conjunction with other MSSQL features, such as stored procedures, triggers, and views.
- Functions can be nested, meaning one function can call another function as part of its definition.
- Functions are useful for encapsulating complex logic, making it easier to reuse code and maintain database performance.
- Functions can be:
    - Deterministic: Same o/p each time: ROUND, SQUARE, etc
    - Non Deterministic: Varying o/p: RANK, DATE, etc

Syntax:

```
CREATE FUNCTION [schema_name.]function_name
( [parameter [datatype] [ = default ] ] [ ,...n ] )
RETURNS return_datatype
[ WITH <function_option> [ ,...n ] ]
AS
BEGIN
   -- Function body
   RETURN return_value;
END;
```

- **CREATE FUNCTION**: This statement is used to create a new function.
- **[schema_name.]function_name**: This is the name of the function. The schema name is optional and specifies the schema in which the function will be created.
- **( [parameter [datatype] [ = default ] ] [ ,...n ] )**: This is the list of input parameters for the function. Parameters are optional and can have a datatype and a default value.
- **RETURNS return_datatype**: This statement specifies the datatype of the value that the function will return.
- **[WITH <function_option> [ ,...n ] ]**: This is an optional clause that specifies various function options, such as SCHEMABINDING, ENCRYPTION, and RECOMPILE.
- **AS**: This keyword indicates the start of the function body.
- **BEGIN...END**: This block contains the body of the function, which can include SELECT statements, calculations, and other programming logic.
- **RETURN return_value;**: This statement specifies the value that the function will return.

## Functions vs Stored Procedures

Functions and stored procedures are both database objects in MSSQL that are used to perform specific tasks, but they have some important differences. Here are the main differences between functions and stored procedures:

1. Return value: The main difference between functions and stored procedures is that functions return a single value or a table of values, whereas stored procedures do not return a value by default. Instead, stored procedures can return data using OUTPUT parameters or by inserting data into a table.
2. Execution: **Functions are executed as part of a query(sql statements in where/having/select) or expression, whereas stored procedures are executed using an EXECUTE statement or called from an application.**
3. Parameter passing: Both functions and stored procedures can have input parameters, but functions are often used to return a computed value based on those parameters, whereas stored procedures are often used to modify data or perform other operations based on those parameters.
4. **Data modification: Functions cannot modify data in the database can only use select, whereas stored procedures can modify data using INSERT, UPDATE, or DELETE statements.**
5. Transaction and exception handling: Stored procedures can be used to handle transactions and exceptions(try - catch) whereas functions cannot. Stored procedures can be part of a transaction and can use COMMIT and ROLLBACK statements to manage transactional integrity.
6. Recompilation: Functions are typically recompiled each time they are executed, whereas stored procedures are compiled and optimized when they are created or modified, which can result in improved performance.
7. Security: Stored procedures can be granted or denied permissions to specific users or roles, whereas functions cannot.

## Built IN Functions

- **Aggregate functions**: These functions perform calculations on sets of values and return a single value. Examples include:
    - **AVG**: Returns the average value of a set of values.
    - **COUNT**: Returns the number of rows in a set.
    - **MAX**: Returns the maximum value in a set.
    - **MIN**: Returns the minimum value in a set.
    - **SUM**: Returns the sum of values in a set.
- **String functions**: These functions work with strings and text data. Examples include:
    - **CONCAT**: Concatenates two or more strings together.
    - **LEN**: Returns the length of a string.
    - **LOWER**: Converts a string to lowercase.
    - **UPPER**: Converts a string to uppercase.
    - **SUBSTRING**: Returns a part of a string.
- **Date and time functions**: These functions work with date and time data. Examples include:

    - **GETDATE**: Returns the current date and time.
    - **DATEADD**: Adds a specified number of units (such as days or months) to a date.
    - **DATEDIFF**: Returns the difference between two dates.
    - **YEAR**, **MONTH**, **DAY**: Extracts the year, month, or day from a date.
- **Conversion functions**: These functions convert data from one datatype to another. Examples include:
    - **CAST**: Converts a value from one datatype to another.
    - **CONVERT**: Converts a value from one datatype to another.
- **Mathematical functions**: These functions perform mathematical operations on numeric data. Examples include:
    - **ABS**: Returns the absolute value of a number.
    - **CEILING**: Returns the smallest integer greater than or equal to a number.
    - **FLOOR**: Returns the largest integer less than or equal to a number.
    - **POWER**: Returns a number raised to a power.

## User Defined Functions

- Scalar functions return a single value, such as a string, number, or date. They can be used in queries, as part of expressions, or as arguments to other functions.

```
CREATE FUNCTION function_name (input_parameter data_type)
RETURNS return_data_type
AS
BEGIN
   -- Function body
END

-- Creating a scalar function that returns the length of a string
CREATE FUNCTION GetStringLength(@string VARCHAR(100))
RETURNS INT
AS
BEGIN
   DECLARE @length INT
   SET @length = LEN(@string)
   RETURN @length
END
GO

-- Calling the scalar function in a SQL query
SELECT GetStringLength('Hello, world!') AS string_length
```

- Table-valued functions return a table of values, which can be used in a query like any other table. They can be of two types: inline table-valued functions and multi-statement table-valued functions.
- Inline table-valued functions are defined using a single SELECT statement and are useful for simple calculations or data manipulations.

```
CREATE FUNCTION function_name (input_parameter data_type)
```

```
RETURNS TABLE
AS
RETURN (
  SELECT ...
)

-- Creating an inline table-valued function that returns the names and ages of employees with a specified job title
CREATE FUNCTION GetEmployeesByJobTitle(@job_title VARCHAR(50))
RETURNS TABLE
AS
RETURN (
  SELECT first_name + ' ' + last_name AS name, age
  FROM employees
  WHERE job_title = @job_title
)
GO

-- Calling the inline table-valued function in a SQL query
SELECT name, age
FROM GetEmployeesByJobTitle('Manager')
```

- Multi-statement table-valued functions are defined using a BEGIN and END block and can contain multiple SELECT statements or other programming logic.

```
CREATE FUNCTION function_name (input_parameter data_type)
RETURNS @table_variable TABLE (column1 data_type, column2 data_type, ...)
AS
BEGIN
  -- Function body
  RETURN
END
```

```
-- Creating a multi-statement table-valued function that returns the names and ages of employees in a department
CREATE FUNCTION GetEmployeesByDepartment(@department_id INT)
RETURNS @employees TABLE (name VARCHAR(100), age INT)
AS
BEGIN
  INSERT INTO @employees (name, age)
  SELECT first_name + ' ' + last_name, age
  FROM employees
  WHERE department_id = @department_id

  RETURN
END
GO
```

```
-- Calling the multi-statement table-valued function in a SQL query
DECLARE @employee_table TABLE (name VARCHAR(100), age INT)

INSERT INTO @employee_table
SELECT name, age
FROM GetEmployeesByDepartment(1)

SELECT *
FROM @employee_table
```

### APPLY

In Microsoft SQL Server (MSSQL), the **APPLY** operator is used to invoke a table-valued function for each row returned by a query. It allows you to join a table with a table-valued function, applying the function's logic to each row of the table.

The **APPLY** operator comes in two forms: **CROSS APPLY** and **OUTER APPLY**.

**1.CROSS APPLY**:

- The **CROSS APPLY** operator returns only the rows from the left table (the table preceding **CROSS APPLY**) for which the table-valued function returns a result.
- It works like an inner join, where only matching rows are returned.
- The table-valued function is called for each row of the left table.
- Example
  ```
  SELECT e.EmployeeID, t.TaskName
  FROM Employees e
  CROSS APPLY dbo.GetTasksByEmployeeID(e.EmployeeID) t;
  ```

**2.OUTER APPLY**

- The OUTER APPLY operator returns all rows from the left table, whether the table-valued function returns a result or not.
- It works like a left join, including all rows from the left table.
- The table-valued function is called for each row of the left table.
- Example:
  ```
  SELECT e.EmployeeID, t.TaskName
  FROM Employees e
  OUTER APPLY dbo.GetTasksByEmployeeID(e.EmployeeID) t;
  ```

## Triggers

- A trigger is a special type of stored procedure that automatically executes in response to certain events, such as insertions, updates, or deletions on a table.
- Triggers are often used to enforce business rules or perform additional actions when data is modified.

- DML Triggers: These triggers execute in response to Data Manipulation Language (DML) events, such as INSERT, UPDATE, or DELETE statements. DML triggers can be defined to execute either before or after the DML event and can be defined at the table level.

- DDL Triggers: These triggers execute in response to Data Definition Language (DDL) events, such as CREATE, ALTER, or DROP statements. DDL triggers can be defined to execute either before or after the DDL event and can be defined at the database level or the server level.

- Logon Triggers: These triggers execute in response to a logon event for a specific instance of MSSQL. Logon triggers can be used to perform security checks or to log logon events.

## DDL Triggers

- DDL triggers can be defined to execute either before or after the DDL event (i.e., **AFTER** or **BEFORE** triggers).
- DDL triggers can be defined at the database level or at the server level, using the **CREATE TRIGGER** statement with the **ON DATABASE** or **ON ALL SERVER** clause, respectively.

```
CREATE TRIGGER [trigger_name]
ON ALL SERVER -- or ON DATABASE
[AFTER | INSTEAD OF] [DDL_EVENT_TYPE]
AS
BEGIN
-- Trigger logic here
END;
```

Ex 1:

```
CREATE TRIGGER prevent_table_drop
ON DATABASE
FOR DROP_TABLE
AS
BEGIN
   PRINT 'Tables cannot be dropped in this database.'
   ROLLBACK;
END;
```

Ex 2:

```
-- Creating a DDL trigger that logs all database schema changes
CREATE TRIGGER LogSchemaChanges
ON ALL SERVER
FOR DDL_DATABASE_LEVEL_EVENTS
AS
BEGIN
   DECLARE @EventData XML
   SET @EventData = EVENTDATA()

   INSERT INTO schema_change_log (event_type, object_type, object_name, event_date)
   VALUES (@EventData.value('(/EVENT_INSTANCE/EventType)[1]', 'NVARCHAR(100)'),
```

```
        @EventData.value('(/EVENT_INSTANCE/ObjectType)[1]', 'NVARCHAR(100)'),
        @EventData.value('(/EVENT_INSTANCE/ObjectName)[1]', 'NVARCHAR(100)'),
        GETDATE())
    END
    GO
```

## Logon triggers

```
CREATE TRIGGER [trigger_name]
ON ALL SERVER
FOR LOGON
AS
BEGIN
-- Trigger logic here
END;
```

Example:

```
CREATE TRIGGER prevent_login_time
ON ALL SERVER
FOR LOGON
AS
BEGIN
  DECLARE @current_time datetime
  SET @current_time = GETDATE()

  IF DATEPART(hour, @current_time) < 9 OR DATEPART(hour, @current_time) > 17
  BEGIN
    PRINT 'You can only log in between 9 AM and 5 PM.'
    ROLLBACK;
  END
END;
```

## DML Triggers

```
CREATE TRIGGER [trigger_name]
ON [table_name]
[FOR/AFTER/INSTEAD OF] [INSERT/UPDATE/DELETE]
[WITH [NO]CHECK [OPTION]]
AS
BEGIN
-- Trigger logic here
END;
```

- **trigger_name** is the name of the trigger.
- **table_name** is the name of the table to which the trigger applies.
- **FOR/AFTER** specifies when the trigger should fire. **FOR** fires before the data manipulation statement, while **AFTER** fires after the data manipulation statement. If not specified, the trigger fires **AFTER** by default. **INSERT/UPDATE/DELETE** specifies the type of DML operation that the trigger should fire on.
- **WITH [NO]CHECK [OPTION]** specifies whether the trigger should check the referential integrity of the affected data. **WITH CHECK** enforces referential integrity, while **WITH**

**NOCHECK** skips the referential integrity check. **OPTION** specifies additional options that can be used with the **WITH CHECK** or **WITH NOCHECK** clause.

- **BEGIN** and **END** enclose the trigger logic.

Inside the trigger body, you can access the **inserted** and **deleted** tables, which contain the affected rows before and after the data manipulation operation, respectively.

Example:

```
CREATE TRIGGER update_customer_total_amount
ON orders
AFTER INSERT
AS
BEGIN
   UPDATE customers
   SET total_amount = total_amount + i.total_amount
   FROM customers c
   INNER JOIN inserted i ON c.customer_id = i.customer_id
END;
```

## Inserted and Deleted table

In MSSQL, **inserted** and **deleted** are two special tables that are used in DML (Data Manipulation Language) triggers to access the rows affected by the data manipulation operation.

- **inserted**: This is a temporary table that is created by MSSQL during an **INSERT** or **UPDATE** operation. It contains the new rows that were inserted or updated in the target table.
- **deleted**: This is also a temporary table that is created by MSSQL during an **UPDATE** or **DELETE** operation. It contains the old rows that were updated or deleted from the target table.

The **inserted** and **deleted** tables are only available within the scope of a trigger and are automatically populated by MSSQL whenever a DML trigger fires. These tables have the same schema as the table on which the trigger is defined, and you can use them in your trigger logic to access the old and new values of the affected rows.

Here's an example of how you can use the **inserted** and **deleted** tables in a trigger to audit changes to a table:

```
CREATE TRIGGER audit_changes
ON orders
AFTER INSERT, UPDATE, DELETE
AS
BEGIN
   IF EXISTS(SELECT * FROM inserted)
   BEGIN
      -- new rows were inserted or updated
      INSERT INTO order_changes (order_id, change_type, change_date)
      SELECT i.order_id, 'INSERT/UPDATE', GETDATE()
      FROM inserted i
   END
```

```
        ELSE
        BEGIN
          -- rows were deleted
          INSERT INTO order_changes (order_id, change_type, change_date)
          SELECT d.order_id, 'DELETE', GETDATE(
          FROM deleted d
        END
      END;
```

Note:

- To know how many rows an operation affects, use @@ROWCOUNT
- After update statement, to see if any columns are updated, gives Boolean answer: columns_updated()
- Triggers can be disabled.

### Nested Triggers

Triggers can call triggers.

This feature can be enabled/disabled: exec sp_configure 'nested_triggers' 0 --(or 1 to enable)

To see trigger level: @@nestlevel

Infinite Loop in triggers: Can happen if trigger calls itself. However, MSSQL allows only 32 nest levels. Hence, infinite loop avoided.

## Indexes

Heap is a table without index.

In MSSQL, an index is a database object that helps to speed up the retrieval of data from tables. Indexes are created on one or more columns of a table and provide a quick way to access the data based on the values in those columns.

The main purpose of an index is to improve the performance of queries that are executed against a table. Without an index, MSSQL must scan the entire table(called TABLESCAN) to find the rows that match the query criteria, which can be slow and inefficient, especially for large tables. However, with an index, MSSQL can quickly locate the relevant rows and return them to the user.

Indexes also provide other benefits, such as:

- Reduced disk I/O: When you query a table, MSSQL reads the data from disk and stores it in memory. If the table has an index, MSSQL can use the index to minimize the amount of disk I/O that is required to retrieve the data.
- Faster sorting and grouping: If you sort or group the data in a query, MSSQL can use the index to perform the operation more quickly than if it had to sort or group the data in memory.
- Primary key and unique constraints: In MSSQL, a primary key or unique constraint is implemented as an index on one or more columns. This ensures that the values in the columns are unique and can be used as a reference to related data in other tables.

### Structure of Index

The structure of a B-tree index can be broken down into the following components:

1. Root node: The root node is the topmost node in the B-tree structure. It contains index keys and pointers to child nodes or data pages.
2. Internal nodes: The internal nodes are non-leaf nodes in the B-tree structure. They contain index keys and pointers to child nodes or data pages.
3. Leaf nodes: The leaf nodes are the bottommost nodes in the B-tree structure. They contain index keys and pointers to the actual data rows in the table.
4. Index key: The index key is the column or set of columns that are used to create the index. The values of these columns are used to sort the index entries.
5. Pointer: The pointer is a reference to the location of the index entry in the leaf node.
6. Page: The index entries are stored in pages, which are blocks of data that are read and written to disk as a unit.

The B-tree structure allows for efficient search and retrieval of data, even for large datasets. When a query is executed, the query optimizer can use the index to quickly locate the relevant rows in the table, without having to scan the entire table. The B-tree structure also allows for efficient updates to the index, as the tree can be reorganized as needed to maintain optimal performance.

## Types of index

1. Clustered index: A clustered index is an index that determines the physical order of data in a table. Each table can have only one clustered index, and it is typically created on the primary key column of the table. When a clustered index is created, the data in the table is physically rearranged to match the order of the index. In the B-Tree the leaf node contains actual data instead of pointer.

2. Non-clustered index: A non-clustered index is an index that is separate from the actual data in a table. It contains a copy of the indexed columns and a pointer to the corresponding row in the table. Each table can have multiple non-clustered indexes, and they are typically created on columns that are frequently used in queries. Upto 999 non clustered index supported.Leaf node in B tree contains pointer to data and not actual data.

3. Unique index: A unique index is an index that enforces the uniqueness of the values in the indexed columns. Each table can have multiple unique indexes, and they are typically created on columns that have a unique/PK constraint.

4. Composite Index: Index made up of more than one column, upto 16 columns. If it is also an unique index, the combination of col values must be unique, not individual value.

5. Covering index: A covering index is an index that includes all of the columns that are needed to satisfy a query, so that the query can be fulfilled by scanning the index alone, without having to read the data from the table. This can result in a significant performance improvement for queries that access a large amount of data.

6. Filtered index: A filtered index is an index that is created on a subset of rows in a table, based on a filter predicate. This can be useful for queries that frequently access a specific subset of data, as it allows for faster access to that data.

## Index Considerations

1. Choose the right index type: Choose the appropriate index type based on the query patterns and data characteristics of the table.

2. Identify the most frequently used queries: Identify the queries that are most frequently used and optimize the indexes to support them.

3. Minimize the number of indexes: While indexes can improve query performance, they also require additional disk space and can impact write performance. Therefore, it's important to minimize the number of indexes while still providing adequate coverage for the most frequently used queries.
4. Choose the right column order: For non-clustered indexes, choose the column order carefully to ensure that the index can be used efficiently for the most frequently used queries.
5. Consider the data distribution: Consider the distribution of data within the table and choose index columns that have high selectivity to minimize the number of rows that need to be scanned.
6. Avoid over-indexing: Over-indexing can result in reduced query performance due to the additional overhead of maintaining the indexes. Only create indexes that are necessary to support the most frequently used queries.
7. Regularly monitor and optimize indexes: Regularly monitor the performance of queries and indexes and make adjustments as necessary to optimize performance.
8. Try to avoid nulls on index columns

## Syntax of Index:

```
CREATE [UNIQUE] [CLUSTERED | NONCLUSTERED] INDEX index_name
ON table_name (column_name [ASC | DESC], column_name [ASC | DESC], ...)
```

## INCLUDE

hen working with a database, creating indexes on tables can significantly improve query performance by facilitating efficient data retrieval. An index is a database structure that organizes data in a specific way to allow for faster searching and retrieval of information. Including columns in an index can further enhance its effectiveness. Here's some information about including columns in indexes in Microsoft SQL Server (MSSQL):

1. Non-Clustered Indexes:
    - Non-clustered indexes are separate data structures that store a copy of a portion of the table's data.
    - By default, non-clustered indexes include the indexed column(s) along with a pointer to the actual data row.
    - Including additional columns in a non-clustered index improves the index's covering capability, enabling queries to retrieve all required data from the index itself, without accessing the table's data pages.
    - Including columns can enhance query performance by reducing the number of disk I/O operations and improving data locality.
2. Included Columns:
    - Included columns are additional non-key columns that are stored with the index leaf pages but not included in the index's key structure.
    - Including non-key columns in the index as included columns extends the coverage of the index, making it a covering index for more queries.
    - Included columns can be useful when the selected non-key columns are frequently accessed in queries and need to be retrieved efficiently.

3. Syntax for Including Columns:
    - When creating or altering a non-clustered index, you can specify additional columns as included columns using the **INCLUDE** keyword.
    - Included columns do not affect the order of the data in the index, but they can improve query performance.
    - Here's an example of creating a non-clustered index with included columns: CREATE NONCLUSTERED INDEX IX_IndexName ON TableName (KeyColumn) INCLUDE (IncludedColumn1, IncludedColumn2);
4. Considerations:
    - Including too many columns in an index may increase the index's size and impact insert, update, and delete operations.
    - Choose included columns judiciously, based on the columns frequently accessed by queries and the ones that provide the most benefit in terms of query performance.

By including additional columns in non-clustered indexes as included columns, you can create covering indexes that optimize query performance by reducing disk I/O and improving data locality. However, it's essential to strike a balance between the number of included columns and the impact on data modification operations.

## Index Fragmentation

Index fragmentation is a condition that occurs when the pages of an index are not stored contiguously on disk. Fragmentation can occur in both clustered and nonclustered indexes in MSSQL, and it can have a negative impact on database performance.

There are two types of index fragmentation:

1. **Internal fragmentation**: This occurs when the data on a page is not completely full, meaning there is unused space on the page. This can happen when rows are deleted or when data is inserted into an index with a fill factor less than 100. When internal fragmentation occurs, there may be more pages in the index than necessary, which can increase the time it takes to read data from the index(index scan takes more time).
2. **External fragmentation**: This occurs when the pages of an index are not physically adjacent to each other on disk. This can happen when data is inserted into an index that has a high fill factor or when the index is rebuilt or reorganized. External fragmentation can lead to increased disk I/O and decreased performance.

To mitigate index fragmentation, MSSQL provides two main options:

1. **Rebuilding indexes**: This option creates a new index from scratch, copying the data from the old index to the new one. This can eliminate both internal and external fragmentation, but it can be an expensive operation in terms of time and disk space. Do when fragment > 30%
2. **Reorganizing indexes**: This option rearranges the pages of an existing index to eliminate external fragmentation. Reorganizing an index is faster than rebuilding it, but it does not eliminate internal fragmentation. Do when fragment btw 11 to 30%

You can use the **ALTER INDEX** statement in MSSQL to rebuild or reorganize indexes. For example:

-- Rebuild the idx_customer_last_name index on the customers table

```
ALTER INDEX idx_customer_last_name ON customers REBUILD;
-- Reorganize the idx_orders_customer_id index on the orders table
ALTER INDEX idx_orders_customer_id ON orders REORGANIZE;

-- Check the fragmentation of the idx_orders_customer_id index on the orders table
SELECT * FROM sys.dm_db_index_physical_stats(DB_ID(), OBJECT_ID('orders'),
                                    INDEXPROPERTY(OBJECT_ID('orders'), 'idx_orders_customer_id', 'IndexId'),
NULL, 'DETAILED');
```

Can also check fragmentation in properties.

# Aggregate Queries

## OVER()

"OVER" clause in Microsoft SQL Server, it is used in combination with aggregate functions to perform calculations over a specific window or partition of rows. On its own, it is used to apply a function on all rows and display it as a column with all rows.

Examples:

```
SELECT Column1, Column2, SUM(Column2) OVER () AS Total FROM TableName;
```

### ORDER BY: to define the sorting order within a window.

1.For running totals

```
SELECT Column1, Column2, SUM(Column2) OVER (ORDER BY Column1) AS RunningTotal FROM TableName;
```

2.For rank/rownumber:

```
SELECT Column1, Column2, RANK() OVER (ORDER BY Column2 DESC) AS Rank FROM TableName;

SELECT Column1, Column2, row_number() OVER (ORDER BY Column2 DESC) AS rwnbr FROM TableName;
```

3. Retrieve the first and last values in a window using the FIRST_VALUE() and LAST_VALUE()

```
SELECT Column1, Column2, FIRST_VALUE(Column2) OVER (ORDER BY Column1) AS FirstValue,

LAST_VALUE(Column2) OVER (ORDER BY Column1) AS LastValue FROM TableName;
```

### Partition By

The "PARTITION BY" clause in the "OVER" clause is used to divide the result set into partitions based on the values in one or more columns. It allows you to perform calculations and apply window functions independently within each partition.

Ex:

```
SELECT Column1, Column2, SUM(Column2) OVER (PARTITION BY Column1) AS PartitionSum

FROM TableName;
```

- The "PARTITION BY" clause defines the column(s) that determine the partitions. In this example, the result set is divided into separate partitions based on the distinct values in "Column1".
- The "OVER" clause specifies the window within which the calculation is performed. By using "PARTITION BY" in conjunction with the aggregate function, the calculation is applied separately within each partition.

## ROWS BETWEEN

```
SELECT Column1, Column2, SUM(Column2) OVER (ORDER BY Column1 ROWS BETWEEN
UNBOUNDED PRECEDING AND CURRENT ROW) AS RunningTotal FROM TableName;
```

In the above query, the "ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW" clause specifies that the window for the "SUM()" function includes all the rows from the beginning of the partition up to and including the current row.

You can also modify the "ROWS BETWEEN" clause to define different window ranges, such as "BETWEEN x PRECEDING AND y FOLLOWING" to include a specific number of preceding and following rows. There are several options for specifying the range:

- BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW: This range includes all rows from the beginning of the partition up to and including the current row.
- BETWEEN x PRECEDING AND y FOLLOWING: This range includes x preceding rows and y following rows, relative to the current row within the partition.
- BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING: This range includes the current row and all rows following it within the partition.
- BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING: This range includes all rows within the partition.

PARTITION BY and ROWS BETWEEN:

```
SELECT Column1, Column2, SUM(Column2) OVER (PARTITION BY Column1 ORDER BY
Column3 ROWS BETWEEN 2 PRECEDING AND 2 FOLLOWING) AS Calculation FROM
TableName;
```

In this example, the result set is partitioned by the values in "Column1". Within each partition, the "SUM()" function is applied to "Column2" with a window that includes the current row and two preceding rows, as well as two following rows. The calculation is performed based on the order specified by "Column3".

## RANGE VS ROWS

In SQL, the "RANGE" and "ROWS" clauses are used in combination with the "OVER" clause to define the window within which calculations and aggregations are performed. The main difference between "RANGE" and "ROWS" lies in how they determine the rows included in the window.

1. RANGE: The "RANGE" clause operates on the values of the ordering column(s) and includes rows that have equal values to the current row. It defines a logical range based on the values, rather than

the physical position of the rows. This means that rows with equal values are treated as peers, regardless of their actual position in the result set.

2. ROWS: The "ROWS" clause operates based on the physical position of the rows and includes a specified number of preceding or following rows relative to the current row. It defines the range based on the physical order of the rows in the result set.

Ex:

```
SELECT OrderDate, TotalAmount, SUM(TotalAmount) OVER (PARTITION BY OrderDate
ORDER BY TotalAmount RANGE UNBOUNDED PRECEDING) AS CumulativeSum FROM Orders;
```

In this example, the "RANGE UNBOUNDED PRECEDING" includes all rows with equal "TotalAmount" values, regardless of their physical position. So if there are multiple rows with the same "TotalAmount", they will be treated as peers and included in the cumulative sum calculation.

```
SELECT OrderDate, TotalAmount, SUM(TotalAmount) OVER (PARTITION BY OrderDate
ORDER BY TotalAmount ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS
CumulativeSum FROM Orders;
```

In this example, the "ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW" includes all rows from the beginning of the partition up to and including the current row. The range is based on the physical position of the rows in the result set.

So, in summary, "RANGE" operates on the values of the ordering column(s), treating equal values as peers, while "ROWS" operates based on the physical position of the rows. The choice between "RANGE" and "ROWS" depends on the specific requirements of your calculation and the nature of your data.

## Ranking Functions

In Microsoft SQL Server (MSSQL), ranking functions are used to assign a rank or row number to each row in a result set based on specified criteria. These functions help to perform various ranking operations and analyze data in a meaningful way. MSSQL provides several ranking functions, including ROW_NUMBER(), RANK(), DENSE_RANK(), and NTILE().

1. ROW_NUMBER(): The ROW_NUMBER() function assigns a unique sequential integer to each row within the partition of a result set. It does not consider any specific order or ranking criteria.
2. RANK(): The RANK() function assigns a unique rank to each row within the partition, based on the ordering specified in the ORDER BY clause. If multiple rows have the same values, they receive the same rank, and the subsequent rank is skipped.
3. DENSE_RANK(): The DENSE_RANK() function is similar to RANK(), but it assigns a unique rank to each row without skipping any ranks. If multiple rows have the same values, they all receive the same rank, and the subsequent rank is not skipped.

4. NTILE(N): The NTILE() function divides the rows into a specified number(N) of groups (buckets) and assigns a bucket number to each row. The function evenly distributes the rows across the buckets based on the ordering specified in the ORDER BY clause.

Example:

```
SELECT CustomerName, OrderDate,
ROW_NUMBER() OVER (ORDER BY OrderDate) AS RowNum,
RANK() OVER (PARTITION BY CustomerName ORDER BY OrderDate) AS Rank,
DENSE_RANK() OVER (PARTITION BY CustomerName ORDER BY OrderDate) AS DenseRank,
NTILE(4) OVER (ORDER BY OrderDate) AS Bucket
FROM  Orders
```

# Advanced Analytic Functions

## LAG and LEAD:

The LAG and LEAD functions allow you to access data from previous or subsequent rows within the same result set based on a specified order. Here's an example using the LAG

```
SELECT OrderID, OrderDate,
  LAG(OrderDate) OVER (ORDER BY OrderDate) AS PreviousOrderDate
FROM Orders
```

This query retrieves the OrderID, OrderDate, and the previous OrderDate using the LAG function, which helps analyze the time difference between consecutive orders.

## FIRST_VALUE and LAST_VALUE:

The FIRST_VALUE and LAST_VALUE functions return the first or last value in a specified order within a window frame. Here's an example using the FIRST_VALUE function:

```
SELECTProductName, UnitPrice,
   FIRST_VALUE(UnitPrice) OVER (PARTITION BY CategoryID ORDER BY UnitPrice) AS
FirstPrice FROM Products
```

## PERCENTILE_CONT and PERCENTILE_DISC:

The PERCENTILE_CONT and PERCENTILE_DISC functions calculate percentile values for a specific expression within a group of rows. Here's an example

```
SELECT OrderID, OrderDate,
   PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY OrderDate) OVER () AS
MedianOrderDate
FROM Orders
```

This query calculates the median OrderDate using the PERCENTILE_CONT function. It helps determine the middle point in the distribution of order dates.

## CUME_DIST:

The CUME_DIST function calculates the cumulative distribution of a value within a group.

```
SELECT
   ProductName, UnitPrice,
   CUME_DIST() OVER (ORDER BY UnitPrice) AS CumulativeDistribution
FROM  Products
```

This query retrieves the ProductName, UnitPrice, and the cumulative distribution of each product's price using the CUME_DIST function. It helps understand the relative position of each product's price within the overall price distribution.

## STRING_AGG:

The STRING_AGG function concatenates values from multiple rows into a single string, with an optional delimiter between each value.

```
SELECT CustomerID, STRING_AGG(OrderID, ',') AS ConcatenatedOrderIDs
FROM Orders GROUP BY CustomerID
```

## Grouping Sets

Grouping Sets is a feature in Microsoft SQL Server (MSSQL) that allows you to perform multiple grouping operations within a single query. It enables you to aggregate data at different levels and generate results for various combinations of grouping columns. The result is a consolidated set of aggregated data based on the specified grouping sets.

The GROUPING SETS clause is used to define multiple groupings in a single query. Each grouping set represents a combination of columns by which the data is grouped. The result set includes rows for each grouping set, along with the corresponding aggregated values.

```
SELECT
   ProductCategory,
   ProductSubcategory,
   SUM(Quantity) AS TotalQuantity,
   SUM(Revenue) AS TotalRevenue
FROM
   Sales
GROUP BY
   GROUPING SETS (
      (ProductCategory, ProductSubcategory),
      (ProductCategory),
      ()
   )
```

In this example, the query groups the data by the combination of ProductCategory and ProductSubcategory, as well as by ProductCategory alone. The GROUPING SETS clause specifies three grouping sets: the first one includes both ProductCategory and ProductSubcategory, the second one includes only ProductCategory, and the third one is an empty set (denoted by ()) representing the grand total.

## ROLLUP and CUBE

In Microsoft SQL Server (MSSQL), both ROLLUP and CUBE are extensions to the GROUP BY clause that enable multi-level aggregations and facilitate the generation of comprehensive summary results.

1. ROLLUP: The ROLLUP operator allows you to create hierarchical rollup summaries within a query. It generates a result set that includes subtotal rows and a grand total row. The

ROLLUP operator works by progressively rolling up the data based on the specified columns in the GROUP BY clause.

Here's an example to illustrate the usage of ROLLUP in MSSQL:

```
SELECT
   ProductCategory, ProductSubcategory,
   SUM(Quantity) AS TotalQuantity, SUM(Revenue) AS TotalRevenue
FROM
   Sales
GROUP BY
   ROLLUP (ProductCategory, ProductSubcategory)
```

In this example, the ROLLUP operator is applied to the ProductCategory and ProductSubcategory columns. The result set will include subtotal rows for each level of aggregation (ProductCategory, ProductSubcategory, and the grand total). The TotalQuantity and TotalRevenue are aggregated values for each level.

2. CUBE: The CUBE operator, like ROLLUP, is an extension to the GROUP BY clause that generates a result set with all possible combinations of the specified columns. It produces a comprehensive summary that includes subtotal and grand total rows for each combination of columns.
   Here's an example to illustrate the usage of CUBE in MSSQL:

```
SELECT
   ProductCategory, ProductSubcategory,
   SUM(Quantity) AS TotalQuantity, SUM(Revenue) AS TotalRevenue
FROM  Sales
GROUP BY CUBE (ProductCategory, ProductSubcategory)
```

## PIVOT

In Microsoft SQL Server (MSSQL), the PIVOT operator is used to rotate rows into columns, transforming the result set based on a specific column's values. The PIVOT operator allows you to perform dynamic or static pivoting of data, making it easier to analyze and present information in a different format.

```
SELECT <non-pivoted column(s)>,
    [pivot column1], [pivot column2], ...,
    [aggregate function(column to be aggregated)]
FROM <source table>
PIVOT
(
  <aggregate function(column to be aggregated)>
  FOR <pivot column>
  IN ([pivot value1], [pivot value2], ...)
) AS <alias>

SELECT * FROM
(
  SELECT Category, Product, SalesAmount
  FROM Sales
```

```
) AS SourceTable
PIVOT
(
   SUM(SalesAmount)
   FOR Product
   IN ([Product A], [Product B], [Product C])
) AS PivotTable;
```

---

```
Category    Product A    Product B    Product C
-----------------------------------------------
Category 1  1000         500          800
Category 2  1500         2000         1200
Category 3  800          600          900
```

## UNPIVOT

In Microsoft SQL Server (MSSQL), the UNPIVOT operator is used to convert columns into rows, reversing the process performed by the PIVOT operator. The UNPIVOT operation is useful when you want to transform a pivoted result set back into its original row-based format.

The basic syntax of the UNPIVOT operator in MSSQL is as follows:

```
SELECT <non-unpivoted column(s)>
FROM
(
   SELECT <pivoted column(s)>
   FROM <source table>
) AS PivotTable
UNPIVOT
(
   <value column>
   FOR <column to create>
   IN ([pivot column1], [pivot column2], ...)
) AS UnpivotTable;
```

Example:

```
SELECT Category, Product, SalesAmount
FROM
(
   SELECT Category, [Product A], [Product B], [Product C]
   FROM PivotTable
) AS PivotSource
UNPIVOT
(
   SalesAmount
   FOR Product IN ([Product A], [Product B], [Product C])
) AS UnpivotTable;
```

---

```
Category    Product      SalesAmount
```

---
Category 1 Product A 1000
Category 1 Product B 500
Category 1 Product C 800
Category 2 Product A 1500
Category 2 Product B 2000
Category 2 Product C 1200

## Sys Schema

In Microsoft SQL Server (MS SQL), the "sys" schema is a system schema that contains a collection of system-defined objects. These objects are used by the SQL Server database engine to store and manage metadata about the database and its objects.

The "sys" schema provides access to various system catalog views and functions that allow users to query information about database objects, system configurations, server statistics, and more. These system catalog views and functions expose metadata about tables, columns, indexes, views, stored procedures, user-defined functions, and other database-related entities.

Here are a few commonly used system catalog views in the "sys" schema:

- **sys.tables**: Contains information about tables in the database.
- **sys.columns**: Provides information about columns within tables.
- **sys.indexes**: Stores details about indexes on tables.
- **sys.views**: Contains information about views defined in the database.
- **sys.procedures**: Stores details about stored procedures.
- **sys.schemas**: Provides information about all schemas in the database.

You can query these system catalog views, along with other views and functions in the "sys" schema, to retrieve information about the structure and properties of database objects.

For example, to retrieve a list of all tables in a database, you can use the following query:

```
SELECT name FROM sys.tables;
```

### DMVs and Functions

Dynamic Management Views (DMVs) and functions in Microsoft SQL Server are system database objects that provide valuable information about the SQL Server instance, its databases, and various aspects of its performance and configuration. DMVs and functions are used to query and retrieve metadata, statistics, and other diagnostic information.

Here are some commonly used DMVs and functions in SQL Server:

1. sys.dm_exec_requests: Provides information about the currently executing requests or queries.
2. sys.dm_exec_sessions: Contains details about active user connections and their associated session-level properties.
3. sys.dm_exec_query_stats: Provides aggregate performance statistics for cached query plans.
4. sys.dm_os_wait_stats: Displays information about the various wait types encountered by SQL Server.
5. sys.dm_db_index_usage_stats: Shows usage information for indexes in a database, including the number of seeks, scans, and updates.

6. sys.dm_db_missing_index_details: Identifies missing indexes that could potentially improve query performance.
7. sys.dm_os_performance_counters: Retrieves performance counter information related to the SQL Server instance.
8. sys.dm_db_file_space_usage: Provides details about the space used by database files.

In addition to DMVs, SQL Server also provides system functions that can be used in queries to perform specific calculations or retrieve information. Some commonly used functions include:

1. GETDATE(): Returns the current system date and time.
2. OBJECT_ID(): Retrieves the object ID of a database object by specifying its name.
3. DB_NAME(): Returns the name of the current database.
4. DATABASEPROPERTYEX(): Retrieves various properties of a database, such as its compatibility level or recovery model.
5. SERVERPROPERTY(): Returns information about the SQL Server instance, such as its version or edition.

These are just a few examples, and SQL Server offers a wide range of DMVs and functions to cater to different administrative and diagnostic needs. You can explore the SQL Server documentation for a comprehensive list and detailed information about each DMV and function.

## GUIDs

Globally Uniques Identifiers: To generate unique values. GUIDs are 128-bit unique identifiers that are generated using algorithms designed to guarantee their uniqueness across different computers and networks.

Set @nvalue = NEWID();

Using GUID in table:

```
CREATE TABLE MyTable
(
   ID uniqueidentifier DEFAULT NEWID() PRIMARY KEY,
   Name nvarchar(50) NOT NULL,
   -- Other columns...
);
```

SEQUENCE
Sequences are used to generate a sequence of numeric values in an ascending or descending order. They are often used to generate primary key values or other unique identifiers.

```
-- Create a sequence
CREATE SEQUENCE MySequence
   START WITH 1
   INCREMENT BY 1
   MINVALUE 1
   MAXVALUE 99999
   CYCLE;

-- Use the sequence to generate values
```

```
SELECT NEXT VALUE FOR MySequence;

-- Reset the sequence
ALTER SEQUENCE MySequence RESTART WITH 1;

-- Delete the sequence
DROP SEQUENCE MySequence;

--In create table:
-- Create the table using the sequence
CREATE TABLE MyTable
(
   ID INT DEFAULT NEXT VALUE FOR MySequence PRIMARY KEY,
   Name NVARCHAR(50) NOT NULL,
   -- Other columns...
);
```

# Handling XML data

## XML Data Type:

MSSQL provides a built-in XML data type that allows you to store and manipulate XML data within a column. Use the xml data type when defining a column that will store XML data.

```
CREATE TABLE MyTable
(
   ID INT PRIMARY KEY,
  XMLData XML
);
```

## XML Functions:

- MSSQL offers a variety of XML functions to query, modify, and transform XML data stored in columns.
- Functions like XML value(), XML query(), and XML modify() are commonly used for XML data manipulation.
- These functions allow you to extract values, perform filtering, join XML data, and more.

```
-- Extract a value from XML
SELECT XMLData.value('(/Root/Element)[1]', 'nvarchar(max)') AS ExtractedValue
FROM MyTable;

-- Modify XML data
UPDATE MyTable
SET XMLData.modify('replace value of (/Root/Element)[1] with "New Value"');
```

## XML Indexing:

- For efficient querying of XML data, consider using XML indexes.
- XML indexes improve performance by providing optimized paths for XML queries.
- You can create primary, secondary, or selective XML indexes based on your query patterns.

```
-- Create an XML index
CREATE PRIMARY XML INDEX PX_MyTable_XMLData
ON MyTable (XMLData);

-- Query XML using the XML index
SELECT *
FROM MyTable
WHERE XMLData.exist('/Root/Element[@Attribute="Value"]') = 1;
```

## XML Schema Collections:

- XML schema collections define the structure and validation rules for XML data stored in columns.
- You can create and associate XML schema collections with XML columns to enforce data integrity and enable typed XML operations.
- XML schema collections are useful when you have specific XML structure requirements.

```
-- Create an XML schema collection
CREATE XML SCHEMA COLLECTION MySchemaCollection AS
N'<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Root">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Element" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>';

-- Associate the schema collection with the XML column
ALTER TABLE MyTable
ALTER COLUMN XMLData xml(CONTENT MySchemaCollection);
```

## XQuery Language:

- XQuery is a powerful language for querying and manipulating XML data.
- MSSQL supports XQuery for XML data retrieval and modification.
- XQuery provides a rich set of expressions and functions for filtering, selecting, and transforming XML data.
-

```
-- Perform XQuery on XML data
SELECT XMLData.query('/Root/Element')
FROM MyTable;
```

## OPENXML Function:

- The OPENXML function is used to shred XML data into relational rows and columns.
- It allows you to query XML data using SQL statements by converting the XML into a rowset.
- OPENXML can be useful when you need to work with XML data in a more traditional relational format.

```
-- Shred XML data using OPENXML
DECLARE @XMLData XML;
SET @XMLData = '<Root><Element>Value 1</Element><Element>Value
2</Element></Root>';

EXEC sp_xml_preparedocument @handle OUTPUT, @XMLData;

SELECT *
FROM OPENXML(@handle, '/Root/Element', 1)
WITH (ElementValue VARCHAR(100));

EXEC sp_xml_removedocument @handle;
```

## FOR XML Clause:

- The FOR XML clause is used to generate XML output from relational data.
- It allows you to retrieve query results as XML, making it convenient for integration with other systems or web services.
- FOR XML can be used with various options to control the XML structure and formatting.

```
SELECT ID, Name
FROM MyTable
FOR XML AUTO, ROOT('Root');
```

## XML Validation:

XML data can be validated against an XML schema using the WITH XMLSCHEMA option in the CAST or CONVERT functions.

This validation ensures that the XML data adheres to the specified schema.

## Import/Export XML data

To import and export data to/from Microsoft SQL Server (MSSQL), you have several options available. Here are some common methods:

1. SQL Server Import and Export Wizard:
    - SQL Server Management Studio (SSMS) provides an Import and Export Wizard that allows you to easily transfer data between different data sources.
    - With the wizard, you can choose the source and destination, specify mappings, and customize the import/export process.
    - You can access the wizard by right-clicking on a database in SSMS, selecting Tasks, and then choosing either Import Data or Export Data.
2. Bulk Copy Program (BCP):
    - BCP is a command-line utility provided by MSSQL for importing and exporting data.
    - It allows you to efficiently transfer large amounts of data between files and MSSQL tables.
    - BCP supports various file formats, including CSV, text files, and native MSSQL binary format.
    - Here's an example of using BCP to export data to a CSV file:

bcp DatabaseName.SchemaName.TableName out C:\Path\To\OutputFile.csv -c -t , -S ServerName -U Username -P Password

3. SQL Server Integration Services (SSIS):
    - SSIS is a powerful data integration and transformation tool provided by MSSQL.
    - It offers a visual development environment for creating sophisticated ETL (Extract, Transform, Load) processes.
    - SSIS allows you to design packages to import and export data from various sources, perform transformations, and load it into MSSQL tables.
4. T-SQL Commands:
    - You can use T-SQL commands to import and export data directly within SQL scripts.
    - For example, you can use the **BULK INSERT** statement to import data from a file into a table.
    - Similarly, you can use the **INSERT INTO ... SELECT** statement to export data from a table into another table or a file.
    - OPENROWSET
      SELECT *
      FROM OPENROWSET(BULK 'C:\Path\To\YourXMLFile.xml', SINGLE_CLOB) AS XMLData;

# JSON

## JSON Data Type:

MSSQL introduced native support for the JSON data type starting from version 2016. You can store JSON data in columns of the JSON data type, which allows efficient storage and retrieval of JSON documents.

```
CREATE TABLE MyTable (
   ID INT,
   Data JSON
);

INSERT INTO MyTable VALUES (1, '{"name": "John", "age": 30}');
```

# JSON Functions:

MSSQL provides a set of built-in functions to work with JSON data. Some commonly used functions include:

1. JSON_VALUE: Extracts a scalar value from a JSON string.

```
SELECT JSON_VALUE(Data, '$.name') AS Name FROM MyTable;
```

2. JSON_QUERY: Extracts an object or an array from a JSON string.

```
SELECT JSON_QUERY(Data, '$.hobbies') AS Hobbies FROM MyTable;
```

3. ISJSON: Checks whether a string contains valid JSON.

```
SELECT Data FROM MyTable WHERE ISJSON(Data) = 1;
```

4. JSON_MODIFY: Modifies the value of a property in a JSON string.

UPDATE MyTable SET Data = JSON_MODIFY(Data, '$.age', 31) WHERE ID = 1;

### FOR JSON

SELECT ID, Data FROM MyTable FOR JSON AUTO;

## Temporal Table

### Ways of handling data history

Type 0: Don’t preserve, only original value, no updates allowed.
Type 1: Don’t preserve, only latest value, no history saved.
Type 2: Every version stored in a separaterow with a colmn giving period of validity
Type 3: Keep limited history for select columns using additional column in same row.
Type 4: Keep history in separate table while original table has latest values.

MSSQL uses combination of type 2 and 4
A temporal table in Microsoft SQL Server (MSSQL) is a type of table that allows you to store and query data as it existed at different points in time. It provides built-in support for tracking changes to data over time, making it useful for scenarios where you need to analyze historical data or perform point-in-time analysis. Here are some key points about temporal tables in MSSQL:

- Table Structure: A temporal table consists of two parts: the main table and a history table. The main table stores the current version of the data, while the history table maintains the historical versions of the data.
- Period Columns: Temporal tables have two special columns called period columns that define the validity period for each row. These columns represent the start and end timestamps during which a particular version of the row was valid. By default, the period columns are named "SysStartTime" and "SysEndTime."
- System-Versioning: MSSQL handles the automatic management of historical data using system-versioning. When you enable system-versioning on a temporal table, the database automatically populates the history table with previous versions of the data as changes occur in the main table.
- Querying Temporal Tables: Temporal tables allow you to query data as it existed at a specific point in time or during a time range. You can use the special syntax and functions provided by MSSQL to retrieve the appropriate version of the data based on the desired timestamp or time range.
- Modifying Temporal Tables: When you update or delete rows in a temporal table, MSSQL handles the modifications by closing the current version of the row in the main table and inserting a new row with the updated data. The previous version of the row is then moved to the history table, preserving the historical changes.
- System-Versioning Configuration: System-versioning for temporal tables is configured using the PERIOD FOR SYSTEM_TIME clause in the table definition. You specify the names of the period columns and the history table to be associated with the temporal table.

Example:

```
-- Create the main table
```

```
CREATE TABLE Employees
(
   EmployeeID INT PRIMARY KEY,
   Name VARCHAR(50),
   Salary DECIMAL(10, 2),
   SysStartTime DATETIME2 GENERATED ALWAYS AS ROW START HIDDEN NOT NULL,
   SysEndTime DATETIME2 GENERATED ALWAYS AS ROW END HIDDEN NOT NULL,
--HIDDEN is optional, used to hide the cols
   PERIOD FOR SYSTEM_TIME (SysStartTime, SysEndTime)
)
WITH (SYSTEM_VERSIONING = ON (HISTORY_TABLE = dbo.EmployeesHistory));

-- Insert data into the main table
INSERT INTO Employees (EmployeeID, Name, Salary)
VALUES (1, 'John Doe', 5000),
     (2, 'Jane Smith', 6000)
-- Query the current data from the main table
SELECT * FROM Employees;

-- Query the historical data from the history table
SELECT * FROM EmployeesHistory FOR SYSTEM_TIME ALL;
SELECT * FROM EmployeesHistory FOR SYSTEM_TIME FROM '2019-01-01' TO '2022-01-01';
SELECT * FROM EmployeesHistory FOR SYSTEM_TIME
BETWEEN '2019-01-01' AND '2022-01-01';
```

Use generate script to drop table.

## Query Optimization

Good to know:

1. Join: No indexes on join columns in tables will result in hash match(worst case). If these tables are sorted on join columns, merge match is used(efficient)

Nested Loop Join: Primitive algorithm: Nested loop join is a join operation in database systems where two tables are joined using nested loops. It involves iterating through each row of one table and comparing it with each row of the other table to find matching rows.

Always try to reduce rows using where clause to improve efficiency.

*2. SARG: Where clause in which there is index on field used in where. Makes query efficient.*

3. Avoid sorting if not necessary.

4. Plan Guide: *A plan guide is created when a procedure runs for the first time and is reused each time. This is not good if plan not optimized for all cases. Instead when creating procedure:*

Create procedure proc WITH RECOMPILE

AS …..

5. Hints: In Microsoft SQL Server (MSSQL), query hints provide instructions to the query optimizer on how to execute a specific query. Hints can influence the query plan chosen by the optimizer, allowing you to fine-tune query performance or enforce specific behaviors. *Don't use unless you're very sure.*
Ex: E left loop join T – or left hash join/left merge

Select * from tblEmp(noLock)

*6. Statistics I/O:*

In Microsoft SQL Server (MSSQL), the "SET STATISTICS IO" option allows you to display the detailed I/O (Input/Output) statistics for a query execution. It provides information about the number of logical and physical I/O operations performed by the query, which can be useful for performance tuning and understanding the query's impact on disk I/O. Here's how you can use the "SET STATISTICS IO" option:

SET STATISTICS IO ON;

SELECT * FROM TableName;

To disable: SET STATISTICS IO OFF;

1. Interpretation of I/O Statistics:
    - After executing the query with "SET STATISTICS IO ON," SQL Server will display the I/O statistics in the Messages tab or output window.
    - The statistics typically include three values: logical reads, physical reads, and read-ahead reads.
    - Logical Reads:
        - Represents the number of pages read from the buffer cache.
        - Logical reads occur when a data page is already in memory and can be read directly without accessing the disk.
    - Physical Reads:
        - Indicates the number of pages read from disk into the buffer cache.
        - Physical reads occur when a data page is not in memory and needs to be retrieved from disk.
    - Read-Ahead Reads:
        - Refers to the number of pages read into the buffer cache in anticipation of future requests.
        - Read-ahead reads help to minimize disk I/O by proactively bringing data pages into memory before they are requested.
2. Analyzing the I/O Statistics:
    - By examining the I/O statistics, you can evaluate the efficiency of your query and identify potential performance bottlenecks.
    - Higher numbers of physical reads suggest that the query is accessing a significant amount of data from disk, which can impact performance.
    - Lower logical reads indicate that the query is efficiently using the buffer cache and retrieving data from memory.

### *7. SHOWPLAN ALL*

In Microsoft SQL Server (MSSQL), the "SHOWPLAN" option allows you to view the execution plan of a query without actually executing it. The execution plan provides valuable information about how SQL Server intends to execute the query, including the chosen join methods, index usage, and data access operations. Here's how you can use the "SHOWPLAN" option to view query execution plans:

```
SET SHOWPLAN_ALL ON;

GO

SELECT * FROM TableName;
```

To turn off:

```
SET SHOWPLAN_ALL OFF;
```

In addition to "SHOWPLAN_ALL," there are other SHOWPLAN options available, such as "SHOWPLAN_TEXT" and "SHOWPLAN_XML," which provide different formats for displaying the execution plan.

### *8. STATISTICS TIME*

In Microsoft SQL Server, you can use the SET STATISTICS TIME ON statement to enable the display of the execution time statistics for each query in the query result. When this option is turned on, SQL Server provides information about the amount of CPU time (in milliseconds) used to execute the query and the elapsed time (in milliseconds) taken for the query to complete.

```
SET STATISTICS TIME ON;

-- Your query goes here

SET STATISTICS TIME OFF;
```

## MSSQL Management Tools Overview

Microsoft SQL Server Management Studio (SSMS) is the primary management tool for Microsoft SQL Server, which is a relational database management system (RDBMS) developed by Microsoft.

1. SSMS is a powerful and comprehensive tool that enables database administrators to manage and administer SQL Serve
2. r instances, create and modify databases and database objects, write and execute queries, and perform various other tasks.
3. SQL Server Configuration Manager: A tool that enables administrators to configure SQL Server services, protocols, and network settings.
4. SQL Server Data Tools (SSDT): A toolset that enables developers to design, develop, and deploy SQL Server databases and database objects(BI Services).

5. SQL Server Reporting Services (SSRS): A tool that enables users to create, deploy, and manage reports in SQL Server.
6. SQL Server Integration Services (SSIS): A tool that enables administrators to create, deploy, and manage data integration and ETL solutions.
7. The SQL Server Database Engine Tuning Advisor (DTA) is a tool provided by Microsoft that helps database administrators and developers improve the performance of SQL Server databases. The tool analyzes a workload of SQL statements, including queries, stored procedures, and batches, and makes recommendations on how to improve database performance.
8. Import Export Tool: To import data from file to db or export from db to a file.

## SQL Server Profiler

SQL Server Profiler is a tool provided by Microsoft that enables administrators and developers to capture and analyze events that occur in SQL Server. It allows users to monitor the activity of SQL Server, including queries, stored procedures, and other operations, and analyze that activity to diagnose performance issues, identify security vulnerabilities, and troubleshoot errors.

Some of the key features of SQL Server Profiler include:

1. Event Tracing: SQL Server Profiler enables users to trace events that occur in SQL Server and capture information about those events, including the SQL statements executed, the duration of each operation, and the resources used by each event.
2. Customizable Traces: Users can customize the traces by selecting which events to capture, defining filters to limit the scope of the trace, and specifying output options such as file format and destination.
3. Real-Time Monitoring: SQL Server Profiler can also be used for real-time monitoring of SQL Server activity, allowing administrators to track server activity as it happens and respond quickly to performance issues or errors.
4. Integration with other SQL Server tools: SQL Server Profiler can be used in conjunction with other SQL Server tools, such as SQL Server Management Studio and the Database Engine Tuning Advisor, to perform more comprehensive analysis and optimization of database performance.

Run Profiler on different instance as it adds load on server and only use on objects where tracing is needed.

## SSMS

Microsoft SQL Server Management Studio (SSMS) is a comprehensive tool for managing and administering SQL Server instances, databases, and database objects. Can connect to local and remote instances of database engine, analysis service, etc. Here are some key features of SSMS:

1. Object Explorer: Object Explorer is a graphical user interface that allows users to view and manage the database objects, including tables, views, stored procedures, and user-defined functions. Users can perform common tasks such as creating, modifying, and deleting objects, and view object properties.

    - Object Explorer Details Pane: This pane displays a list of objects and their properties. You can use this pane to view and edit properties of objects such as tables, columns, indexes, and stored procedures.
    - Object Explorer Tree View: This pane displays a hierarchical view of the SQL Server instance and its components. You can expand and collapse nodes to navigate to different levels of the hierarchy, and you can right-click on objects to access context menus that allow you to perform various tasks, such as creating or deleting objects.
    - Object Explorer Search: This pane allows you to search for objects in the SQL Server instance based on specific criteria such as object name, type, or schema. You can use this feature to quickly find specific objects within large databases.
    - Object Explorer Details Tab: This tab displays detailed information about the selected object in the Object Explorer tree view. This includes properties, permissions, dependencies, and other metadata about the object.
    - Registered Servers: This pane allows you to manage multiple SQL Server instances and group them into folders for easy access. You can use this feature to quickly connect to different servers and manage objects across multiple instances.

2. Query Editor: Query Editor is a tool that enables users to write and execute SQL queries against SQL Server databases. It includes features such as syntax highlighting, code snippets, and query execution plan visualization.

    Right click query and you'll get options like tracing, get data as text, execution plans, profiler and many other options.

3. Activity Monitor: In SQL Server Management Studio (SSMS), the "Activity Monitor" is a tool that provides a real-time view of SQL Server system activity and resource usage. It allows you to monitor processes, locks, sessions, and queries, and to identify performance bottlenecks and resource contention issues.

4. To open the Activity Monitor in SSMS, you can right-click on the SQL Server instance in the Object Explorer and select "Activity Monitor" from the context menu. The Activity Monitor window will open, displaying several panes that provide different views of the SQL Server activity.

    The main components of the Activity Monitor in SSMS are:

    - Processes Pane: This pane displays a list of processes that are currently running on the SQL Server instance, along with their associated CPU usage, memory usage, and input/output (I/O) statistics. You can sort and filter the processes based on various criteria to identify processes that are consuming the most resources.

- Resource Waits Pane: This pane displays a list of the types of resources that are currently being waited on by processes, such as CPU, memory, disk, or network. You can use this pane to identify resource contention issues and to see which processes are waiting for which resources.
- Data File I/O Pane: This pane displays a list of the data files that are currently being accessed by SQL Server, along with their associated I/O statistics. You can use this pane to identify I/O bottlenecks and to see which databases and files are consuming the most I/O resources.
- Recent Expensive Queries Pane: This pane displays a list of the queries that have consumed the most resources over a specified period of time. You can use this pane to identify queries that are causing performance issues and to optimize them for better performance.

5. SQL Server Agent: SQL Server Agent is a tool that enables administrators to automate administrative tasks such as backups, maintenance, and job scheduling. Users can create jobs, schedules, alerts, and operators to automate routine tasks and notifications.

6. Database Engine Tuning Advisor: Database Engine Tuning Advisor is a tool that helps database administrators and developers improve the performance of SQL Server databases by analyzing a workload of SQL statements and making recommendations for improving database performance.

7. Profiler: Profiler is a tool that allows administrators and developers to capture and analyze SQL Server activity to diagnose performance issues. Users can capture a SQL Server Profiler trace, which is a recording of all SQL statements executed against the database, and analyze that trace to identify performance bottlenecks and optimize database performance.

8. Backup and Restore: SSMS includes features for backing up and restoring SQL Server databases. Users can create full or differential backups, schedule backups, and restore databases to a point in time.

9. Security: SSMS includes features for managing SQL Server security, including creating and managing logins, roles, and permissions. Users can also audit server activity and view security logs.

10. Server Properties: SSMS provides a Server Properties window that allows users to view and modify server-level settings such as memory allocation, processor affinity, network settings, and security settings.
    Open SQL Server Management Studio and connect to the SQL Server instance you want to manage.

    In Object Explorer, expand the server node and right-click on the server name. Select "Properties" from the context menu.

    The Server Properties dialog box will open, displaying the General tab by default. This tab displays general information about the server, including the product version, operating system version, and server name. You can also specify the default language for new logins, as well as the maximum number of concurrent connections allowed.

Click on the "Memory" tab to configure memory-related settings. Here, you can specify the amount of memory allocated to SQL Server, as well as other memory-related settings such as maximum server memory and minimum memory per query. You can also enable "Lock Pages in Memory" to prevent the operating system from paging SQL Server memory to disk.

Click on the "Processors" tab to configure processor-related settings. Here, you can specify processor affinity settings for SQL Server, which can help improve performance by dedicating specific processors to SQL Server. You can also enable "Boost SQL Server priority" to give SQL Server a higher priority than other processes running on the server.

Click on the "Security" tab to configure security-related settings. Here, you can configure various security-related settings, such as authentication mode, login auditing, and server-level permissions. You can also specify the default database for new logins, and enable "Cross-database ownership chaining" to allow users to access objects in other databases without explicitly granting permissions.

Click on the "Advanced" tab to configure other advanced settings. Here, you can specify various settings related to network protocols, startup options, and other miscellaneous settings.

After configuring the desired settings, click "OK" to save the changes.

Note that some server properties may require you to restart the SQL Server instance for the changes to take effect. Be sure to read the documentation carefully and follow any required steps to ensure that your server is configured correctly.

11. In SQL Server Management Studio (SSMS), the "Parse" option is used to validate the syntax of a Transact-SQL (T-SQL) script without actually executing it. This can be useful for catching syntax errors before executing a script, especially if the script is complex or contains a large number of statements. This is a tick option besides debug in ssms.

12. Query Execution Plan: In SQL Server Management Studio (SSMS), the "Query Execution Plan" is a visual representation of how SQL Server will execute a specific query. It shows the order of operations, the data access methods, and the join algorithms that SQL Server will use to retrieve and manipulate the data. The query execution plan can be generated by SSMS, and it is displayed as a graphical representation that can be used to identify performance bottlenecks and to optimize queries.

    To generate a query execution plan in SSMS, you can follow these steps:

    Open a new query window and type or paste the T-SQL query you want to analyze.

    Click the "Include Actual Execution Plan" button on the toolbar, or select "Include Actual Execution Plan" from the Query menu.

Execute the query by clicking the "Execute" button on the toolbar or selecting "Execute" from the Query menu.

The query execution plan will be displayed in a separate tab in the query editor window. You can click on various parts of the execution plan to see details about each step, such as the estimated number of rows, the cost of the operation, and the physical and logical reads.

The query execution plan is an important tool for optimizing queries, because it can help identify performance issues such as full table scans, inefficient joins, and expensive sorts or aggregates. By analyzing the execution plan, you can make changes to the query, such as adding or removing indexes, rewriting the query to use more efficient join algorithms, or optimizing the data retrieval strategy. This can lead to significant improvements in query performance, which can have a big impact on application performance as a whole.

We have **estimated execution plan**(gives plan without running query based on past knowledge) and **actual execution plan**(actually runs query)

13. sp_who2 is a built-in system stored procedure in SQL Server that provides information about the current SQL Server sessions and processes. It returns a result set that includes information such as the session ID, login name, database being used, status, command being executed, and more.
    - The sp_who2 stored procedure is often used for troubleshooting and monitoring SQL Server activity, particularly when trying to diagnose blocking or other performance issues. By running sp_who2, you can get a quick overview of the current sessions and their status, which can help you identify any sessions that are blocking other sessions or using excessive resources.
    - To execute sp_who2, you can open a new query window in SQL Server Management Studio (SSMS) and type sp_who2 into the query editor. Then, click on the "Execute" button to run the query. The result set will be displayed in the query results window, showing information about each session.

14. SQL Server logs in SSMS (SQL Server Management Studio) are used to record events and messages that occur within SQL Server, such as errors, warnings, and informational messages. The logs can be used for troubleshooting, auditing, and performance tuning purposes. There are several types of logs available in SQL Server, including:
    - SQL Server Error Logs: These logs contain information about errors and warnings that occur within SQL Server, such as failed login attempts, backup and restore errors, and deadlock errors. By default, SQL Server error logs are stored in the LOG folder of the SQL Server instance directory.
    - SQL Server Agent Error Logs: These logs contain information about errors and warnings that occur within SQL Server Agent, such as job failures and step failures. By default, SQL Server Agent error logs are stored in the LOG folder of the SQL Server instance directory.
    - Windows Event Logs: These logs contain information about SQL Server events that are logged in the Windows event log, such as service startup and shutdown events, and failed login attempts. Windows Event Logs are stored in the Windows Event Viewer.
    - To view SQL Server logs in SSMS, you can use the "SQL Server Logs" node in the Object Explorer. Right-click on the "SQL Server Logs" node and select "View SQL Server Log"

from the context menu to open the Log File Viewer. From here, you can view the contents of the logs, search for specific messages, filter the messages based on severity or other criteria, and save the logs to a file.

Overall, SSMS is a powerful tool that provides a wide range of features for managing and administering SQL Server instances and databases. With its comprehensive set of features, it is an essential tool for database administrators and developers working with SQL Server. Lots o options, keep exploring.

## SQL Server Configuration Manager

1. SQL Server Configuration Manager is a tool that is used to manage and configure the various components of SQL Server.
2. The tool provides several options for configuring SQL Server services, protocols, and network connectivity.
3. Some of the options available in SQL Server Configuration Manager include:

   a. SQL Server Services: This option allows you to manage the SQL Server services that are installed on your system. You can start, stop, or restart services, and configure their properties.

   b. SQL Server Network Configuration: This option allows you to configure the protocols that SQL Server uses to communicate over the network. You can enable or disable protocols, configure their properties, and specify the ports on which SQL Server listens.

   c. SQL Native Client Configuration: This option allows you to configure the SQL Native Client driver, which is used by applications to connect to SQL Server. You can configure settings such as connection timeouts, default language, and encryption options.

   d. SQL Server Native Client 11.0 Configuration: This option allows you to configure the SQL Server Native Client 11.0 driver, which is used by applications to connect to SQL Server. You can configure settings such as connection timeouts, default language, and encryption options.

   e. SQL Server Configuration Manager: This option allows you to configure the SQL Server Configuration Manager tool itself. You can set options such as the default view, logging levels, and error reporting.
4. SQL Server Configuration Manager is an important tool for managing and configuring SQL Server components. It is important to understand the available options and how they can be used to optimize the performance and security of your SQL Server installation.

## Jobs

In Microsoft SQL Server (MSSQL), jobs are a key component of the SQL Server Agent, which is responsible for executing scheduled tasks and monitoring the server for events. Here's an example syntax of creating a job in MSSQL:

```
USE msdb
GO

EXEC dbo.sp_add_job
   @job_name = N'MyJob',
   @enabled = 1,
   @description = N'This is a sample job.',
   @owner_login_name = N'sa'

EXEC sp_add_jobstep
   @job_name = N'MyJob',
   @step_name = N'ExecuteMyScript',
   @subsystem = N'TSQL',
   @command = N'SELECT GETDATE()',
   @retry_attempts = 5,
   @retry_interval = 5

EXEC sp_add_schedule
   @schedule_name = N'EveryHour',
   @freq_type = 4,
   @freq_interval = 1

EXEC sp_attach_schedule
   @job_name = N'MyJob',
   @schedule_name = N'EveryHour'

EXEC dbo.sp_add_jobserver
   @job_name = N'MyJob',
   @server_name = N'(local)'
```

In this example, we're creating a new job called "MyJob" that executes a simple T-SQL command of getting the current date and time. The job is scheduled to run every hour, and the job owner is set to "sa".

The steps involved in creating the job are as follows:

1. Use the msdb database.
2. Create the job using the sp_add_job stored procedure, specifying the job name, description, owner, and enabling it.
3. Create a job step using sp_add_jobstep, specifying the step name, subsystem (T-SQL), and the command to execute.
4. Create a schedule using sp_add_schedule, specifying the schedule name, frequency type (4 = daily), and interval (1 = every day).
5. Attach the schedule to the job using sp_attach_schedule.
6. Add the job to the SQL Server Agent using sp_add_jobserver.

Once the job is created, it will execute on the specified schedule and execute the T-SQL command, in this case getting the current date and time.

This is just a simple example, but MSSQL jobs can be much more complex and powerful, allowing database administrators to automate many tasks and streamline their workflows.

Creating a job in SQL Server Management Studio (SSMS) using the UI is a straightforward process. Here are the steps to create a job using the UI in SSMS:

1. Open SQL Server Management Studio and connect to the SQL Server instance you want to create the job on.
2. In the Object Explorer window, expand the SQL Server Agent node and right-click on the Jobs folder.
3. Select "New Job" from the context menu. This will open the New Job dialog box.
4. In the General tab of the New Job dialog box, enter a name and description for the job.
5. In the Steps tab, click "New" to create a new step. In the New Job Step dialog box, enter a name for the step and select the type of command to execute (SQL Server Integration Services package, T-SQL script, or Operating System command).
6. Enter the command or script to execute in the Command box. If necessary, you can specify additional options such as advanced error handling, retry attempts, or output files.
7. In the Schedules tab, click "New" to create a new schedule. In the New Job Schedule dialog box, specify the frequency and duration of the job execution.
8. In the Alerts tab, you can select one or more alerts to be triggered when the job fails or completes successfully.
9. In the Notifications tab, you can specify one or more email recipients to be notified when the job fails or completes successfully.
10. Click OK to save the job and close the New Job dialog box.

Once the job is created, you can manually execute it, modify its properties, or delete it using the SSMS interface. Additionally, you can use T-SQL scripts to perform the same actions, giving you more flexibility and control over the job creation process.

## Alerts

In SQL Server, alerts are a way to notify database administrators and other personnel when certain events or conditions occur within the system. Alerts can be set up to monitor a wide range of events, including system errors, performance issues, and security breaches. Here are the steps to create an alert in SQL Server:

1. Open SQL Server Management Studio and connect to the SQL Server instance you want to create the alert on.

2. Right-click on the SQL Server Agent node in Object Explorer and select "New Alert."
3. In the New Alert dialog box, give the alert a name and a description.
4. Select the type of alert you want to create from the "Type" drop-down list. You can choose from several predefined alert types, such as "SQL Server performance condition alert" or "SQL Server event alert."
5. Configure the alert conditions by selecting the appropriate values for the alert type you have chosen. For example, if you are creating a performance condition alert, you would specify the performance counter and the threshold value that should trigger the alert.
6. Choose the actions that should be taken when the alert is triggered. You can configure the alert to send an email, write to the Windows event log, or run a specific job.
7. Click "OK" to create the alert.

Once the alert has been created, it will be active and will monitor the conditions you have specified. When the conditions are met, the alert will trigger the configured action(s) and notify the appropriate personnel. You can also modify or delete alerts at any time using SQL Server Management Studio.

## Database Mail

In SQL Server, you can use Database Mail to send email messages from the SQL Server database engine. This feature allows you to send notifications, alerts, and reports via email to administrators, developers, and other users. Here are the steps to set up Database Mail in SQL Server:

1. Open SQL Server Management Studio and connect to the SQL Server instance you want to set up Database Mail on.
2. In Object Explorer, right-click on the "Database Mail" node and select "Configure Database Mail."
3. In the "Database Mail Configuration Wizard," choose the option to "Set up Database Mail by performing the following tasks" and click "Next."
4. Enter a name for the mail profile and click "Add" to configure the email server.
5. In the "New Database Mail Account" dialog box, enter the email account information, including the email address, display name, mail server, and port.
6. Choose the email authentication method, such as "Basic Authentication," and enter the necessary credentials.
7. Select the default public profile for the email account and click "Next."
8. Test the email settings by sending a test email.
9. Once the test email is successful, click "Finish" to complete the Database Mail setup.

After setting up Database Mail, you can use the sp_send_dbmail stored procedure to send email messages from SQL Server. This stored procedure accepts parameters for the email recipients, subject, body, and attachment, if applicable. You can also include SQL query results in the email message by using the @query parameter.

For example, to send an email message from SQL Server, you could use the following code:

```
EXEC msdb.dbo.sp_send_dbmail
@recipients='email@example.com',
@subject='Test Email',
@body='This is a test email from SQL Server.',
@query='SELECT * FROM [dbo].[MyTable]'
```

This will send an email message to the specified recipient with the query results from the MyTable table included in the email body.

# Working of MSSQL

## Pages and Extents

### Pages

Pages are the basic unit of storage in MSSQL. A page is an 8 KB unit of data that can hold a single row or multiple rows of a table or index. Each page has a header that contains information about the page type, data stored, and other metadata.

### MDF, NDF Files

In Microsoft SQL Server, the MDF (Master Data File) and NDF (Secondary Data File) are file types used to store the data and objects of a database. Let's discuss how these files relate to pages in MSSQL:

1. MDF File:
    - The MDF file is the primary data file of a SQL Server database. It contains the system tables, user-defined tables, indexes, stored procedures, and other database objects.
    - The data in the MDF file is stored in 8 KB pages. These pages hold the actual rows of the tables or indexes.
    - The MDF file consists of a collection of pages that are organized in a hierarchical structure, forming a B-tree-like structure for efficient data retrieval and storage.
2. NDF File:
    - The NDF file is an optional secondary data file that can be added to a SQL Server database. It provides additional storage space for the database when the size of the MDF file is not sufficient or when you want to segregate specific objects onto different filegroups.
    - Like the MDF file, the NDF file also stores data in 8 KB pages. It follows the same page structure and architecture as the MDF file.
    - Multiple NDF files can be added to a database, allowing for parallel I/O operations and distributing the database workload across multiple files and filegroups.
3. Pages in MDF and NDF Files:
    - Both the MDF and NDF files consist of pages that store the actual data. Each page is an 8 KB unit of storage.

- The pages in the MDF and NDF files hold the rows of tables, indexes, and other objects. These pages are allocated and managed by the SQL Server storage engine.
- Pages in the MDF and NDF files are organized into extents (contiguous groups of eight pages) for efficient storage and retrieval.

It's important to note that while the MDF file is mandatory for a SQL Server database, the NDF file is optional and can be added based on the specific needs of your database and storage requirements.

Understanding the organization of data into pages within the MDF and NDF files can help in optimizing database performance, managing storage, and troubleshooting issues related to data retrieval and storage in SQL Server.

Note: LDF files store logs and they don't follow page concept.

## Data and Index Pages

In Microsoft SQL Server, data and index pages are two types of pages used to store different types of information within a database. Let's explore data pages and index pages in more detail:

1. Data Pages:
    - Data pages store the actual data of tables and other heap-organized objects in a SQL Server database.
    - Each data page is an 8 KB unit of storage and holds one or more rows of a table. The number of rows stored on a data page depends on the size of the rows and the amount of free space available on the page.
    - When a table grows and requires more storage, additional data pages are allocated to accommodate the new rows.
    - Data pages do not have any specific order, and the data within them is typically unsorted. The order of rows in a table is not guaranteed unless an explicit ORDER BY clause is used during queries.

2. Index Pages:
    - Index pages store the index entries and related information for efficient data retrieval in SQL Server.
    - Each index has its own set of index pages, separate from the data pages of the corresponding table.
    - Index pages are organized in a B-tree-like structure, which allows for efficient searching, sorting, and range-based queries.
    - Index pages contain key values and pointers to the data pages where the corresponding rows are stored. This enables quick lookup and retrieval of data based on the index key.
    - Depending on the size of the index and the configuration of the server, index pages can be allocated dynamically as needed.

Data and index pages work together to provide efficient data storage and retrieval in SQL Server. Data pages hold the actual data rows, while index pages facilitate faster data access by providing a structured mechanism for locating specific rows based on the index key.

It's important to design and maintain appropriate indexes in your SQL Server database to optimize query performance. Proper index selection and maintenance can significantly enhance data retrieval

speed, particularly for frequently queried columns or complex queries involving joins, sorting, or filtering operations.

## System Pages

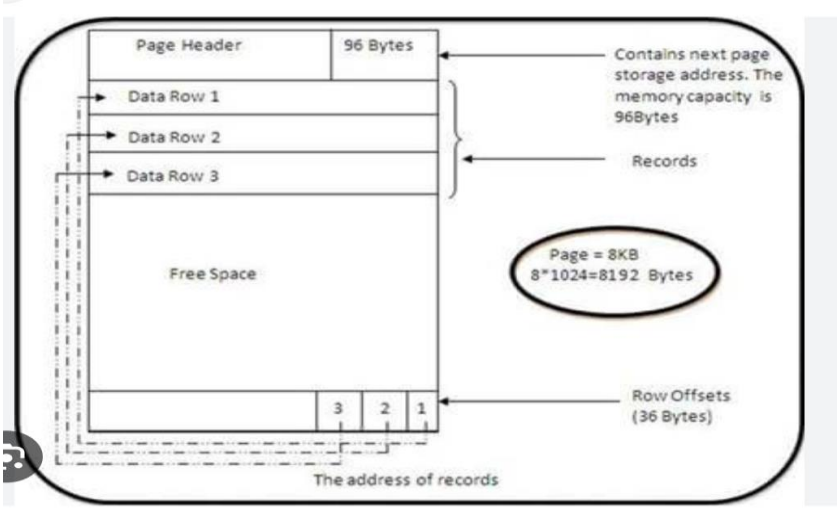
In Microsoft SQL Server, system pages are special types of pages that store critical information about the database and its internal structures. These pages are part of the system databases and play a vital role in the functioning of SQL Server. Here are some key types of system pages:

1. Boot Page (Page 0):
    - The boot page, also known as page 0, is the first page of a database file. It contains essential information about the database, such as the file header and metadata necessary for database recovery.
    - The boot page provides details about the database file size, version, compatibility level, and file layout.
2. File Header Page:
    - Each database file, including data files (.mdf), log files (.ldf), and secondary data files (.ndf), has a file header page.
    - The file header page stores information specific to the file, such as the file type, file ID, file size, and database ID. It also includes critical metadata related to the file, such as the page size, checksum information, and filegroup details.
3. Allocation Bitmap Pages:
    - Allocation bitmap pages track the allocation status of data pages within a file or filegroup.
    - These pages maintain a record of allocated and unallocated pages to efficiently manage and allocate storage space within the database.
4. Differential Bitmap Pages:
    - Differential bitmap pages are used in database backup and restore operations to track changes since the last full database backup.
    - These pages store information about the extents that have been modified since the last full backup. This allows for faster backup and restore operations by identifying the extents that need to be included in a differential backup.
5. Global Allocation Map (GAM) and Shared Global Allocation Map (SGAM):
    - GAM and SGAM pages are used to manage the allocation of extents within a database.
    - The GAM page tracks the allocation status of each extent in the database file, indicating whether an extent is allocated or unallocated.
    - The SGAM page is responsible for tracking the shared allocation of extents, indicating whether an extent is partially allocated or not.

These are just a few examples of system pages in SQL Server. The system pages provide critical metadata and information necessary for the proper functioning of the database engine, ensuring data integrity, storage allocation, and recovery operations.

## Page Architecture

- Header: The header section of a page contains metadata about the page, including information about the page type, page ID, and allocation status.
- Data Section: The data section of a page contains the actual data for the table or index. Depending on the page type, the data section may contain a combination of rows, index nodes, or other data structures.
- Row Offset Table: The row offset table is used to locate individual rows within the data section of the page. It contains a list of offsets that point to the start of each row.




The page architecture in Microsoft SQL Server refers to the structure and organization of individual database pages. Each page is an 8 KB unit of storage that stores data and metadata within a SQL Server database. Here is an overview of the page architecture in SQL Server:

1. Page Header:
    - The page header is located at the beginning of each page and contains metadata and control information about the page.
    - The page header includes details such as the page type, page ID, file ID, and allocation unit ID.
    - It also stores information about the page's free space, checksum for data integrity checks, and timestamp for tracking modifications.
2. Data Section:
    - The data section of a page holds the actual data, including rows of tables, index nodes, and other objects.
    - The data section is divided into multiple data structures, depending on the page type.

- For data pages, the data section contains rows of a table or heap-organized object. For index pages, it holds index key values and pointers to the corresponding data pages.

3. Row Offset Table (for Data Pages):
    - In data pages, there is a row offset table located immediately after the data section.
    - The row offset table contains the offset values that indicate the starting position of each row within the data section.
    - By using the row offset table, SQL Server can quickly locate and access individual rows within the data section.
4. Index Structures (for Index Pages):
    - Index pages have a specific structure based on the type of index (e.g., clustered index or non-clustered index).
    - The index structures are organized in a B-tree-like structure, allowing for efficient data retrieval and index maintenance.
    - Index pages contain key values and pointers to the corresponding data pages or other index pages, depending on the index structure.
5. Page Footer:
    - The page footer, also known as the page trailer, is located at the end of each page.
    - The page footer stores additional metadata, such as the offset to the beginning of the free space and the count of free bytes on the page.
    - It helps SQL Server manage the allocation and utilization of space within the page efficiently.

The page architecture in SQL Server is designed to optimize data storage, retrieval, and management. Understanding the structure and components of database pages can be beneficial for performance tuning, troubleshooting, and understanding how data is organized and accessed within the database.

## Extents

In SQL Server, an extent is a collection of eight physically contiguous pages, with each page containing 8 KB of data. Extents are used to store data and index pages in SQL Server.

There are two types of extents in SQL Server:

1. Uniform extents: A uniform extent is one that is entirely allocated to a single object or index. In other words, all eight pages of a uniform extent are used by a single object or index. Most commonly used in dbs.

2. Mixed extents: A mixed extent is one that is shared by multiple objects or indexes. A mixed extent contains pages that are allocated to different objects or indexes. Each mixed extent contains a maximum of eight pages, with each page being allocated to a different object or index. Mixed extents are used to conserve disk space by reducing the amount of unused space that is allocated to objects and indexes.

Can manually set up this extent type for a database.

## Finding Page details of a table

To find page details of a table in Microsoft SQL Server, you can use the following steps:

1. Identify the table: Determine the name of the table for which you want to find the page details.
2. Enable trace flag 3604: Open a new query window in SQL Server Management Studio (SSMS) and run the following command to enable trace flag 3604, which redirects the DBCC PAGE command output to the query window:

```
DBCC TRACEON (3604);
```

3. Get the object ID: Execute the following query to obtain the object ID of the table:

```
SELECT OBJECT_ID('schema_name.table_name') AS ObjectID;
```

4. Use DBCC PAGE command: Run the DBCC PAGE command with the obtained object ID, file ID, and page number to view the page details. The syntax is as follows:

```
DBCC PAGE ('database_name', file_id, page_number, print_option);
```

Replace 'database_name' with the name of the database containing the table, 'file_id' with the ID of the file in which the table is stored (can be obtained from sys.database_files catalog view), 'page_number' with the desired page number, and 'print_option' with 0, 1, 2, or 3 depending on the level of details you want to display.

For example:

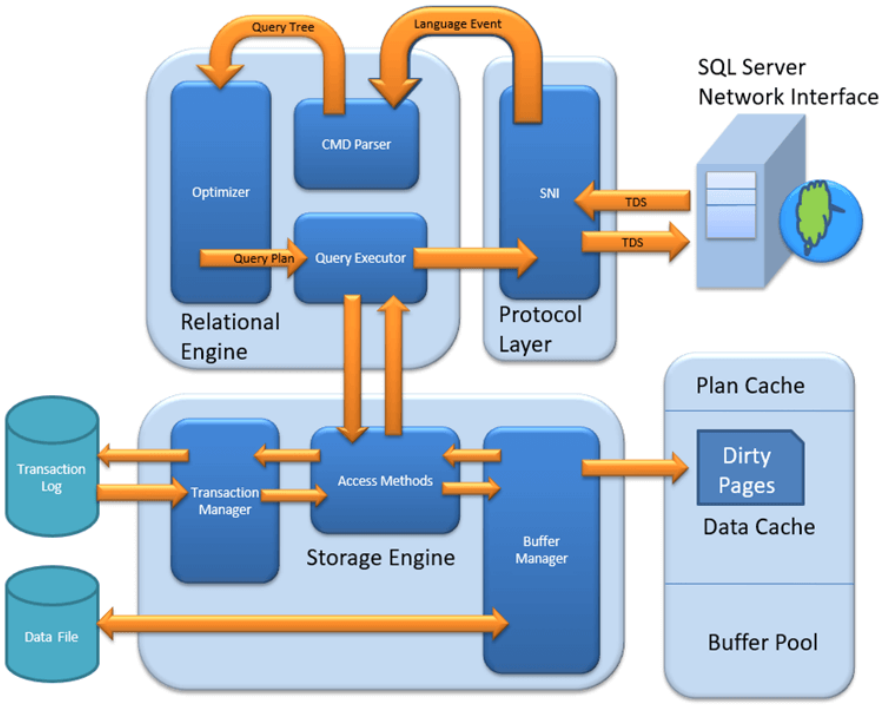
```
DBCC PAGE ('YourDatabaseName', 1, 123, 3);
```

This will display the page details for page number 123 in file ID 1 of the specified database.

Remember that using DBCC PAGE command requires advanced knowledge of SQL Server internals, and it should be used cautiously by experienced database professionals for troubleshooting and analysis purposes.

## MSSQL Architecture(learnt from Guru99)

MS SQL Server is a client-server architecture. MS SQL Server process starts with the client application sending a request. The SQL Server accepts, processes and replies to the request with processed data.
Storage Engine




## Protocol Layer – SNI

MS SQL SERVER PROTOCOL LAYER supports 3 Type of Client Server Architecture.

### *Shared Memory*

**CLIENT** and **MS SQL** server run on the same machine. Both can communicate via Shared Memory protocol.

For Connection to Local DB – In SQL Management Studio, “Server Name” Option could be

“.” or “localhost” or “127.0.0.1” or “Machine\Instance”

### *TCP/IP*

MS SQL SERVER provides the capability to interact via TCP/IP protocol, where CLIENT and MS SQL Server are remote to each other and installed on a separate machine.

**Notes from the desk of Configuration/installation:**

- In SQL Management Studio – For Connection via TCP\IP, “Server Name” Option has to be “Machine\Instance of the server.”
- SQL server uses port 1433 in TCP/IP.

### *Named Pipes*

the CLIENT and **MS SQL SERVER** are in connection via **LAN**.

**Notes from the desk of Configuration/installation:**

- **For Connection via Named Pipe.** This option is disabled by default and needs to be enabled by the SQL Configuration Manager.

### *What is TDS?*

- TDS stands for Tabular Data Stream.
- All 3 protocols use TDS packets. TDS is encapsulated in Network packets. This enables data transfer from the client machine to the server machine.

- TDS was first developed by Sybase and is now Owned by Microsoft

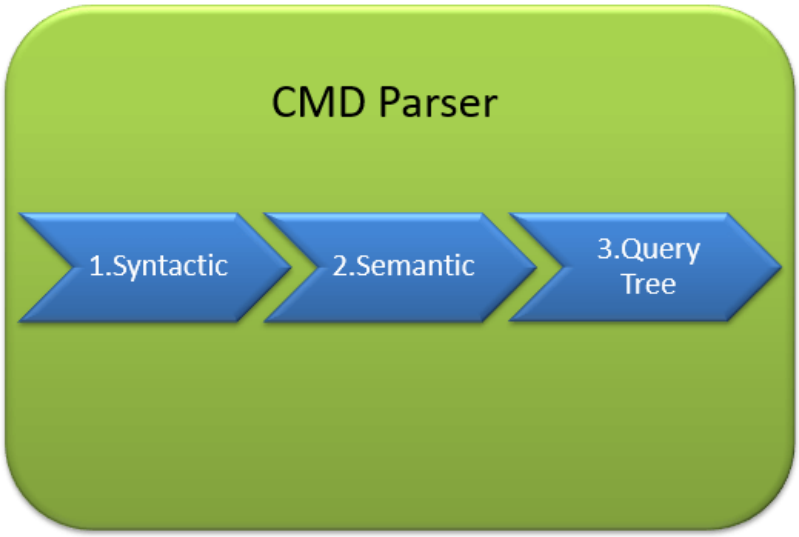
## Relational Engine

The Relational Engine is also known as the Query Processor. It is responsible for the execution of user queries by requesting data from the storage engine and processing the results that are

There are **3 major components** of the Relational Engine:

### *CMD Parser*

Data once received from Protocol Layer is then passed to Relational Engine. **"CMD Parser"** is the first component of Relational Engine to receive the Query data. The principal job of CMD Parser is to check the query for **Syntactic and Semantic error.**




### Syntactic check:

- Like every other Programming language, MS SQL also has the predefined set of Keywords. Also, SQL Server has its own grammar which SQL server understands.
- SELECT, INSERT, UPDATE, and many others belong to MS SQL predefined Keyword lists.
- CMD Parser does syntactic check. If users' input does not follow these language syntax or grammar rules, it **returns an error.**

SELECT * from <TABLE_NAME>;

Now, to get the perception of what syntactic does, say if the user runs the basic query as below:

SELECR * from <TABLE_NAME>

Note that instead of 'SELECT' user typed "SELECR."

**Result:** THE CMD Parser will parse this statement and will throw the error message. As "SELECR" does not follow the predefined keyword name and grammar. Here CMD Parser was expecting "SELECT."

### Semantic check

- This is performed by **Normalizer**.
- In its simplest form, it checks whether Column name, Table name being queried exist in Schema. And if it exists, bind it to Query. This is also known as **Binding**.

- Complexity increases when user queries contain VIEW. Normalizer performs the replacement with the internally stored view definition and much more.

Let's understand this with help of below example –

SELECT * from USER_ID

**Result:** THE CMD Parser will parse this statement for Semantic check. The parser will throw an error message as Normalizer will not find the requested table (USER_ID) as it does not exist.

### Create Query Tree

- This step generates different execution tree in which query can be run.
- Note that, all the different trees have the same desired output.

### *Optimizer*

The work of the optimizer is to create an execution plan for the user's query. This is the plan that will determine how the user query will be executed.

Note that not all queries are optimized. Optimization is done for DML (Data Modification Language) commands like SELECT, INSERT, DELETE, and UPDATE. Such queries are first marked then send to the optimizer. DDL commands like CREATE and ALTER are not optimized, but they are instead compiled into an internal form. The query cost is calculated based on factors like CPU usage, Memory usage, and Input/ Output needs.

Optimizer's role is to find the **cheapest, not the best, cost-effective execution plan.**

Before we Jump into more technical detail of Optimizer consider below real-life example:

Similarly, MS **SQL Optimizer works on inbuilt exhaustive/heuristic algorithms. The goal is to minimize query run time.** All the Optimizer algorithms are **propriety of Microsoft and a secret.** Although**,** below are the high-level steps performed by MS SQL Optimizer. Searches of Optimization follows three phases:

### Phase 0: Search for Trivial Plan:

- This is also known as **Pre-optimization stage**.
- For some cases, there could be only one practical, workable plan, known as a trivial plan. There is no need for creating an optimized plan. The reason is, searching more would result in finding the same run time execution plan. That too with the extra cost of Searching for optimized Plan which was not required at all.
- If no Trivial plan found, then 1$^{st}$ Phase starts.

### Phase 1: Search for Transaction processing plans

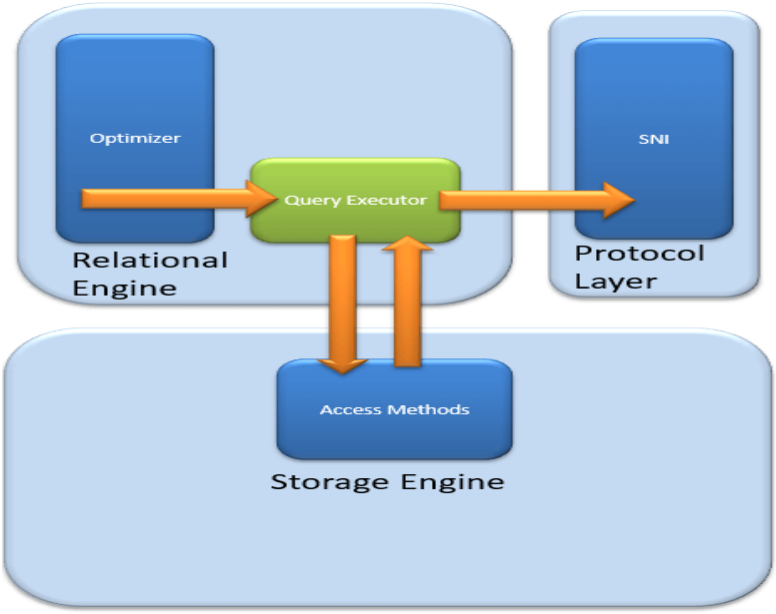
- This includes the search for **Simple and Complex Plan**.
- Simple Plan Search: Past Data of column and Index involved in Query, will be used for Statistical Analysis. This usually consists but not restricted to one Index Per table.
- Still, if the simple plan is not found, then more complex Plan is searched. It involves Multiple Index per table.

### Phase 2: Parallel Processing and Optimization.

- If none of the above strategies work, Optimizer searches for Parallel Processing possibilities. This depends on the Machine's processing capabilities and configuration.

- If that is still not possible, then the final optimization phase starts. Now, the final optimization aim is finding all other possible options for executing the query in the best way. Final optimization phase Algorithms are Microsoft Propriety.

*Query Executor*

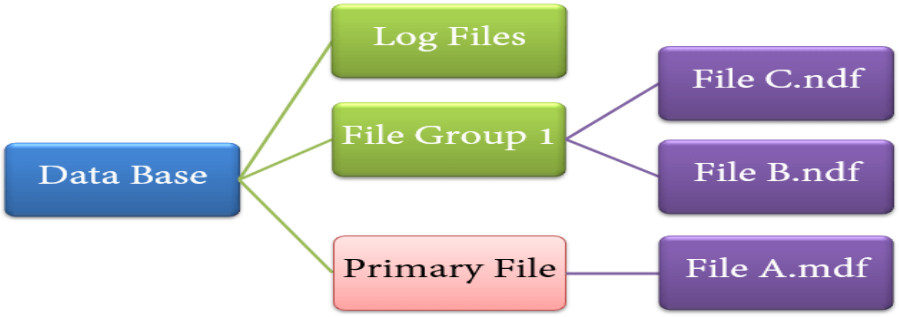



Query executer calls **Access Method.** It provides an execution plan for data fetching logic required for execution. Once data is received from Storage Engine, the result gets published to the Protocol layer. Finally, data is sent to the end user.

## Storage Engine

The work of the Storage Engine is to store data in a storage system like Disk or SAN and retrieve the data when needed.

*File types*




## Primary file

- Every database contains one Primary file.
- This store all important data related to tables, views, Triggers, etc.
- Extension is **.mdf** usually but can be of any extension.

## Secondary file

- Database may or may not contains multiple Secondary files.
- This is optional and contain user-specific data.

- Extension is .**ndf** usually but can be of any extension.

### Log file

- Also known as Write ahead logs.
- Extension is .**ldf**
- Used for Transaction Management.
- This is used to recover from any unwanted instances. Perform important task of Rollback to uncommitted transactions.

Storage Engine has 3 components;

#### *Access Method*

It acts as an interface between query executor and Buffer Manager/Transaction Logs.

Access Method itself does not do any execution.

The first action is to determine whether the query is:

1. Select Statement (DDL)
2. Non- Select Statement (DDL & DML)

Depending upon the result, the Access Method takes the following steps:

1. If the query is **DDL**, SELECT statement, the query is pass to the **Buffer Manager** for further processing.
2. And if query if **DDL, NON-SELECT statement**, the query is pass to Transaction Manager. This mostly includes the UPDATE statement.

#### *Buffer Manager*

Buffer manager manages core functions for modules below:

- Plan Cache
- Data Parsing: Buffer cache & Data storage
- Dirty Page

### Plan Cache

- **Existing Query plan:** The buffer manager checks if the execution plan is there in the stored Plan Cache. If Yes, then query plan cache and its associated data cache is used.
- **First time Cache plan:** If the first-time query execution plan is being run and is complex, it makes sense to store it in in the Plane cache. This will ensure faster availability when the next time SQL server gets the same query. So, it's nothing else but the query itself which Plan execution is being stored if it is being run for the first time.

### Data Parsing: Buffer cache & Data Storage

Buffer manager provides access to the data required. Below two approaches are possible depending upon whether data exist in the data cache or not:

### Buffer Cache – Soft Parsing:

Buffer Manager looks for Data in Buffer in Data cache. If present, then this Data is used by Query Executor. This improves the performance as the number of I/O operation is reduced when fetching data from the cache as compared to fetching data from Data storage.

### Data Storage – Hard Parsing:

If data is not present in Buffer Manager than required Data is searched in Data Storage. If also stores data in the data cache for future use.

### Dirty Page

It is stored as a processing logic of Transaction Manager.

#### *Transaction Manager*

Transaction Manager is invoked when access method determines that Query is a Non-Select statement.

### Log Manager

- Log Manager keeps a track of all updates done in the system via logs in Transaction Logs.
- Logs have **Logs Sequence Number with the Transaction ID and Data Modification Record**.
- This is used for keeping track of **Transaction Committed and Transaction Rollback**.

### Lock Manager

- During Transaction, the associated data in Data Storage is in the Lock state. This process is handled by Lock Manager.
- This process ensures **data consistency and isolation**. Also known as ACID properties.

### Execution Process(including Dirty Page explanation)

- Log Manager start logging and Lock Manager locks the associated data.
- Data's copy is maintained in the Buffer cache.
- Copy of data supposed to be updated is maintained in Log buffer and all the events updates data in Data buffer.
- Pages which store the data is also known as **Dirty Pages**.
- **Checkpoint and Write-Ahead Logging:** This process run and mark all the page from Dirty Pages to Disk, but the page remains in the cache. Frequency is approximately 1 run per minute.But the page is first pushed to Data page of the log file from Buffer log. This is known as **Write Ahead Logging.**
- **Lazy Writer:** The Dirty page can remain in memory. When SQL server observes a huge load and Buffer memory is needed for a new transaction, it frees up Dirty Pages from the cache. It operates on **LRU** – Least recently used Algorithm for cleaning page from buffer pool to disk.

## Database Architecture

- SQL Server maps the database over a set of operating system files that store the database objects.
- Physically, a SQL Server database is a set of two or more operating system files.
- Each database file has two names:
    - SQL Server database files can be stored on either a FAT or an NTFS filesystem.

### Primary data file

- This is the initial default file.
- It contains the configuration information for the database, pointers to the other files in the database, and all of the database objects.

- Every database has one primary data file.
- The preferred filename extension for a primary data file is .mdf.
- Although you can store user objects within the main data file, but it is not recommended.

## Secondary data file

- Secondary data files are optional and used to hold user database objects.
- You can create one or more secondary files within the database to hold the user database objects.
- The recommend filename extension for a secondary data file is .ndf.
- Secondary data files can be spread across multiple disks and are useful as the database's additional storage area.

## Transaction log file

- This is the log file for the database that holds information about all database modification events.
- The information in the transaction log file is used to recover the database.
- A database can have one or more transaction log files.
- Multiple transaction log files do not improve database performance as the SQL Server database engine writes log information sequentially.
- The recommended filename extension for transaction logs is .ldf.

## Filegroups

- In SQL Server databases, you can group the secondary data files logically for administrative purposes. This administrative grouping of data files is called filegroups.
- By default, the SQL Server databases are created with one filegroup, also known as the default filegroup (or primary
- filegroup) for the database.
- The primary database is a member of the default filegroup and can add secondary database files to the default filegroup, Not recommended.
- It is recommended that you create separate filegroups for your secondary data files. This is known as a secondary filegroup (or user-defined filegroup).
- The SQL Server database engine allows you to create one or more filegroups, which can contain one or more secondary data files.
- Transaction log files do not belong to any filegroup.
- The main advantage of filegroups is that they can be backed up or restored separately, or they can be brought online or taken offline separately.

Here's an explanation of how filegroups can be set up in MSSQL(Can also be done with UI, order of operations must be the same)

1. Create a Filegroup: To create a filegroup, you can use the **CREATE DATABASE** statement with the **FILEGROUP** clause. For example:

   ```
   CREATE DATABASE YourDatabase
   CONTAINMENT = NONE
   ON PRIMARY (NAME = YourDatabase_Data,
   FILENAME = 'C:\YourDataPath\YourDatabase.mdf')
   LOG ON (NAME = YourDatabase_Log, FILENAME = 'C:\YourLogPath\YourDatabase.ldf')
   GO
   ```

   In the above example, the filegroup "PRIMARY" is created with a data file ('YourDatabase_Data') and a log file ('YourDatabase_Log'). The data file and log file are stored in different physical locations.

2. Create Additional Filegroups: You can create additional filegroups using the **ALTER DATABASE** statement with the **ADD FILEGROUP** clause. For example:

   ```
   ALTER DATABASE YourDatabase ADD FILEGROUP YourFilegroup
   ```

   This statement adds a new filegroup called "YourFilegroup" to the existing database.

3. Add Files to Filegroups: Once you have created a filegroup, you can add files to it using the **ALTER DATABASE** statement with the **ADD FILE** clause.

   ```
   ALTER DATABASE YourDatabase
   ADD FILE ( NAME = YourDataFile, FILENAME = 'C:\YourDataPath\YourDataFile.ndf',
   SIZE = 100MB, MAXSIZE = UNLIMITED, FILEGROWTH = 10MB )
   TO FILEGROUP YourFilegroup
   ```

   This statement adds a new data file ('YourDataFile') to the "YourFilegroup" filegroup.

4. Move Objects to Filegroups: After setting up filegroups, you can move existing tables and indexes to specific filegroups using the **CREATE INDEX** or **ALTER INDEX** statements. You can specify the desired filegroup in the **ON** clause. For example:

   ```
   CREATE INDEX IX_YourTable_Column
   ON YourTable (Column) WITH (DROP_EXISTING = ON)
   ON YourFilegroup
   ```

This statement creates an index called "IX_YourTable_Column" on "YourTable" and places it in the "YourFilegroup" filegroup.

By utilizing filegroups effectively, you can control the placement of data files and achieve better performance, scalability, and manageability in your MSSQL databases.

## Transaction Log Architecture

- The SQL Server transaction log is a single file which usually has an .LDF file extension.

- Although possible to have multiple log files for a database, the transaction log is always written sequentially and multiple physical log files get treated as one continuous circular file.
- SQL Server uses the transaction log to ensure that all transactions maintain their state even in case of a server or database failure.
- All transactions are written to the Transaction Log before it is written to the data files. This is known as write ahead logging.
- Every action performed on SQL Server is logged in the SQL Server transaction log, multiple entries may be created for a transaction as well as all locks that were taken during the operation.
- Each log entry has a unique number known as the LSN (log sequence number).
- Enough information is written to the log to allow for a transaction to be either re-done (rolled forward) or undone (rolled back).
- Logically the SQL Server Transaction log is divided into multiple sections known as virtual log files or VLFs.
- The logical transaction log gets truncated and expanded in units of VLFs.
- If a VLF no longer contains an active transaction, that VLF can be marked for re-use.
- If the log needs more space, space is allocated in increments of VLFs.
- The number and size of the VLFs is decided by the database engine and it will endeavor to assign as few VLFs as possible.
- Although the size and number of VLFs cannot be configured, it is affected by the initial size and the growth increment of the transaction log.
- If the log growth increment is set too low, it may result in an excessive amount of VLFs which can have an adverse effect on performance. In order to avoid this, it is important to size the log correctly and grow it in sufficiently large increments.
- The **DBCC LOGINFO** command is a database console command in Microsoft SQL Server (MSSQL) used to retrieve information about the transaction log files of a database. It returns a row for each virtual log file (VLF) contained within the transaction log.

# Backup and Restore

Backup of a Database, its Importance for DBA and Different Media's use to take backups

## What is a Database Backup

- Database backup is the process of backing up the operational state, architecture and stored data of database software.
- It is accomplished by replication of the database and can be done for a database or database server. Usually, the RDBMS or equivalent database management program performs file backup.
- Database administrators can use the backup copy of the database to restore the database and its data and logs to its operating state.
- The backup of the database can be saved either locally or on a backup server.
- Also, all the business uses backups to protect their important database from disaster.

## Why Database Backup Is Important

Business Protection : Your business ' growth and survival depend on your operations ' important information. During any kind of disaster, a recovery process and backups help to restore computing devices and also restore data after files have been destroyed or removed. Backups of servers are important for data loss prevention, which can fully interrupt business operations.

Unlimited Data Access : it allows you to access your data without time or location constraints. Multiple copies of data are kept secure at different locations to ensure that all of the information is preserved.

Credibility and Accountability : No customer wants to work with a company that continues to lose information and can not find a way to solve a problem like this. It breaks the faith. It tells a lot about your reliability that able to recover information after a disaster.

Time effective : Unless there is a prompt and reliable backup solutions, businesses can lose their reputation and even face legal action if they lose valuable information from customers or employees. If you are responsible for storing information about your customers and staff, your security system needs to be improved to ensure the best possible protection for important data.

## Where to Store a Backup?

A best practice for maintaining database backup is to keep a copy of the backup files on-site whether it is home or on-site for easy access and a copy off-site in case of fire, any damage to your location that could destroy your on-site copy of the backup.

- Online Backup Services
- USB Drives
- External Hard Drives
- LAN Storage
- Tape Storage
- Backup System

## Recovery Models in MSSQL and their Impact on Database Restore

### Understanding the MSSQL Recovery Models

The recovery model of a database determines the options a DBA has when recovering a database. Each database within SQL Server has a recovery model setting.

- Simple
- Full
- Bulklogged

The transaction log is a detailed log file that is used to record how the database is updated for each transaction.

The SQL Server engine uses this log to maintain the integrity of the database. In the event that a transaction needs to be backed out, or the database crashes for some reason, the transaction log is used to restore the database to a consistent state.

The recovery model for a database determines how much data the SQL Server engine needs to write to the transaction log and what kind of backup can be performed. The initial recovery model setting for a new database is set based on the recovery model of the model system database.

A database's recovery model setting can be changed easily, by either using SSMS(properties tab of database) or TSQL code.

## Simple Recovery Model

In the Simple recovery model, only minimal log information is maintained. The transaction log is automatically truncated, which means that once the transaction is committed, the space used by the log is immediately made available for reuse. This recovery model offers the least amount of protection for data loss since you can only restore a full database backup to recover data

Because log records are removed when a checkpoint occurs, transaction log backups are not supported when using the simple recovery model. This means point-in-time restores cannot be performed when a database has its recovery model set to SIMPLE.

Because the transaction log is automatically cleaned up in this mode, this helps keep the transaction log small and from growing out of control.

Can only perform full and differential backups. You can only restore a database to the point-in-time when a full or differential backup has completed.

Most suited for Development and Test databases, where data loss is acceptable

## Full Recovery Model

The Full recovery model provides the highest level of data protection. In this model, all transactions are fully logged, and the transaction log file grows until it is backed up or truncated manually. It allows for the recovery of a database to a specific point in time (using transaction log backups) or to restore the entire database (using full database backups). Transaction log backups are essential for point-in-time recovery

The full recovery model supports all the options for backing up and restoring a database.

When a transaction log backup is performed against a database that is in full recovery mode, the log records are written to the transaction log backup, and the completed transaction log records are removed from the transaction log. Since every transaction is being written to the transaction log, the full recovery model supports point-in-time restores.

Also means the transaction log needs to be big enough to support logging of all the transactions until a transaction log backup is run. If an application performs lots of transactions, there is the possibility that the transaction log will become full. When the transaction log becomes full, the database stops accepting transactions until a transaction log backup is taken, the transaction log is expanded, or the transaction log is truncated. Therefore, when a database uses the full recovery model, you need to ensure transaction log backups are taken frequently enough to remove the completed transactions from the transaction log before it fills up.

In addition to inserts and update transaction filling up the log, other operations like index create/alter and bulk load operations also write lots of information to the transaction log. Transaction log keeps filling up due to the index and bulk load operations, consider switching to the bulk-logged recovery model while these operations are being performed.

## Bulk-Logged Recovery Model

The bulk-logged recovery model minimizes transaction log space usage when bulk-logged operations like BULK INSERT, SELECT INTO, or CREATE INDEX are executed.

Bulk-logged recovery model functions similar to the full recovery model with the exception that transactions logrecords are minimally logged while bulk-logged operations are running. Minimal logging helps keep the log smaller, by not logging as much information.

Improves the performance of large bulk loading operations by reducing the amount of logging performed.

Bulk-logged transactions are not fully logged, it reduces the amount of space written to the transaction log, which reduces the chance of the transaction log running out of space.

Because bulk-logged operations are minimally logged, it affects point-in-time recoveries. Point-in-time recoveries can still be performed when using the bulk-logged recovery model in some situations.

If the database should become damaged while a minimal bulk-logged operation is being performed, the database can only be recovered to the last transaction log backup created prior to the first bulk-logged operation. When the transaction log backup contains a bulk logged operation, the stopat options cannot be used.

If no bulk-logged operations are performed at all while a database is using the bulk-logged recovery model, then you can still do a point-in-time restore just like you can in the full recovery model.

To minimize data lose when using bulk-load operations you should take a transaction log backup just prior to a bulk-load operation, and then another one right after the bulk-load operation completes. By doing this, a point-in-time recovery can be perform using any transaction log backups taken prior to the bulk-load operation.

As well as for any transaction log backups taken after the special log backup has been taken following the completion of the bulk-load operation.

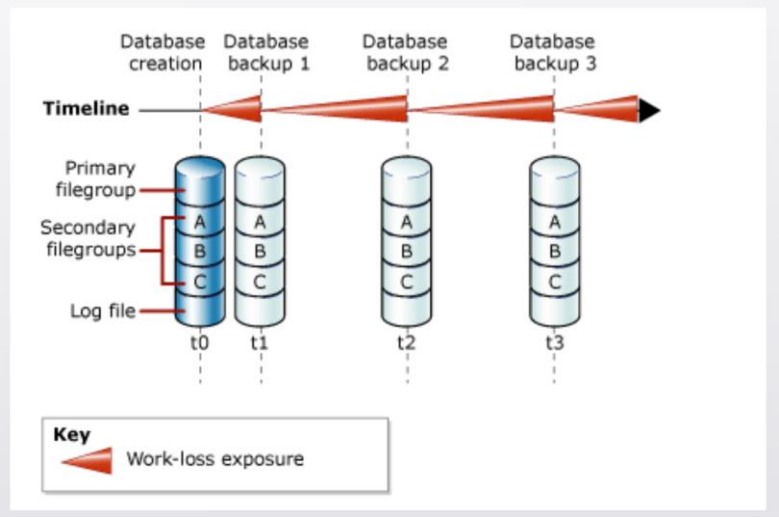
## Full Database Backup

A Full database backup backs up the whole database. This includes part of the transaction log so that the full database can be recovered after a full database backup is restored. Full database backups represent the database at the time the backup finished. It is the foundation of any kind of backup. This is a complete copy, which stores all the objects of the database: Tables, procedures, functions, views, indexes, etc. Having a full backup, you will be able to easily restore a database in exactly the same form as it was at the time of the backup. A full backup must be done at least once before any of the other types of backups can be run—this is the foundation for every other kind of backup.

## Full Backup with Simple Recovery

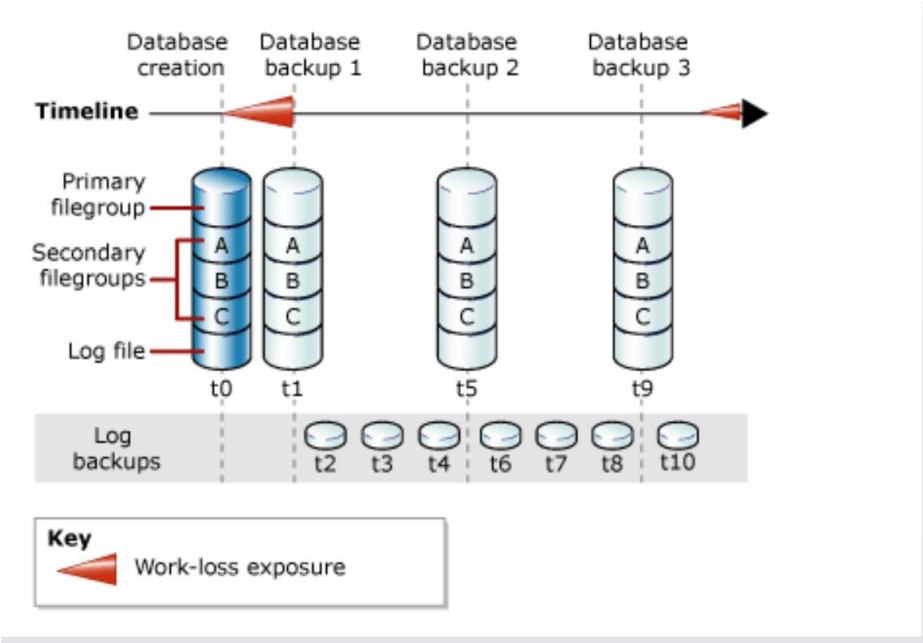
Full Database Backups in Simple Recovery:

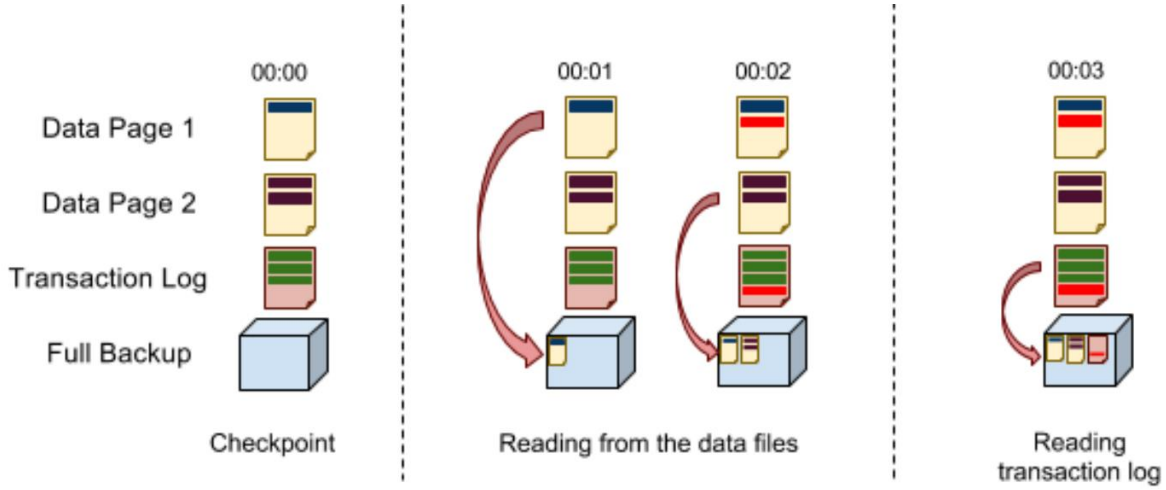
- Under the simple recovery model, after each backup, the database is exposed to potential work loss if a disaster were to occur.
- The work-loss exposure increases with each update until the next backup.
- When the work-loss exposure returns to zero, a new cycle of work-loss exposure starts.
- Work-loss exposure increases over time between backups.




## Full Database Backups in Full Recovery:

- For databases that use full and bulk-logged recovery, database backups are necessary but not sufficient.
- Transaction log backups are also required.
- A full database backup backs up all data files and the active part of the transaction log.
- The active part of the transaction log is necessary to restore the database to a transactionally consistent point.






- The backup process starts at 00:00 and forces a database checkpoint, flushing all dirty pages on the disk. Note that in the simple recovery model, the log is truncated automatically, but not in full and bulk-logged recovery models.
- The backup process reads Data Page 1 at 00:01 and adds it to the backup file.
- At 00:02, the backup process reads Data Page 2 while changes in Data Page 1 take place. These changes are added to the transaction log. The backup contains Data Page 2 and the old version of Data Page 1 (as it was at 00:01).
- At 00:03, the backup process completes data reading and proceeds to read the transaction log. The transaction log contains changes made in Data Page 1 at 00:01, which are added to the backup for later application during the restore process recovery stage.

## Backup database command and options:

- Full backup: BACKUP DATABASE AdventureWorks TO DISK = 'C:\AdventureWorks.BAK'
- File level backup: BACKUP DATABASE TestBackup FILE = 'TestBackup' TO DISK = 'C:\TestBackup_TestBackup.FIL'
- File group backup: BACKUP DATABASE TestBackup FILEGROUP = 'ReadOnly' TO DISK = 'C:\TestBackup_ReadOnly.FLG'
- Full backup multiple files: BACKUP DATABASE AdventureWorks TO DISK = 'C:\AdventureWorks_1.BAK',DISK = 'D:\AdventureWorks_2.BAK',DISK = 'E:\AdventureWorks_3.BAK'
- Full backup with password: BACKUP DATABASE AdventureWorks TO DISK = 'C:\AdventureWorks.BAK' WITH PASSWORD = 'Q!W@E#R$'
- Full backup with stats: BACKUP DATABASE AdventureWorks TO DISK = 'C:\AdventureWorks.BAK' WITH STATS
- Full backup with description: BACKUP DATABASE AdventureWorks TO DISK = 'C:\AdventureWorks.BAK' WITH DESCRIPTION = 'Full backup for AdventureWorks'
- Full backup with mirror: BACKUP DATABASE AdventureWorks TO DISK = 'C:\AdventureWorks.BAK' MIRROR TO DISK = 'D:\AdventureWorks_mirror.BAK' WITH FORMAT

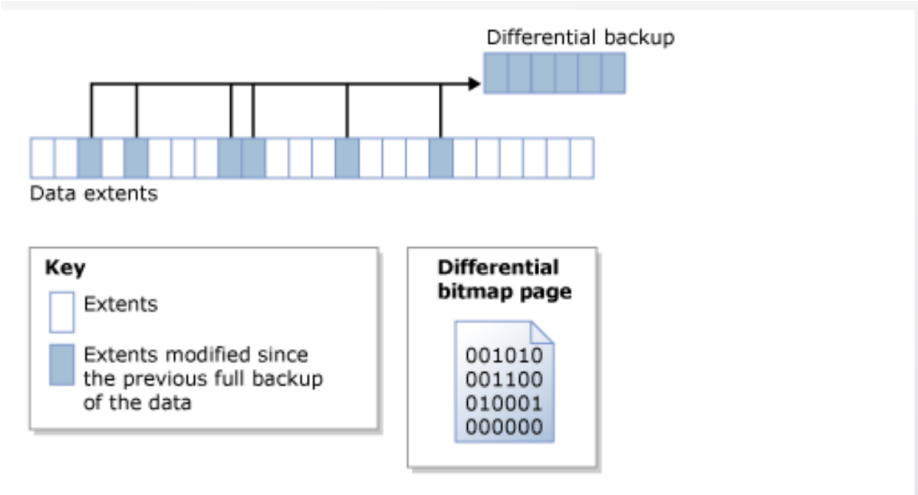
## Full Backup Syntax with important options

```
BACKUP DATABASE [DatabaseName]
TO DISK = 'BackupFilePath'
[WITH
  [OPTION [, ...]]
]
```

1. FORMAT: This option allows overwriting an existing backup file if specified.
2. INIT: This option initializes the backup media, overwriting any existing backup sets. It is often used in combination with FORMAT to ensure a clean start for backups.
3. COMPRESSION: This option enables backup compression to reduce the size of the backup file. It can help save storage space and reduce backup and restore times.
4. CHECKSUM: This option adds checksum validation while reading and writing backup data, ensuring the integrity of the backup.
5. STATS: This option provides detailed information about the backup operation, such as the percentage complete, backup speed, and estimated time to complete.
6. PASSWORD: This option allows setting a password for the backup file, adding an extra layer of security.

## Differential Backups

- A differential backup is based on the most recent, previous full data backup.
- A differential backup captures only the data that has changed since that full backup.
- The full backup upon which a differential backup is based is known as the base of the differential.
- Full backups, except for copy-only backups, can serve as the base for a series of differential backups, including database backups, partial backups, and file backups.
- Creating a differential backup can be very fast compared to creating a full backup.
- This facilitates taking frequent data backups, which decrease the risk of data loss.
- However, before you restore a differential backup, you must restore its base.
- Therefore, restoring from a differential backup will necessarily take more steps and time than restoring from a full backup because two backup files are required.
- Under the full recovery model, using differential backups can reduce the number of log backups that you have to restore.




- A differential backup captures the state of any extents (collections of eight physically contiguous pages) that have changed between when the differential base was created and when the differential backup is created.

- The size of a given differential backup depends on the amount of data that has changed since the base.
- Generally, the older a base is, the larger a new differential backup will be.
- In a series of differential backups, a frequently updated extent is likely to contain different data in each differential backup.
- The differential backup contains only these changed data extents.
- The differential backup operation relies on a bitmap page that contains a bit for every extent. For each extent updated since the base, the bit is set to 1 in the bitmap.

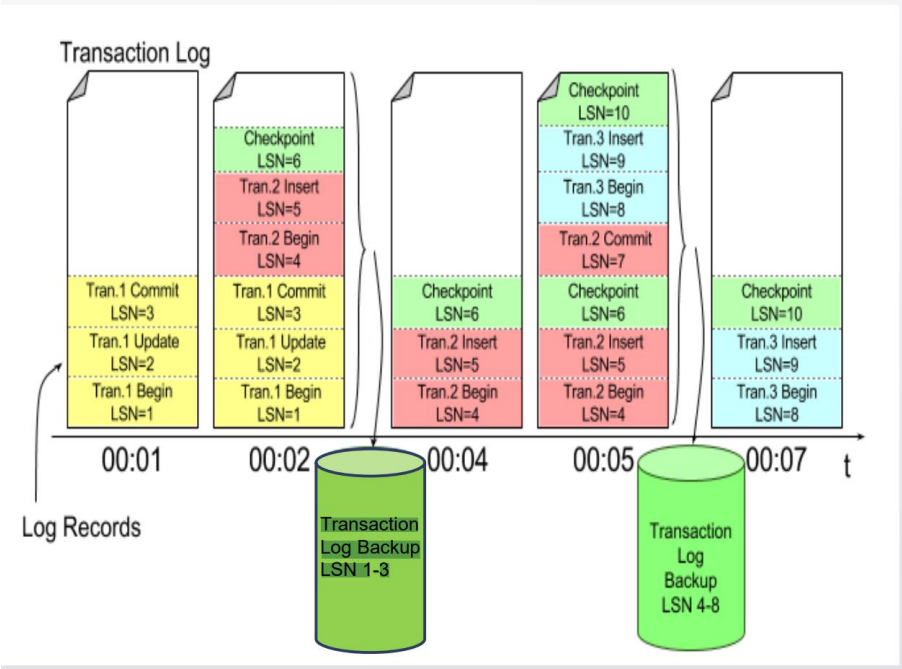
- A differential backup taken soon after its base is usually significantly smaller than the differential base, saving storage space and backup time.
- However, as a database changes over time, the difference between the database and a specific differential base increases.
- The longer the time between a differential backup and its base, the larger the differential backup is likely to be.
- This means that differential backups can eventually approach the size of the differential base, losing the advantages of a faster and smaller backup.
- As the size of differential backups increases, restoring a differential backup can significantly increase the time required to restore a database.
- To establish a new differential base for the data, take a new full backup at set intervals.
- For example, you might take a weekly full backup of the whole database followed by a regular series of differential database backups during the week.
- At restore time, before restoring a differential backup, you must restore its base. Then, restore only the most recent differential backup to bring the database forward to the time when that differential backup was created.

Differential Backup database command:

- Differential backup 1: BACKUP DATABASE AdventureWorks TO DISK = 'C:\Temp\DatabaseBackups\AdventureWorks_Diff_1.bak' WITH DIFFERENTIAL

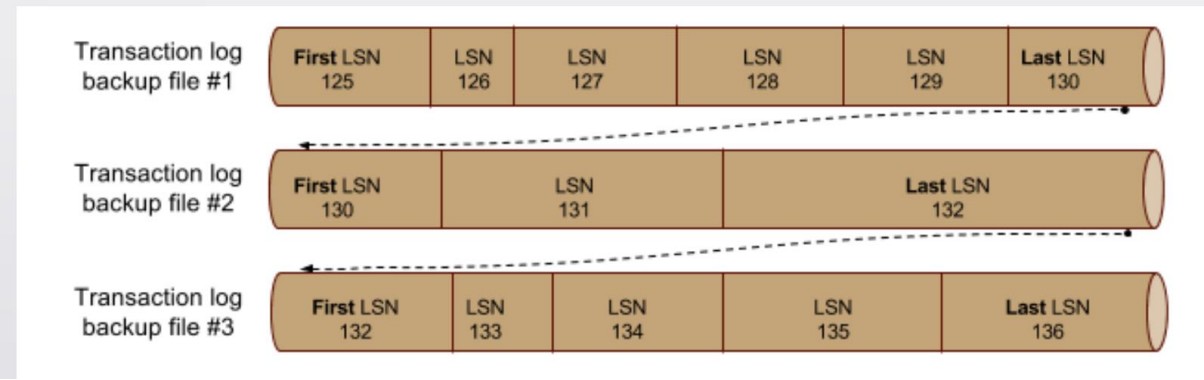
## Transaction Log Backups

- Minimally, you must have created at least one full backup before you can create any log backups.
- After that, the transaction log can be backed up at any time unless the log is already being backed up.
- Take log backups frequently, both to minimize work loss exposure and to truncate the transaction log.
- Typically, create a full database backup occasionally, such as weekly.
- Create a series of differential database backups at a shorter interval, such as daily.
- Independent of the database backups, the database administrator backs up the transaction log at frequent intervals.
- For a given type of backup, the optimal interval depends on factors such as the importance of the data, the size of the database, and the workload of the server.




- A transaction log backup contains all transaction log records that have been made between the last transaction log backup or the first full backup and the last log record that is created upon completion of the backup process.
- The transaction log backup allows restoring a database to a particular point-in-time before the failure has occurred.
- It is incremental, meaning that to restore a database to a certain point-in-time, all transaction log records are required to replay database changes up to that particular point-in-time.

## Transaction Log Backup Chain




- A transaction log backup chain is a continuous sequence of transaction log backups starting with the first full database backup.
- Each backup from the chain has its own FirstLSN (oldest log record in the backup set) and LastLSN (the number of the next log record after the backup set), that help to restore transaction log backup files in the right sequence.
- To get backup's FirstLSN and LastLSN values, you can use the following command:
    - RESTORE HEADERONLY FROM DISK = 'log1.bak'
- Some instances when the log backup chain is broken:

    - Adding T-SQL options TRUNCATE_ONLY or NO_LOG after a BACKUP LOG command.
    - Switching from full or bulk-logged recovery models to simple and back again.

- Two common instances in which the log backup chain is not broken:

    - Starting a full, differential, or file/filegroup backup.
    - Switching from full recovery model to bulk-logged and back again.

### Tail-log Backup

- A tail-log backup captures any log records that have not yet been backed up (the tail of the log) to prevent work loss and to keep the log chain intact.
- Before you can recover a SQL Server database to its latest point in time, you must back up the tail of its transaction log.
- The tail-log backup will be the last backup of interest in the recovery plan for the database.
- Not all restore scenarios require a tail-log backup. You do not need a tail-log backup if the recovery point is contained in an earlier log backup.
- Also, a tail-log backup is unnecessary if you are moving or replacing (overwriting) a database and do not need to restore it to a point of time after its most recent backup.

Log Backup database command

- BACKUP LOG AdventureWork TO DISK = 'log1.trn'

## Restore Overview

Restore of a database:

- To recover a SQL Server database from a failure, a database administrator has to restore a set of SQL Server backups in a logically correct and meaningful restore sequence.
- SQL Server restore and recovery supports restoring data from backups of a whole database, a data file, or a data page.
- The database (a complete database restore): The whole database is restored and recovered, and the database is offline for the duration of the restore and recovery operations.
- The data file (a file restore): A data file or a set of files is restored and recovered. During a file restore, the filegroups that contain the files are automatically offline for the duration of the restore. Any attempt to access an offline filegroup causes an error.
- The data page (a page restore): Under the full recovery model or bulk-logged recovery model, you can restore individual databases. Page restores can be performed on any database, regardless of the number of filegroups.

## Restore Scenarios

| Restore Scenario | Under Simple Recovery | Under Full/Bulk-Logged Recovery |
|---|---|---|
| Complete database restore | This is the basic restore strategy. A complete database restore might involve simply restoring and recovering a full database backup. Alternatively, a complete database restore might involve restoring a full database backup followed by restoring and recovering a differential backup. | This is the basic restore strategy. A complete database restore involves restoring a full database backup and, optionally, a differential backup (if any), followed by restoring all subsequent log backups (in sequence). The complete database restore is finished by recovering the last log backup and also restoring it (RESTORE WITH RECOVERY). |
| File restore | Restore one or more damaged read-only files, without restoring the entire database. File restore is available only if the database has at least one read-only filegroup. | Restores one or more files, without restoring the entire database. File restore can be performed while the database is offline or, for some editions of SQL Server, while the database remains online. During a file restore, the filegroups that contain the files that are being restored are always offline. |
| Page restore | Not applicable | Restores one or more damaged pages. Page restore can be performed while the database is offline or, for some editions of SQL Server, while the database remains online. During a page restore, the pages that are being restored are always offline. An unbroken chain of log backups must be available, up to the current log file, and they must all be applied to bring the page up-to-date with the current log file. |

## Recovery Models and Supported Restore Operations

| Restore Operations | Full Recovery | Bulk-Logged Recovery | Simple Recovery |
|---|---|---|---|
| Data recovery | Complete recovery (if the log is available). Some data-loss exposure. Any data since the last full or differential backup is lost. | Complete recovery (if the log is available). Some data-loss exposure. Any data since the last full or differential backup is lost. | Not supported. |
| Point-in-time restore | Any time covered by the log backups. Disallowed if the log backup contains any bulk-logged changes. | Any time covered by the log backups. Disallowed if the log backup contains any bulk-logged changes. | Not supported. |

| Restore Operations | Full Recovery | Bulk-Logged Recovery | Simple Recovery |
|---|---|---|---|
| File restore | Full support. | Sometimes.** | Available only for read-only secondary files. |
| Page restore | Full support. | Sometimes.** | None. |

**Note:**

- Full support: The restore operation is fully supported under the mentioned recovery model.
- Sometimes: The restore operation is supported under specific conditions or limitations.

## Full Backup Recovery

A full database restore in MSSQL involves restoring the entire database from a full database backup. It is a common restore strategy used to recover a database to a specific point in time or after a failure. Here's an explanation of the syntax and options used for a full database restore in MSSQL:

Syntax:

RESTORE DATABASE database_name FROM backup_device WITH [OPTIONS]

Explanation:

- **RESTORE DATABASE**: This is the T-SQL statement used to initiate the database restore process.
- **database_name**: Specify the name of the database you want to restore.
- **FROM backup_device**: Specify the source of the backup. The backup device can be a disk file path, URL, or a tape device.
    - If restoring from a disk file: **FROM DISK = 'backup_file_path'**
    - If restoring from a URL: **FROM URL = 'backup_url'**
    - If restoring from a tape: **FROM TAPE = 'tape_device_name'**

Options: When performing a full database restore, you can use various options to customize the restore process. Some commonly used options include:

1. **WITH RECOVERY**:
    - This option brings the database online after the restore and recovery operations are completed.
    - Use this option when you don't need to apply any additional transaction log backups after the full restore.
2. **WITH NORECOVERY**:
    - This option leaves the database in a non-operational state after the restore operation.
    - Use this option when you plan to restore additional transaction log backups and want to keep the database in a restoring state.
3. **WITH REPLACE**:

- This option allows the restore operation to overwrite an existing database with the same name.
- Use this option when you want to replace an existing database with the restored version.

4. **WITH MOVE**:

    - This option is used when the physical file locations of the original database do not match the desired restore location.
    - It allows you to specify the new file locations for the restored database files.
    - Example: **WITH MOVE 'logical_data_file_name' TO 'new_data_file_path', MOVE 'logical_log_file_name' TO 'new_log_file_path'**

## Differential Backup Restore

A differential database backup captures only the data that has changed since the last full database backup, which makes it faster to perform a restore operation since you only need to restore the last full backup and the last differential backup.

To perform a differential database restore in MSSQL, you need to follow these steps:

1. Restore the last full database backup
2. Restore the last differential database backup
3. Recover the database

Here's the syntax for restoring a differential database backup:

RESTORE DATABASE <database_name>

FROM DISK = '<path_to_differential_backup>' WITH NORECOVERY;

In the **RESTORE DATABASE** statement, you specify the name of the database you want to restore in **<database_name>**. In the **FROM DISK** option, you specify the file path of the differential backup you want to restore. Finally, you add the **NORECOVERY** option to indicate that you want to leave the database in a restoring state so that you can add more backups to the restore sequence.

Once you have restored the last full backup and the last differential backup, you can recover the database to bring it online:

RESTORE DATABASE <database_name> WITH RECOVERY;

In this **RESTORE DATABASE** statement, you don't need to specify a backup file because the database is already in the restoring state. You simply add the **WITH RECOVERY** option to indicate that you want to bring the database online and make it available for user access.

Note that you can also restore multiple differential backups in a sequence by specifying them in the **FROM DISK** option, separated by commas. For example:

RESTORE DATABASE <database_name>

FROM DISK = '<path_to_last_full_backup>',

DISK = '<path_to_first_differential_backup>',

DISK = '<path_to_second_differential_backup>', ... WITH NORECOVERY;

## Transaction Log Backup Restore

In Microsoft SQL Server (MSSQL), transaction log backups are used to restore a database to a specific point in time or to recover transactions that occurred after the last full or differential backup. Restoring transaction log backups allows you to bring the database up to the desired state, including all the committed transactions.

To perform a transaction log restore in MSSQL, you need to follow these steps:

1. Restore the last full database backup (if not already restored).
2. Restore the subsequent transaction log backups in sequence.
3. Recover the database.

Here's the syntax for restoring a transaction log backup:

RESTORE LOG <database_name> FROM DISK = '<path_to_log_backup>' WITH NORECOVERY;

In the **RESTORE LOG** statement, you specify the name of the database you want to restore in **<database_name>**. The **FROM DISK** option is used to specify the file path of the transaction log backup you want to restore. The **NORECOVERY** option indicates that you want to leave the database in a restoring state, allowing you to restore additional transaction log backups.

You can restore multiple transaction log backups in sequence by specifying them in the **FROM DISK** option, separated by commas.

After restoring all the necessary transaction log backups, you can recover the database using the following statement:

RESTORE DATABASE <database_name> WITH RECOVERY;

In this **RESTORE DATABASE** statement, you don't need to specify a backup file because the database is already in the restoring state. The **WITH RECOVERY** option indicates that you want to bring the database online and make it available for user access.

It's important to note that transaction log backups can only be restored if the database is in the full or bulk-logged recovery model. The simple recovery model does not support transaction log backups and point-in-time recovery.

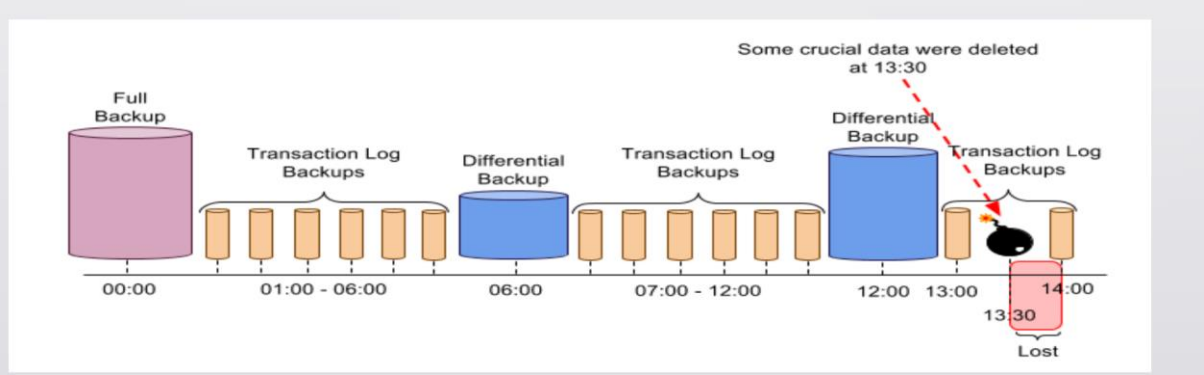
## Point in Time Restore

Point-in-time recovery allows you to restore a database to a specific state at any given point in time.

This type of recovery is applicable only to databases that are running under the full or bulk-logged recovery model.

If the database is running under the bulk-logged recovery model and the transaction log backup contains bulk-logged changes, recovery to a point-in-time is not possible.

For example, let's say some crucial data was deleted at 13:30 accidentally and we don’t want this action to take place during restore, and the last transaction log backup was made at 14:00. The transaction log backup from 14:00 allows us to restore the database to its state at 13:29:59. In this case, only 30 minutes of updates will be lost.




To perform a point-in-time restore of a database, follow these steps:

1. Start by restoring the full database backup using the following syntax:

   RESTORE DATABASE AdventureWorks2016 FROM DISK = 'AdventureWorks2016_full.bak' WITH NORECOVERY, REPLACE

   This restores the database to the state captured by the full backup.

2. After the full backup is restored, restore the last differential backup. In this example, the last differential backup was made at 12:00. Use the following syntax:

   RESTORE DATABASE AdventureWorks2016 FROM DISK = 'AdventureWorks2016_diff.diff' WITH NORECOVERY

The differential backup includes all changes made to the database since the last full backup.

3. Finally, apply all transaction log backups that were created after the differential backup. Restore the transaction log backups in the same sequence in which they were created. Specify the time to which the database should be restored using the STOPAT option when restoring the last log backup. For example:

   RESTORE LOG AdventureWorks2016 FROM DISK = 'AdventureWorks2016_log_013.trn' WITH NORECOVERY RESTORE LOG AdventureWorks2016 FROM DISK = 'AdventureWorks2016_log_014.trn' WITH STOPAT = '2020-11-19 13:29:59.000', RECOVERY

Restore all backups except the last one with the NORECOVERY option. The last backup should be restored with the RECOVERY option to bring the database into its working state.

By following these steps and restoring the backups in the correct sequence, you can perform a point-in-time restore of a database in MSSQL.

## DBCC CHECKDB

- DBCC CHECKDB checks the logical and physical integrity of all the objects in the specified database.
- It includes three commands:

    - DBCC CHECKALLOC: Checks the consistency of disk space allocation structures for a specified database.
    - DBCC CHECKTABLE: Checks the integrity of all the pages and structures that make up the table or indexed view.
    - DBCC CHECKCATALOG: Checks for catalog consistency within the specified database. The database must be online.
- It validates the physical and logical integrity of the database, including indexed views and Service Broker data.

**DBCC CHECKDB Syntax and Options**

- DBCC CHECKDB ('DatabaseName')
- Options:
    - NOINDEX: Skips intensive checks of non-clustered indexes for user tables.
    - NO_INFOMSGS: Suppresses all information messages.
    - PHYSICAL_ONLY: Limits the checking to the integrity of the physical structure of pages and records and the allocation consistency of the database.
    - TABLOCK: Obtains locks instead of using an internal database snapshot, improving performance but reducing concurrency.
    - DATA_PURITY: Checks the database for invalid or out-of-range column values.

**DBCC Execution Results**

- Examples of execution results:
    - Inaccurate page counts can be corrected using DBCC UPDATEUSAGE.
    - Table errors may be reported, and repair_allow_data_loss can be used to attempt repairs, but data loss may occur.
- Data corruption errors may require restoring from a backup, ensuring it doesn't contain corrupted data and minimizing data loss.
- Repair options:
    - REPAIR_REBUILD: Performs repairs with no possibility of data loss, including quick repairs and index rebuilding.
    - REPAIR_ALLOW_DATA_LOSS: Attempts to repair all reported errors, but data loss may occur.

**DBCC CHECKDB Examples**

Checking AdventureWorks2016 database without displaying information messages:

```
DBCC CHECKDB(N'AdventureWorks2016') WITH NO_INFOMSGS
```

Checking AdventureWorks2016 database and performing repairs by rebuilding:

```
DBCC CHECKDB(N'AdventureWorks2016', REPAIR_REBUILD)
```

Checking AdventureWorks2016 database and allowing data loss during repairs:

DBCC CHECKDB(N'AdventureWorks2016', REPAIR_ALLOW_DATA_LOSS)

Switching to Single User Mode(good to do before repair):

ALTER DATABASE AdventureWorks2016 SET SINGLE_USER

Switching to Multi User Mode:

ALTER DATABASE AdventureWorks2016 SET MULTI_USER

## Page Level Restore/Recovery

Page level restore or recovery is a process in which only the damaged or corrupted pages of a database are restored or recovered. This process is often faster and less resource-intensive than restoring or recovering an entire database.

Here are the steps to perform a page-level restore or recovery in MS SQL Server:

1. Identify the damaged or corrupted pages in the database by reviewing the error messages generated by the DBCC CHECKDB command.
2. Take a full database backup if one is not already available. This backup will be used as the starting point for the restore.
3. Use the RESTORE DATABASE command to specify the damaged or corrupted filegroups and their corresponding backup files. Here's an example syntax:

RESTORE DATABASE [DatabaseName] FILEGROUP = 'FileGroupName' FROM DISK = 'BackupFile.bak' WITH NORECOVERY;

In this syntax, the FILEGROUP parameter specifies the name of the filegroup containing the damaged or corrupted pages, and the NORECOVERY parameter specifies that the restore should not be recovered yet.

4. Use the RESTORE PAGE command to restore the damaged or corrupted pages. Here's an example syntax:

RESTORE PAGE [DatabaseName] FILE = 'LogicalFileName' PAGE = 'PageNumber' FROM DISK = 'BackupFile.bak';

In this syntax, the FILE parameter specifies the logical name of the file containing the damaged or corrupted page, the PAGE parameter specifies the number of the damaged or corrupted page, and the FROM DISK parameter specifies the backup file to restore from.

5. Repeat step 4 for all the damaged or corrupted pages.
6. Use the RECOVERY option in the RESTORE DATABASE command to recover the database to a consistent state:

RESTORE DATABASE [DatabaseName] WITH RECOVERY;

This step brings the database back online and allows users to access it.

It's important to note that page-level restore or recovery should only be used as a last resort, as it can result in data loss if not done carefully. It's always best to have regular backups and a solid disaster recovery plan in place to minimize the risk of data loss.

## Restoring Master Database

Restoring the master database in MS SQL Server is a critical task that requires careful attention. The master database contains crucial system information, configuration settings, and metadata. Here are the general steps to restore the master database:

1. Prepare for the restore:
    - Ensure you have a recent backup of the master database.
    - Make note of any custom configurations, such as server-level settings, linked servers, or SQL Server Agent jobs, which may need to be reconfigured after the restore.
2. Stop the SQL Server service:
    - Open the SQL Server Configuration Manager.
    - Stop the SQL Server service to ensure exclusive access to the master database files.
3. Replace the master database files:

- Locate the backup file of the master database.
- Copy or move the backup file to the appropriate location of the SQL Server instance. By default, the master database files are located in the "DATA" folder where SQL Server is installed.
- Rename the existing master database files (e.g., "master.mdf" and "mastlog.ldf") to preserve them as a backup.
- Copy or move the backup file to the same location and rename it to match the original file names ("master.mdf" and "mastlog.ldf").

4. Start the SQL Server service in single-user mode:
    - Open a command prompt with administrative privileges.
    - Navigate to the SQL Server's "BINN" directory.
    - Run the following command to start the SQL Server service in single-user mode:

      sqlservr.exe -m -c
    - This command starts the SQL Server service in a minimal configuration, allowing only a single connection for administrative tasks.
5. Connect to the SQL Server instance:
    - Open a new command prompt or SQL Server Management Studio (SSMS) and connect to the SQL Server instance using the "localhost" server name and specifying the appropriate administrative credentials.
6. Restore the master database:
    - Execute the RESTORE DATABASE command to restore the master database from the backup file. Here's an example syntax:

    RESTORE DATABASE master FROM DISK = 'Path\To\Backup\File.bak' WITH REPLACE;

Ensure to replace "Path\To\Backup\File.bak" with the actual path to your master database backup file.

7. Verify the restore and reconfigure:

- Once the restore completes successfully, verify the status of the master database.
- Check if any custom configurations, such as server-level settings, linked servers, or SQL Server Agent jobs, need to be reconfigured based on your notes from step 1.
- Restart the SQL Server service to enable normal connections.

## Restoring when MSSQL won't start

Restoring the master database from a backup when the master database is corrupted and SQL Server won't start requires additional steps. Here's a general outline of the process:

1. Stop the SQL Server service:
    - Open the Services management console (services.msc).
    - Locate the SQL Server service and stop it.
2. Copy the master database backup file:
    - Locate the backup file of the master database that you want to restore.
    - Copy the backup file to a safe location or an alternate location on the server.
3. Start SQL Server in minimal configuration mode:
    - Open a command prompt with administrative privileges.
    - Navigate to the SQL Server's "BINN" directory. The path is typically:

      cd C:\Program Files\Microsoft SQL Server\MSSQL{InstanceName}\MSSQL\Binn

      Replace **{InstanceName}** with the actual instance name of SQL Server.

    - Start SQL Server in minimal configuration mode by running the following command:

      sqlservr.exe -s{InstanceName} -T3608

      Replace **{InstanceName}** with the actual instance name of SQL Server.
4. Restore the master database:

- Open a new command prompt window.
- Navigate to the SQL Server's "BINN" directory.
- Use the SQLCMD utility to connect to the SQL Server instance in minimal configuration mode:

  sqlcmd -S .\{InstanceName} -T3608

  Replace **{InstanceName}** with the actual instance name of SQL Server.

- Once connected, restore the master database from the backup file using the RESTORE DATABASE command:

  RESTORE DATABASE master FROM DISK = 'Path\To\Backup\File.bak' WITH REPLACE;

Replace **'Path\To\Backup\File.bak'** with the actual path to your master database backup file.

- Exit the SQLCMD prompt by typing **EXIT**.

5. Start the SQL Server service:
    - Go back to the Services management console (services.msc).
    - Start the SQL Server service.
6. Verify the restore:
    - Check if SQL Server starts up correctly.
    - Verify that the master database is functioning as expected.

Remember to adjust the commands and paths according to your specific environment. It's crucial to have a valid and recent backup of the master database before attempting the restore. If you encounter any issues or are unsure about performing these steps, it's recommended to consult the SQL Server documentation or seek assistance from a database professional.

### Alternate Method

- Prepare another MSSQL instance of same major version
- Restore the master db backup in this instance as a userdb(ex master_trial)
- Detach this database from the MSSQL instance
- Copy the mdf and ldf file to the location where corrupted Master files are present.
- Rename corrupted files(ex to master_old)
- Rename copied files from master_trial.mdf to master.mdf, similarly for log file

## Maintenance Plans

In Microsoft SQL Server, maintenance plans are a feature that allows you to automate and schedule routine maintenance tasks for your databases. Maintenance plans provide an easy-to-use graphical interface to create and manage a set of tasks that help optimize the performance, integrity, and availability of your databases. Here are some key points about maintenance plans in MSSQL:

1. Creating a Maintenance Plan:
    - Open SQL Server Management Studio (SSMS) and connect to your SQL Server instance.
    - Expand the "Management" node in the Object Explorer.
    - Right-click on the "Maintenance Plans" node and select "Maintenance Plan Wizard".
    - Follow the wizard to define the maintenance tasks and schedule.
2. Maintenance Tasks: Maintenance plans can include various tasks to perform routine maintenance. Some commonly used tasks are:
    - Back up databases: Create full, differential, or transaction log backups.
    - Rebuild or reorganize indexes: Optimize the performance of your database by rebuilding or reorganizing fragmented indexes.

- Update statistics: Keep statistics up to date for query optimization.
- Check database integrity: Use DBCC CHECKDB to verify the integrity of your databases.
- Shrink database: Reclaim unused space in database files.
- Execute SQL Server Agent jobs: Include other SQL Server Agent jobs as part of your maintenance plan.

3. Scheduling Maintenance Plans:

    - Maintenance plans can be scheduled to run at specific times and intervals.
    - You can choose to run them daily, weekly, or at a custom schedule.
    - Configure notifications to receive emails or alerts when maintenance tasks encounter errors.

4. Managing and Modifying Maintenance Plans:

    - You can modify existing maintenance plans or create new ones using SSMS.
    - Use the Maintenance Plan Designer to add, remove, or modify tasks within the plan.
    - Tasks can be rearranged, edited, or disabled as needed.
    - It's important to periodically review and adjust maintenance plans to meet the changing needs of your databases.

5. Monitoring and Logging:

    - Maintenance plans generate logs that provide information about the execution of each task.
    - Logs can be helpful for troubleshooting and auditing purposes.
    - You can view the maintenance plan history and review the logs within SSMS.

Maintenance plans provide a convenient way to automate routine maintenance tasks in SQL Server. However, it's important to note that they have some limitations and may not be suitable for complex maintenance scenarios. In such cases, you may consider using SQL Server Agent jobs or implementing custom scripts.

## Shrink Operation

In Microsoft SQL Server, the "shrink" operation is used to reduce the physical size of a database or its individual files. When you shrink a database, SQL Server reclaims unused space within the data and log files, resulting in a smaller file size on disk.

However, it's important to note that shrinking a database should be approached with caution, as it can have potential negative effects on performance and fragmentation. Here's some information to consider when deciding how and when to use the shrink operation:

1. Fragmentation: As data is added, modified, and removed in a database, it can become fragmented, meaning that data pages are scattered across the storage space. Shrink operation can alleviate fragmentation to some extent by rearranging data pages. However, frequent shrinking can lead to further fragmentation, which negatively impacts performance.

2. Performance impact: The shrink operation is an intensive process that can consume significant system resources, including CPU and disk I/O. Shrinking large databases or files

can take a considerable amount of time and may affect the overall performance of your SQL Server during the operation.

3. Disk space consideration: While shrinking a database can free up disk space, it's crucial to evaluate whether the space reclamation is necessary. Shrinking a database too frequently or excessively can result in a cycle of growing and shrinking, leading to unnecessary disk activity and potential performance degradation.

4. Maintenance tasks: It's generally recommended to incorporate regular database maintenance tasks like index rebuilds and statistics updates to manage fragmentation and optimize performance. These tasks can help minimize the need for frequent shrink operations.

If you determine that shrinking a database is necessary, here are some guidelines:

- Evaluate the need: Assess whether the space reclamation is necessary based on the specific circumstances, such as a significant reduction in data or disk space constraints.
- Plan maintenance window: Perform the shrink operation during a maintenance window or during periods of low database activity to minimize the impact on the server and its users.
- Shrink in small increments: Instead of shrinking the entire database at once, consider shrinking it in smaller increments. This approach can help minimize the impact on performance and fragmentation.
- Monitor and reorganize: After shrinking, monitor the database for any signs of increased fragmentation. If needed, perform index maintenance tasks like rebuilding or reorganizing indexes to maintain optimal performance.

It's important to exercise caution when deciding to shrink a database and consider the potential implications it may have on your specific environment. It's generally recommended to consult with a database administrator or SQL Server professional who can assess your situation and provide tailored advice based on your specific needs and requirements.

# MSSQL User Management

## MSSQL Security Model:

The MSSQL Security Model can be understood by drawing an analogy to owning a house. Let's go through the key points:

1. House and Keys:

    - The house represents an instance of SQL Server.
    - Keys to the house symbolize logins in SQL Server.
    - As the owner, you have a key (login) to the house (SQL Server instance).
    - Only those with a key (login) can open the door and connect to the instance.

2. Roles:

    - Mom and Dad in the analogy represent logins with the sysadmin role, which grants them full rights and privileges (God rights) at the server and database level.

    - Mom and Dad can do anything they want in any room (database) of the house (SQL Server instance).

3. Children:

- Suzie and John, the kids, represent logins without sysadmin privileges.
- They have their own keys (logins) to enter the house (SQL Server instance), but with some restrictions.
- Suzie and John cannot enter each other's rooms (databases) and are not allowed to perform certain tasks like changing locks, painting walls, or using the stove (limited permissions).

4. Database and Users:

- Rooms in the house correspond to databases in SQL Server.
- To access a specific database (room), a login (key) is required.
- Users in SQL Server are created within databases and represent individuals granted access to use the database (room).
- Suzie and Bobby (hypothetical example) need to be granted access to a database (room), so they are created as users in that particular database.

5. Access Control:

- Mom and Dad, being the owners, do not need to be explicitly granted as users in any database because they have full control over the server and databases.
- Permissions can be granted or denied at various levels, even down to specific database objects like tables.
- Mom can access Dad's man-cave, but the kids are not allowed. Suzie and Bobby are not allowed in each other's rooms either.

6. Granularity:

- The MSSQL Security Model allows for very granular control if needed, based on the requirements of the application or organization.
- Permissions can be fine-tuned to provide access only to specific objects, based on user roles and responsibilities.

7. Windows and SQL Server Logins:

- Logins can be either Windows-based (using Windows authentication) or created within SQL Server (using SQL Server authentication), depending on the configuration and security requirements set by the organization.

By understanding the MSSQL Security Model, one can effectively manage access and permissions within an SQL Server environment, ensuring proper security measures are in place.

## MSSQL Authentication Modes:

To ensure data protection, authentication and authorization mechanisms are essential in SQL Server. These mechanisms verify user identities and determine their access to specific data resources. Here are the key points regarding MSSQL authentication modes:

### Authentication and Authorization:

- Authentication verifies user identities, while authorization controls user access and actions.
- SQL Server achieves these through security principals, securables, and permissions.

### Windows Authentication:

- Also known as integrated security.
- Windows authentication integrates with Windows user and group accounts.
- Users can log into SQL Server using a local or domain Windows account.
- When a Windows user connects to a SQL Server instance, the database engine validates the login credentials against the Windows principal token.
- Windows authentication eliminates the need for separate SQL Server credentials.
- Microsoft recommends using Windows Authentication whenever possible.

### SQL Server and Windows Authentication:

- Sometimes referred to as mixed mode.
- In certain scenarios, SQL Server Authentication may be required.
- For example, when users connect from non-trusted domains or when the server hosting SQL Server is not part of a domain.
- SQL Server Authentication allows users to connect using login mechanisms built into SQL Server without relying on Windows accounts.
- Under this mode, users provide a username and password directly to connect to the SQL Server instance, bypassing Windows Authentication.
- The authentication mode can be specified during SQL Server instance setup or changed later through the server's properties.

Understanding and selecting the appropriate authentication mode is crucial for securing SQL Server and controlling user access to data resources. Windows Authentication is generally recommended, but SQL Server Authentication can be used in specific circumstances where Windows Authentication is not feasible.

## Principals (Part-1) : Server Logins, Server Roles

### SQL Server Security Components:

SQL Server provides three types of components for controlling user access and permissions:

- Principals
- Securables
- Permissions

These components work together to authenticate and authorize SQL Server users. Each principal must be granted the appropriate permissions on specific securables to enable access to SQL Server resources.

## Server Logins:

SQL Server supports four types of logins: Windows, SQL Server, certificate-mapped, and asymmetric key-mapped. The focus here is on Windows and SQL Server logins.

Windows logins are associated with local or domain Windows accounts. They are created using the CREATE LOGIN statement within the context of the master database. The statement includes the Windows account preceded by the computer or domain name and a backslash. Example:

```
USE master;
CREATE LOGIN [win10b\winuser01] FROM WINDOWS
WITH DEFAULT_DATABASE = master, DEFAULT_LANGUAGE = us_english;
```

SQL Server logins are not associated with Windows accounts. They are created using the CREATE LOGIN statement with a specified password. Example:

```
USE master;
CREATE LOGIN sqluser01
WITH PASSWORD = 'tempPW@56789' MUST_CHANGE, CHECK_EXPIRATION = ON,
DEFAULT_DATABASE = master, DEFAULT_LANGUAGE = us_english;
```

### *Creating Server Logins with UI*

To create server logins using SQL Server Management Studio (SSMS) UI, follow these steps:

1. Open SQL Server Management Studio and connect to the SQL Server instance.
2. Expand the "Security" folder in the Object Explorer panel.
3. Right-click on the "Logins" folder and select "New Login" from the context menu. This will open the "Login - New" dialog.
4. In the "General" tab of the dialog, enter the login name in the "Login name" field. This can be a Windows account or a SQL Server login.
5. Select the appropriate authentication mode:
    - For Windows Authentication, leave the "Windows Authentication" option selected.
    - For SQL Server Authentication, choose the "SQL Server Authentication" option and enter a password for the login.
6. If you selected SQL Server Authentication, specify the default database for the login in the "Default database" field.
7. In the "Server Roles" tab, choose the server roles you want to assign to the login. Server roles define the permissions and administrative tasks the login can perform at the server level.
8. In the "User Mapping" tab, you can map the login to specific databases and assign roles to the login at the database level. Select the databases in the "Users mapped to this login" section and choose the appropriate database roles in the "Database role membership" section.
9. Click "OK" to create the login. The new login will now be listed under the "Logins" folder in the Object Explorer.

By following these steps, you can create server logins using the SSMS UI.

## Server Roles:

SQL Server provides server-level roles to manage permissions on a server. There are fixed server roles and user-defined server roles.

Fixed server roles are predefined roles with specific permissions. Examples of fixed server roles include sysadmin, serveradmin, and securityadmin.

User-defined server roles can be created and customized according to specific requirements. You can assign logins to user-defined server roles and grant or deny permissions to these roles.

*Examples of fixed server roles:*

- sysadmin: Members have full administrative privileges on the server.
- securityadmin: Members manage logins and their properties and can grant or revoke server-level and database-level permissions.

Example of creating a user-defined server role and granting permissions:

```
CREATE SERVER ROLE devops;
GRANT ALTER ANY DATABASE TO devops;
ALTER SERVER ROLE devops ADD MEMBER [win10b\winuser01];
```

Server logins and server roles play crucial roles in SQL Server security, ensuring the authentication and authorization of users while managing their access to server resources.

## Principals (Part-2): Database Users, Database Roles

### Database Users:

After setting up server-level logins, you can create database users that map back to those logins, whether they're Windows or SQL Server logins.
It's also possible to create database users that do not map to logins, which are typically used for contained databases, impersonation, or development and testing.
SQL Server provides the CREATE USER statement to create database users, and this statement should be executed within the context of the database where the user is being defined.

```
USE Adventureworks;
CREATE USER [win10b\winuser01];
GRANT ALTER ON SCHEMA::Sales TO [win10b\winuser01];
```

In this example, the user "winuser01" is based on the "win10b\winuser01" login.

If you want to create a user with a different name, you must include the FOR LOGIN or FROM LOGIN clause.

```
CREATE USER winuser03 FOR LOGIN [win10b\winuser01];
GRANT ALTER ON SCHEMA::Sales TO winuser03;
```

Only one user can be created in a database per login.

If you want to try out both of these statements, you'll need to drop the first user before creating the second.

The examples mentioned also include a GRANT statement that assigns the ALTER permission to the user on the Sales schema, allowing the user to alter any object within that schema.

When granting a permission on a specific object, the type of object and its name should be specified, separated by the scope qualifier (double colons).

In some GRANT statements, the securable is implied and doesn't need to be specified.

## Database Roles:

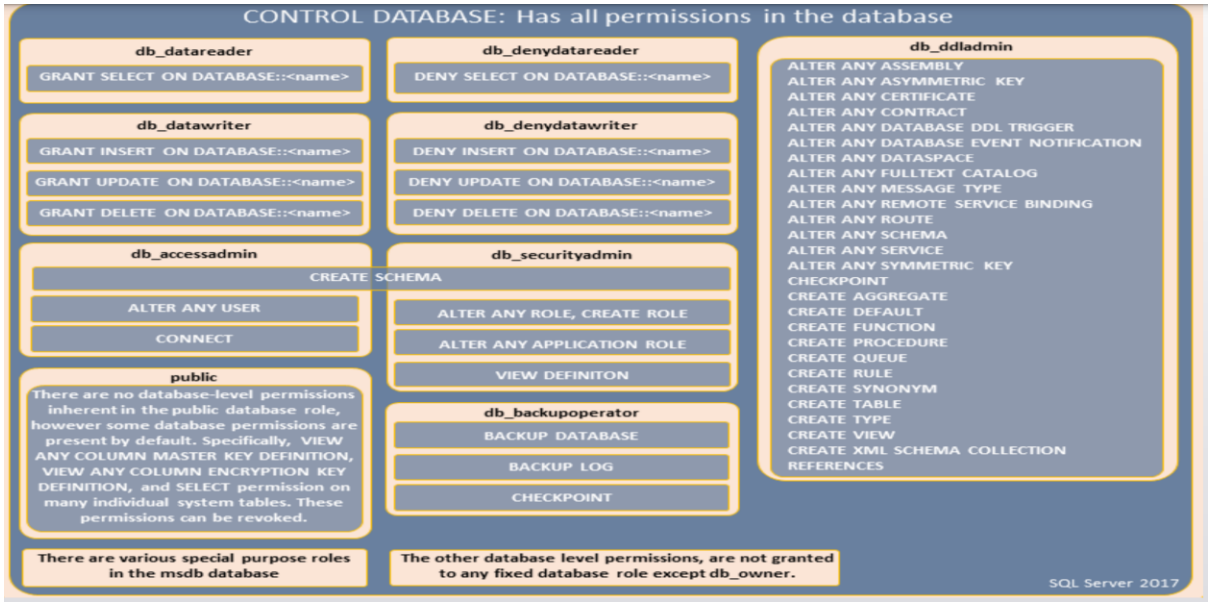
- A database role is a group of users that share a common set of database-level permissions.
- SQL Server supports both fixed and user-defined database roles, similar to server roles.
- Database-level roles have a database-wide scope.
- Fixed-database roles are defined at the database level and exist in each database.
- There are also special-purpose database roles in the msdb database.
- Members of the db_owner database role can manage fixed-database role membership.
- Any database account and other SQL Server roles can be added to database-level roles.
- Server-level permissions cannot be granted to database roles.
- Logins and other server-level principals (such as server roles) cannot be added to database roles.

### *Fixed Database Roles:*

The following are some of the fixed-database roles:

**db_owner**: Members can perform all configuration and maintenance activities on the database and can also drop the database in SQL Server.

- **db_securityadmin**: Members can modify role membership for custom roles only and manage permissions. Their actions should be monitored as they can potentially elevate their privileges.
- **db_accessadmin**: Members can add or remove access to the database for Windows logins, Windows groups, and SQL Server logins.
- **db_backupoperator**: Members can back up the database.
- **db_ddladmin**: Members can run any Data Definition Language (DDL) command in a database.
- **db_datawriter**: Members can add, delete, or change data in all user tables.
- **db_datareader**: Members can read all data from all user tables and views. User objects can exist in any schema except sys and INFORMATION_SCHEMA.
- **db_denydatawriter**: Members cannot add, modify, or delete any data in the user tables within a database.
- **db_denydatareader** Members of the db_denydatareader fixed database role cannot read any data from the user tables and views within a database.




To set up a user-defined database role, you must create the role, grant permissions to the role, and add members to the role (or add members and then grant permissions).

CREATE ROLE dbdev;
GRANT SELECT ON DATABASE::WideWorldImporters TO dbdev;
ALTER ROLE dbdev ADD MEMBER [win10b\winuser01]; ALTER ROLE dbdev ADD MEMBER sqluser01;

## Permissions (GRANT, DENY, REVOKE):

### Permissions:

- SQL Server provides the ability to grant users the access they need at the required level through the GRANT, DENY, and REVOKE statements.
- SQL Server offers three T-SQL statements for working with permissions: GRANT, DENY, and REVOKE.
- The GRANT statement allows granting access to specific securables.
- The DENY statement prevents principals from accessing specific securables and overrides any granted permissions.
- The REVOKE statement removes previously granted permissions on specific securables.
- Permissions are cumulative, meaning the user receives all permissions granted to the database user and its associated login.
- If the user is assigned to a database role or the login is assigned to a server role, the user also receives the role permissions.
- Permissions are transitive, based on the hierarchical nature of server, database, and schema securables.
- For example, granting the UPDATE permission to a user for a specific database also grants the UPDATE permission on all schemas and schema objects within that database.

*GRANT Permissions:*

- Some permissions cover multiple permissions under a single name.
- An example is the CONTROL permission, which includes permissions like INSERT, UPDATE, DELETE, EXECUTE, and others.

```
GRANT CONTROL ON SCHEMA::Sales TO sqluser01;
```

- After granting the CONTROL permission, you can use the fn_my_permissions function to view the effective permissions for the user on the Sales schema.
- The SELECT permission was granted to the user at the database level through the dbdev role and at the schema level as part of the CONTROL permission.

*DENY Permissions:*

- Permissions can be denied on securables.
- This is useful when granting permissions at a higher level but wanting to prevent them from extending to specific child objects.

```
DENY CONTROL ON OBJECT::Sales.BuyingGroups TO sqluser01;
```

- Denying the CONTROL permission also denies all permissions that are part of CONTROL, including the SELECT permission.
- The DENY permission takes precedence over all granted permissions, regardless of where permissions are granted or denied in the object hierarchy.
- Denying permissions on one object does not affect other objects unless they are child objects.

*REVOKE Permissions:*

- The REVOKE statement is used to roll back previously granted permissions.

```
REVOKE CONTROL ON SCHEMA::Sales TO sqluser01;
```

- After revoking the CONTROL permission, the fn_my_permissions function can be used to view the effective permissions for the user on the Sales schema.
- When working with permissions, it's important to distinguish between the DENY statement and the REVOKE statement to avoid unintended consequences.
- Users may receive permissions from multiple sources, so careful consideration is necessary.

*Query to check permissions given to db user/role*

```
-- View permissions for a specific database user

SELECT
   dp.class_desc AS 'Permission Class',
   CASE
      WHEN dp.class_desc = 'DATABASE' THEN OBJECT_NAME(dp.major_id)
      WHEN dp.class_desc = 'OBJECT_OR_COLUMN' THEN OBJECT_NAME(dp.major_id) + '.' +
COL_NAME(dp.major_id, dp.minor_id)
      ELSE ''
   END AS 'Securable',
   dp.permission_name AS 'Permission',
   dp.state_desc AS 'Permission State',
   dp.type_desc AS 'Permission Type',
   USER_NAME(dpr.grantee_principal_id) AS 'Grantee',
```

```
    USER_NAME(dpr.grantor_principal_id) AS 'Grantor'
FROM
    sys.database_permissions AS dp
JOIN
    sys.database_principals AS dpr ON dp.grantee_principal_id = dpr.principal_id
WHERE
    dpr.name = 'YourDatabaseUser'
ORDER BY
    dp.class_desc, dp.major_id, dp.minor_id;

-- View permissions for a specific database role
SELECT
    dp.class_desc AS 'Permission Class',
    CASE
        WHEN dp.class_desc = 'DATABASE' THEN OBJECT_NAME(dp.major_id)
        WHEN dp.class_desc = 'OBJECT_OR_COLUMN' THEN OBJECT_NAME(dp.major_id) + '.' +
COL_NAME(dp.major_id, dp.minor_id)
        ELSE ''
    END AS 'Securable',
    dp.permission_name AS 'Permission',
    dp.state_desc AS 'Permission State',
    dp.type_desc AS 'Permission Type',
    USER_NAME(dpr.grantee_principal_id) AS 'Grantee',
    USER_NAME(dpr.grantor_principal_id) AS 'Grantor'
FROM
    sys.database_permissions AS dp
JOIN
    sys.database_principals AS dpr ON dp.grantee_principal_id = dpr.principal_id
JOIN
    sys.database_role_members AS drm ON dp.grantee_principal_id =
drm.member_principal_id
JOIN
    sys.database_principals AS dr ON drm.role_principal_id = dr.principal_id
WHERE
    dr.name = 'YourDatabaseRole'
ORDER BY
    dp.class_desc, dp.major_id, dp.minor_id;
```

# MSSQL Server Agent Management

## Why we need SQL Server Agent:

- Automation: SQL Server Agent allows us to automate the execution of repetitive scripts and tasks.
- Timely execution: Scripts can be scheduled to run at specific times, ensuring timely execution without human intervention.
- Job management: SQL Server Agent provides a centralized platform to manage and monitor jobs.
- History and monitoring: The execution history of jobs can be viewed, allowing us to track successful and failed executions.
- Restart and recovery: In case of failure, jobs can be manually restarted, ensuring continuity of operations.
- Flexibility: Jobs can be scheduled, triggered by specific events, or run on demand.
- Autostart: SQL Server Agent service is disabled by default but can be configured to start automatically during SQL Server installation.

## SQL Server Agent Components:

1. Jobs: A job is a defined set of actions that SQL Server Agent performs.
2. Schedules: Schedules determine when a job should run.
3. Alerts: Alerts provide automatic responses to specific events.
4. Operators: Operators are individuals responsible for the maintenance of SQL Server instances. Contact information is defined for each operator.

## Security for SQL Server Agent Administration:

- Fixed database roles: SQL Server Agent uses the fixed database roles SQLAgentUserRole, SQLAgentReaderRole, and SQLAgentOperatorRole in the msdb database to control access for non-sysadmin users.
- Subsystems: Subsystems define the functionality available to job steps and help ensure minimal permissions for each task.
- Proxies: Proxies are used to manage security contexts in SQL Server Agent. They allow members of the sysadmin fixed server role to create and use proxies for job steps. Proxies can be shared across multiple job steps.

## SQL Server Agent Jobs and Schedules

### SQL Server Agent Jobs:

- A job is a specified series of actions that SQL Server Agent performs.
- Jobs are used to define administrative tasks that can be executed one or more times and monitored for success or failure.
- Jobs can run on a local server or multiple remote servers.
- There are different ways to run jobs:
    - According to one or more schedules.
    - In response to one or more alerts.
    - By executing the sp_start_job stored procedure.
- Each action within a job is called a job step.
- Examples of job steps include running a Transact-SQL statement, executing an SSIS package, or issuing a command to an Analysis Services server.

- Job steps are managed as part of a job.
- Each job step runs in a specific security context.
- For Transact-SQL job steps, the EXECUTE AS statement is used to set the security context.
- For other types of job steps, a proxy account is used to set the security context.

### Schedules:

- A schedule specifies when a job runs.
- Multiple schedules can be applied to the same job.
- Schedules can define the following conditions for the job's execution time:
    - Whenever SQL Server Agent starts.
    - Whenever CPU utilization of the computer is at a level defined as idle.
    - One-time execution at a specific date and time.
    - Recurring schedule.

To create a job in Microsoft SQL Server, you can follow these steps:

1. Launch SQL Server Management Studio (SSMS) and connect to the SQL Server instance where you want to create the job.
2. Expand the "SQL Server Agent" node in the Object Explorer pane.
3. Right-click on the "Jobs" folder and select "New Job" from the context menu. The "New Job" dialog box will open.
4. In the "General" tab of the "New Job" dialog box, enter a unique name for the job in the "Name" field.
5. Optionally, provide a description for the job in the "Description" field to help identify its purpose.
6. Specify the owner of the job by selecting a login or user from the "Owner" drop-down list. By default, the current user is selected.
7. Switch to the "Steps" tab. Click on the "New" button to add a new step to the job.
8. In the "New Job Step" dialog box, enter a unique name for the step in the "Step name" field.
9. Provide a description for the step in the "Step description" field (optional).
10. In the "Type" section, choose the appropriate type of action for the step. It can be executing a Transact-SQL script, running an Integration Services package, executing a PowerShell script, etc.
11. Enter the specific details related to the selected action type. For example, if you selected "Transact-SQL script," you can enter the T-SQL code in the "Command" field.
12. Configure other options as needed, such as the database context, output file, advanced options, and error handling.
13. Click on the "OK" button to save the step and return to the "New Job" dialog box.
14. Switch to the "Schedules" tab to define a schedule for the job. Click on the "New" button to create a new schedule.
15. In the "New Job Schedule" dialog box, provide a name and description for the schedule.
16. Specify the frequency of the schedule (e.g., one-time, daily, weekly, etc.) and set the desired timing details.
17. Configure additional options like the duration, enabled/disabled status, and any specific advanced settings.
18. Click on the "OK" button to save the schedule and return to the "New Job" dialog box.
19. Review the summary of the job settings in the "Notifications" and "Target Servers" tabs. Configure these tabs if necessary.

20. Click on the "OK" button to create the job.

## SQL Server Agent Alerts:

- An alert is an automatic response to a specific event, such as a job starting or system resources reaching a threshold.
- Alerts are defined with conditions that trigger their occurrence.
- Alerts can respond to SQL Server events, performance conditions, or Windows Management Instrumentation (WMI) events.
- Actions performed by alerts can include notifying one or more operators and running a job.
- SQL Server Agent reads the application log, compares events to defined alerts, and triggers automated responses.

To create a job in Microsoft SQL Server, you can follow these steps:

1. Launch SQL Server Management Studio (SSMS) and connect to the SQL Server instance where you want to create the job.
2. Expand the "SQL Server Agent" node in the Object Explorer pane.
3. Right-click on the "Jobs" folder and select "New Job" from the context menu. The "New Job" dialog box will open.
4. In the "General" tab of the "New Job" dialog box, enter a unique name for the job in the "Name" field.
5. Optionally, provide a description for the job in the "Description" field to help identify its purpose.
6. Specify the owner of the job by selecting a login or user from the "Owner" drop-down list. By default, the current user is selected.
7. Switch to the "Steps" tab. Click on the "New" button to add a new step to the job.
8. In the "New Job Step" dialog box, enter a unique name for the step in the "Step name" field.
9. Provide a description for the step in the "Step description" field (optional).
10. In the "Type" section, choose the appropriate type of action for the step. It can be executing a Transact-SQL script, running an Integration Services package, executing a PowerShell script, etc.
11. Enter the specific details related to the selected action type. For example, if you selected "Transact-SQL script," you can enter the T-SQL code in the "Command" field.
12. Configure other options as needed, such as the database context, output file, advanced options, and error handling.
13. Click on the "OK" button to save the step and return to the "New Job" dialog box.
14. Switch to the "Schedules" tab to define a schedule for the job. Click on the "New" button to create a new schedule.
15. In the "New Job Schedule" dialog box, provide a name and description for the schedule.
16. Specify the frequency of the schedule (e.g., one-time, daily, weekly, etc.) and set the desired timing details.
17. Configure additional options like the duration, enabled/disabled status, and any specific advanced settings.
18. Click on the "OK" button to save the schedule and return to the "New Job" dialog box.
19. Review the summary of the job settings in the "Notifications" and "Target Servers" tabs. Configure these tabs if necessary.
20. Click on the "OK" button to create the job.

## SQL Server Agent Operators:

- Operators define the contact information for individuals responsible for maintaining SQL Server instances.
- An operator can be assigned to one individual or shared among multiple individuals.
- Operators do not contain security information and do not define security principals.
- SQL Server can notify operators via email, pager (through email), or net send.
- To send notifications through email or pagers, SQL Server Agent must be configured to use Database Mail.
- Operators have attributes like name and contact information.

To set up a SQL Server database operator, you can follow these steps:

1. Launch SQL Server Management Studio (SSMS) and connect to the SQL Server instance where you want to create the operator.
2. Expand the "SQL Server Agent" node in the Object Explorer pane.
3. Expand the "Operators" folder and right-click on it. Select "New Operator" from the context menu. The "New Operator" dialog box will open.
4. In the "General" tab of the "New Operator" dialog box, enter a unique name for the operator in the "Name" field.
5. Optionally, provide a description for the operator in the "Description" field to help identify their role or responsibilities.
6. Specify the operator's email address in the "E-mail name" field. This is the email address to which notifications will be sent.
7. Select the method by which the operator will receive notifications. You can choose one or more of the following methods:
    a. E-mail: Select the "E-mail" checkbox to enable email notifications. Enter the email address in the "E-mail name" field.
    b. Pager (e-mail): Select the "Pager (e-mail)" checkbox to enable pager notifications. Enter the pager email address in the "E-mail name" field.
    c. Net send: Select the "Net send" checkbox to enable net send notifications. This sends messages to the operator's computer using the "net send" command.
8. Configure the schedule for the operator. You can specify the time frame during which the operator is available to receive notifications. By default, the operator is available at all times.
9. Click on the "OK" button to save the operator.

## Database Mail:

- Database Mail is a feature in SQL Server for sending email messages from the Database Engine.
- It allows database applications to send emails containing query results or files from any resource on the network.
- Benefits of using Database Mail include reliability, scalability, security, and supportability.
- Database Mail utilizes a queued architecture with service broker technologies for sending email messages.
- Messages are inserted into the mail queue, read by the external Database Mail process, and sent to email servers.

- Database Mail components include configuration and security, messaging, the Database Mail executable, and logging and auditing.

## Configuring Agent to use Database Mail:

- SQL Server Agent can be configured to use Database Mail for alert notifications and job completion notifications.
- Individual job steps within a job can also use Database Mail without configuring SQL Server Agent.
- Configuration involves setting up profiles, accounts, and linking them to SQL Server Agent.

To set up Database Mail and configure SQL Server Agent to use it for sending notifications, you can follow these steps:

1. Launch SQL Server Management Studio (SSMS) and connect to the SQL Server instance where you want to configure Database Mail.
2. Expand the "Management" folder in the Object Explorer pane.
3. Right-click on "Database Mail" and select "Configure Database Mail" from the context menu. The "Database Mail Configuration Wizard" will open.
4. In the "Select Configuration Task" step, choose the option "Set up Database Mail by performing the following tasks".
5. In the " Configure Database Mail" step, click on the "Next" button.
6. In the "Account Name" step, click on the "Add" button to create a new mail account.
7. In the "New Database Mail Account" dialog box, provide a name for the mail account in the "Account name" field.
8. Specify the email address and display name for the mail account in the respective fields.
9. Configure the mail server details. Enter the SMTP server name and port number. If your mail server requires authentication, check the appropriate options and provide the necessary credentials.
10. Test the mail account settings by clicking on the "Test" button. Ensure that the test email is successfully sent and received.
11. Once the account settings are verified, click on the "OK" button to save the mail account.
12. In the "Profile Name" step, click on the "Add" button to create a new mail profile.
13. In the "New Database Mail Profile" dialog box, enter a name for the mail profile in the "Profile name" field.
14. Select the mail account created in the previous steps from the drop-down list.
15. Click on the "Add" button to add operators to the mail profile. Select the desired operators who will receive notifications via email.
16. Click on the "Finish" button to save the mail profile.
17. Close the "Database Mail Configuration Wizard".
18. To configure SQL Server Agent to use Database Mail, right-click on "SQL Server Agent" in the Object Explorer pane and select "Properties".
19. In the "SQL Server Agent Properties" dialog box, go to the "Alert System" tab.
20. Check the "Enable mail profile" checkbox and select the mail profile created earlier from the drop-down list.
21. Click on the "OK" button to save the changes.

## SQL Server Agent Job Activity Monitor:

- The Job Activity Monitor in SQL Server Management Studio allows viewing the sysjobactivity table.
- It provides information on jobs, allowing filtering, sorting, and viewing job properties and history.
- The Job Activity Monitor enables starting and stopping jobs, refreshing information, and managing job activity.
- It helps identify scheduled jobs, the outcome of recently executed jobs, and currently running or idle jobs.
- The previous session in the Job Activity Monitor can help identify jobs interrupted by unexpected SQL Server Agent service failure.
- The sp_help_jobactivity stored procedure can be used to view job activity for the current session.

To use the SQL Server Agent Job Activity Monitor in SQL Server Management Studio (SSMS), you can follow these steps:

1. Launch SQL Server Management Studio (SSMS) and connect to the SQL Server instance where you want to monitor the job activity.
2. In the Object Explorer pane, expand the "SQL Server Agent" node.
3. Right-click on the "Job Activity Monitor" node and select "View Job Activity" from the context menu. The Job Activity Monitor window will open.
4. In the Job Activity Monitor window, you will see a grid that displays information about the currently running jobs, as well as recently completed jobs.
5. By default, the grid shows all jobs on the server. You can apply filters to limit the jobs displayed based on various criteria such as job name, status, or owner. To apply filters, click on the "Filter" button in the toolbar.
6. To start or stop a job, select the job in the grid and click on the corresponding toolbar button or right-click on the job and select the appropriate option from the context menu.
7. To view detailed information about a specific job, select the job in the grid and click on the "View Job Properties" button in the toolbar or right-click on the job and select "View Job Properties".
8. To view the history of a specific job, select the job in the grid and click on the "View Job History" button in the toolbar or right-click on the job and select "View Job History".
9. To manually refresh the information in the Job Activity Monitor, click on the "Refresh" button in the toolbar. You can also set an automatic refresh interval by clicking on the "View Refresh Settings" button in the toolbar.
10. To close the Job Activity Monitor, click on the "Close" button in the toolbar.

# Advanced Topics

## High Availability

High Availability and its Types

What is High Availability?

- In today's IT-based world, the success and growth of businesses depend on the availability of critical systems that run it.

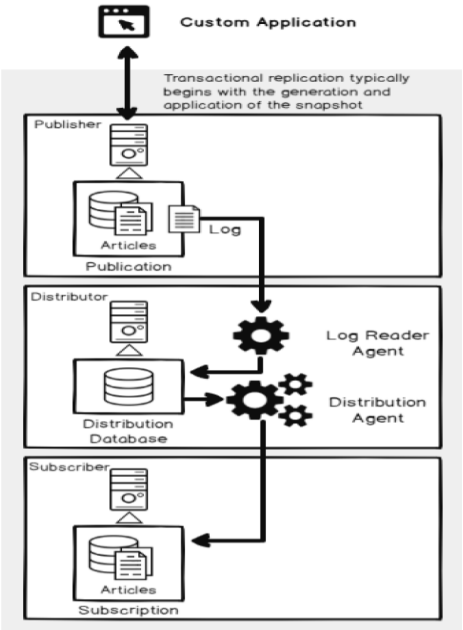
- During Service Level Agreement (SLA) discussions, factors such as Recovery Time Objective (RTO) and Recovery Point Objective (RPO) are considered.
- Any downtime in financial or tracking systems is not acceptable.
- The availability of business applications relies on the availability of the data they process and store.
- High availability ensures minimum downtime for database systems during crashes, failures, or disasters.
- The availability of SQL Server instances or databases is measured by the number of 9's in the availability percentage and the total annual downtime.
- Disaster recovery involves bringing the system online in a secondary site when the primary datacenter experiences a catastrophic event.

| Number of 9's | Availability Percentage | Total Annual Downtime |
|---|---|---|
| 2 | 99% | 3 days, 15 hours |
| 3 | 99.9% | 8 hours, 45 minutes |
| 4 | 99.99% | 52 minutes, 34 seconds |
| 5 | 99.999% | 5 minutes, 15 seconds |

High Availability Types:

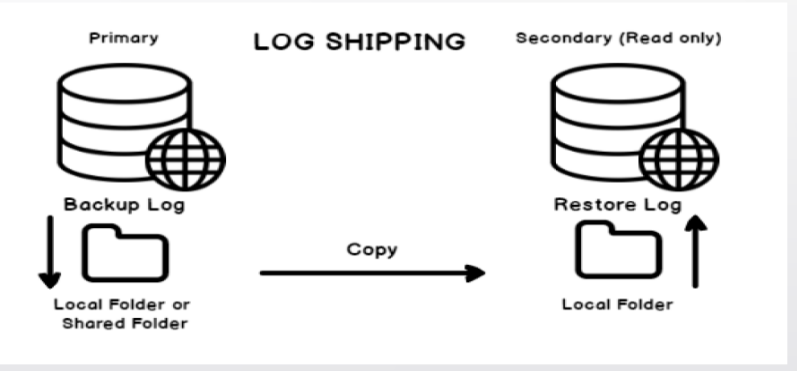
1. Replication:

- Source data is copied to the destination through replication agents using object-level technology.
- Terminology:
  - Publisher: Source server
  - Distributor: Optional server storing replicated data
  - Subscriber: Destination server



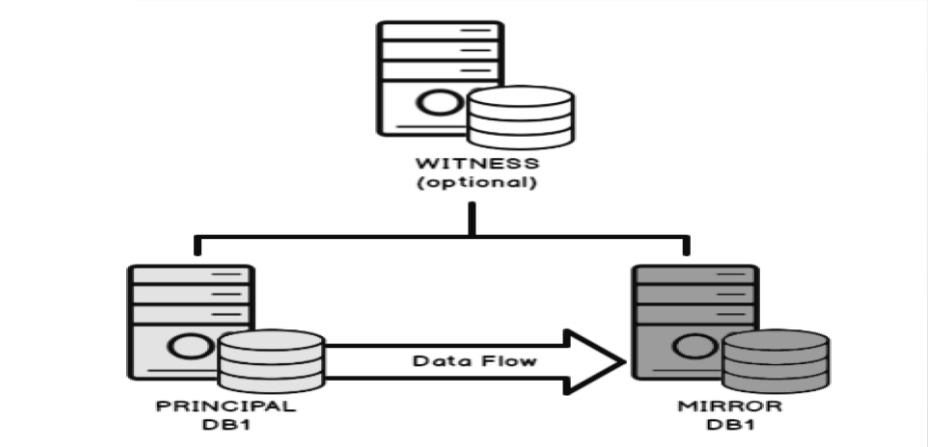
2. Log Shipping:

- Source data is copied to the destination through transaction log backup jobs using database-level technology.
- Terminology:
    - Primary server: Source server
    - Secondary server: Destination server
    - Monitor server: Optional server for monitoring log shipping status



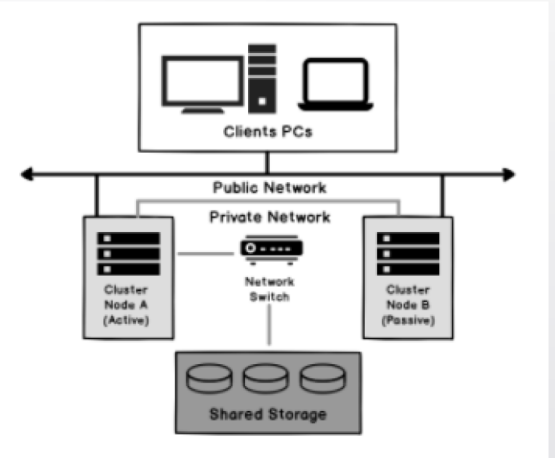

3. Mirroring:

- Primary data is copied to a secondary server on a transaction basis using mirroring endpoint and port number.
- Database-level technology.
- Terminology:
    - Principal server: Source server
    - Mirror server: Destination server
    - Witness server: Optional server for automatic failover



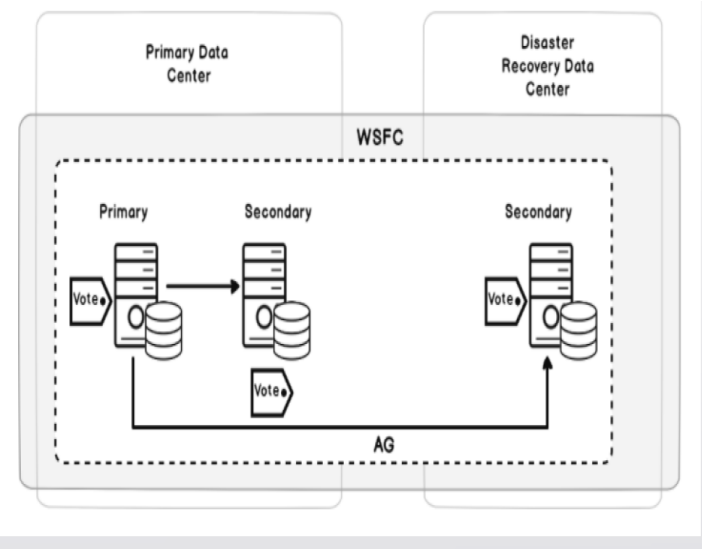

4. Clustering:

- Data is stored in a shared location accessible by both primary and secondary servers based on server availability.
- Instance-level technology. Windows Clustering setup with shared storage is required.
- Terminology:
    - Active node: Server running SQL Services
    - Passive node: Server not running SQL Services




5. AlwaysON Availability Groups:

- Primary data is copied to a secondary server through a network transaction basis.
- Group of database-level technology. Windows Clustering setup without shared storage is required.
- Terminology:
    - Primary replica: Source server
    - Secondary replica: Destination server




These high availability types provide different solutions to ensure the availability, reliability, and continuity of critical systems and data.

## What Is Replication and Transactional Replication

What is Replication:

- SQL Server replication is a technology used to copy and distribute data and database objects from one database to another while maintaining consistency and integrity.
- Replication can be continuous or scheduled to run at predetermined intervals.
- It supports various data synchronization approaches such as one-way, one-to-many, many-to-one, and bi-directional, allowing multiple datasets to stay in sync.

Types of Replication:

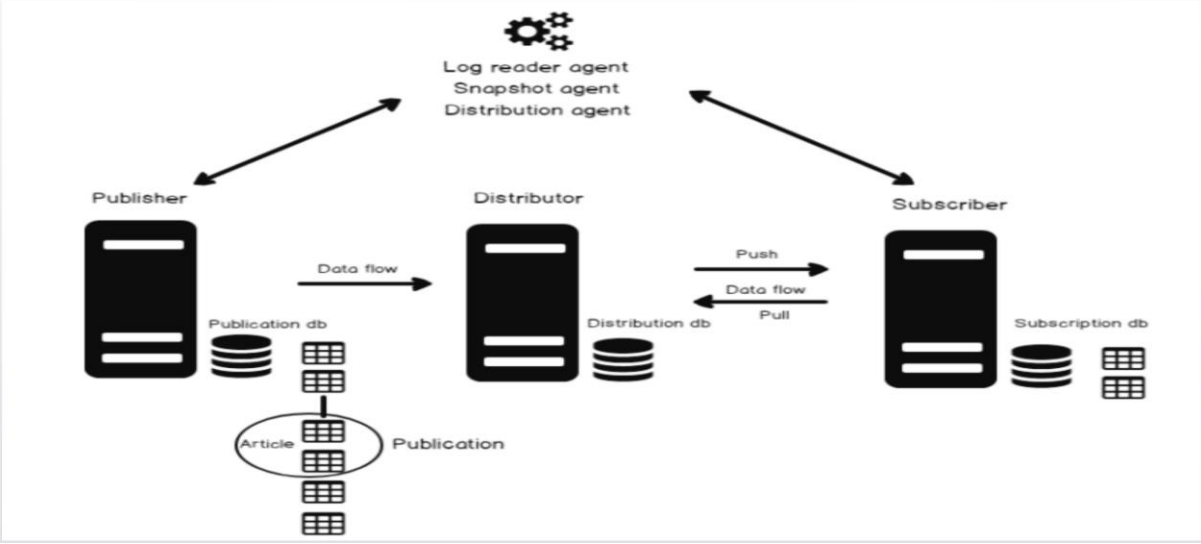
1. Transactional Replication

2. Merge Replication
3. Snapshot Replication

Transactional Replication:

- Transactional Replication is a real-time, database-level, high availability solution.
- It involves a primary server (Publisher) distributing database tables or selected tables (articles) to one or more secondary servers (Subscribers).
- It relies on synchronization with SQL Server Transaction Logs.
- Components of Transactional Replication:
    - Publisher: The database instance that makes data available for replication.
    - Publication Database: The database containing objects designated for replication (articles).
    - Publication: A logical collection of articles from a database.
    - Articles: The basic unit of SQL Server Replication, consisting of tables, stored procedures, and views.
    - Distributor: The database acting as a storehouse for replication-specific data associated with one or more Publishers.
    - Distribution Databases: Databases holding article details, replication metadata, and data.
    - Subscriber: A database instance consuming replication data from a publication.
    - Subscriptions: Requests for copies of a publication delivered to a Subscriber.
    - Subscription Databases: Target databases in a replication model.
    - Replication Agents: Standalone programs responsible for replication tasks, including Snapshot Agent, Distribution Agent, and Log Reader Agent.
- Snapshot Agent: Prepares the starting snapshot file containing the database objects to be replicated.
- Distribution Agent: Copies the initial snapshot and cumulative logs to the subscribers.
- Log Reader Agent: Monitors the SQL Server transaction log in the publisher database and copies transactions to the distribution database, which are then sent to the subscribers.

Transactional Replication provides a reliable and real-time method for replicating data and maintaining high availability across multiple database servers.



## Steps to setup transactional replication

To set up Transactional Replication in SQL Server using SQL Server Management Studio (SSMS), you can follow these step-by-step instructions:

Step 1: Prepare the Environment

1. Ensure that you have two SQL Server instances: a Publisher (source) and a Subscriber (destination).
2. Make sure the instances are accessible to each other and the necessary firewall rules are configured.

Step 2: Configure Distribution Database

1. Connect to the Publisher instance using SSMS.
2. Right-click on the "Replication" folder and select "Configure Distribution..."
3. Follow the wizard to configure the Distribution database and select the appropriate options based on your environment.

Step 3: Configure Publisher

1. Right-click on the "Local Publications" folder under the Publisher instance in SSMS and select "New Publication."
2. Follow the wizard to select the database(s) you want to replicate and choose the Transactional Publication type.
3. Configure article options, such as tables, views, and stored procedures, to include in the publication.
4. Choose the appropriate options for filtering, snapshot generation, and publication schedule.
5. Complete the wizard and generate the initial snapshot.

Step 4: Configure Subscriber

1. Connect to the Subscriber instance using SSMS.
2. Right-click on the "Local Subscriptions" folder under the Subscriber instance and select "New Subscription."
3. Select the publication you created in Step 3 and choose the appropriate subscription type (push or pull).
4. Configure the subscription database and folder location for the initial snapshot.
5. Specify the synchronization schedule and agent security settings.
6. Complete the wizard to create the subscription.

Step 5: Initialize and Synchronize the Subscription

1. Right-click on the newly created subscription under the Subscriber instance and select "View Synchronization Status."
2. Click on the "Start" button to initialize the subscription and begin the synchronization process.
3. Monitor the synchronization progress and resolve any errors or conflicts that may occur.

Step 6: Test the Replication

1. Make changes to the replicated data on the Publisher and verify that the changes are propagated to the Subscriber.
2. Perform additional testing to ensure data consistency and integrity between the Publisher and Subscriber databases.

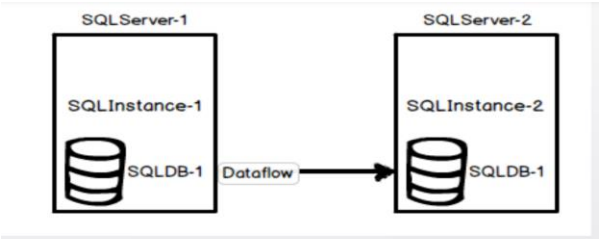
Congratulations! You have successfully set up Transactional Replication in SQL Server using SSMS. Remember to monitor the replication process regularly and address any issues that may arise to ensure the ongoing synchronization of data.

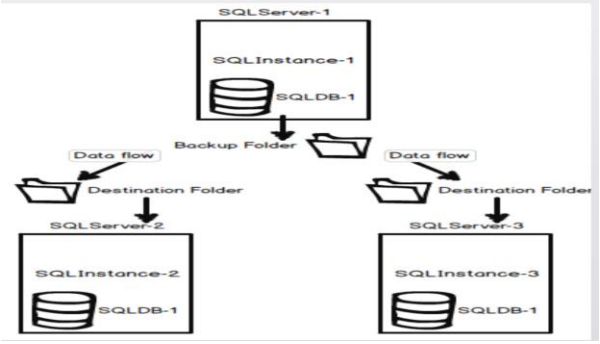
## Log Shipping:

- Log Shipping is a disaster recovery solution in SQL Server that involves automatically sending transaction log backups from a primary (source) database to one or more secondary (destination) databases.
- It is used to maintain a warm standby copy of a database for disaster recovery purposes.
- Log Shipping operates at the database level and copies the transaction log backups and applies them to the secondary database(s) on a scheduled basis.

Components of Log Shipping:

1. Primary Database:
    - The primary database is the source database that generates transaction log backups.
    - It is the database that you want to protect and replicate to the secondary server(s).
2. Secondary Database(s):
    - The secondary database(s) is the destination database(s) that receive and apply the transaction log backups.
    - Secondary databases are set to either Standby or No-Recovery mode, allowing them to be read-only for reporting or backup purposes.
    - There are two available modes and they are related to the state in which the secondary, log shipped, SQL Server database will be:
      • Standby mode – the database is available for querying and users can access it, but in read-only mode. The database is not available only while the restore process is running. Users can be forced to disconnect when the restore job commence . The restore job can be delayed until all users disconnect themselves
      • Restore mode – the database is not accessible
3. Log Shipping Jobs:
    - Log Shipping relies on a set of SQL Server Agent jobs to perform the necessary tasks.
    - Backup job: Takes transaction log backups on the primary database at specified intervals.
    - Copy job: Copies the transaction log backups from the primary to the secondary server(s).
    - Restore job: Restores the copied transaction log backups on the secondary database(s).
4. Monitor Server (Optional):
    - A monitor server can be configured to centrally monitor and report the status of log shipping across multiple primary-secondary pairs.
    - The monitor server tracks the history and status of the log shipping jobs and alerts administrators if any issues occur.




Setting up Log Shipping:

1. Configure Backup Settings:
    - Set up the primary database to perform regular transaction log backups.
    - Specify the backup location and schedule for the transaction log backups.
2. Configure Copy Settings:
    - Set up the secondary server(s) and database(s) to receive the transaction log backups.
    - Specify the destination folder for copied transaction log backups.
3. Configure Restore Settings:
    - Configure the secondary database(s) to restore the transaction log backups.
    - Choose the restore mode (Standby or No-Recovery) for the secondary database(s).
4. Initialize Log Shipping:
    - Take a full backup of the primary database and restore it on the secondary server(s).
    - Manually restore the latest transaction log backup on the secondary database(s) to bring them up to date.
5. Monitor Log Shipping:
    - Regularly monitor the log shipping jobs and check the status of transaction log backups, copy, and restore operations.
    - Address any errors or issues that may occur during the process.

Another Setup way:

The database log shipping setup needs to be initiated from the principal server using the SQL Server Management Studio wizard.

The first step defines transaction log backup settings:

- A network path to the backup
- How long backup files should be kept before deleting
- An alert if no backup is taken
- The backup job itself
- Schedule of the job
- Schedule type
- Frequency
- Duration

The next step defines secondary databases which involve choosing the secondary SQL Server instance and secondary database.
The full database backup, from the primary database, must be restored on the secondary server before log shipping commences.
After initializing the secondary database you must define the copy folder where the transaction log backups from the primary server will be stored.
The final step involves choosing from two available modes:
The No recovery – Restore mode and Standby mode.
You can also delay the restoring process and set up an alert if no restore occurs within the specified time.
Once the log shipping is ready for use, it will run in the background, and if the problem occurs the alert will signalize the problem.

Advantages of Log Shipping:

- Provides a warm standby copy of the primary database for disaster recovery purposes.
- Allows read-only access to the secondary database(s) for reporting or backup purposes.
- Simple and straightforward setup and maintenance.
- Transaction log backups can be compressed, reducing network bandwidth usage.
- Works well for databases with low to moderate transactional activity.

Limitations of Log Shipping:

- Recovery time is longer compared to other high availability solutions like Always On Availability Groups.
- Manual intervention is required to bring the secondary database(s) online during a disaster.
- Log Shipping does not support automatic failover.
- The secondary database(s) are read-only and cannot be used for write operations.

Remember to practice setting up and configuring Log Shipping in a test environment to gain hands-on experience and reinforce your understanding of the concepts.

## SQL Server Encryption and Encryption Types:

### Encryption:

- Data protection is crucial for regulatory compliance and maintaining the trust of clients and business partners.
- Encryption is the process of obfuscating data using a key or password, rendering it useless without the corresponding decryption key.
- It enhances security by limiting data loss even if access controls are bypassed.
- Encryption does not solve access control problems but adds an extra layer of security.

Encryption Types:

#### *SSL Transport Encryption:*

- Uses Secure Sockets Layer (SSL) to encrypt traffic between the SQL Server instance and client application.
- SSL protects data in transit, and the client can validate the server's identity using its certificate.
- Available in all supported versions and editions of SQL Server.
- Requires installing a certificate on the SQL Server, preferably from your own enterprise certification authority (CA).

*SQL Server Transparent Data Encryption (TDE):*

- Protects data at rest by encrypting database data and log files on disk.
- Works transparently with existing applications without requiring any changes.
- Uses real-time encryption at the page level, encrypting pages before they are written to disk and decrypting them when read into memory.
- Available only in the Enterprise editions of SQL Server.

*Column/Cell-Level Encryption:*

- Available in all editions of SQL Server.
- Enables encryption on columns containing sensitive data, encrypting the data on disk and keeping it encrypted in memory until decrypted using the DECRYPTBYKEY function.
- Requires modifications to client applications to work with cell-level encryption.
- Provides encryption but relies on user context and function usage for decryption.

*Backup Encryption:*

- Similar to TDE, but encrypts SQL backups rather than active data and log files.
- Available in SQL Server 2014 and later versions.
- Supports AES 128, AES 192, AES 256, or Triple DES encryption, using a certificate or asymmetric key stored in EKM.
- Backup Encryption can be used in conjunction with TDE, requiring separate certificates or keys.
- Backing up the certificate or key is essential for restoring data encrypted with Backup Encryption.

*Always Encrypted:*

- Encrypts sensitive data in client applications without revealing encryption keys to the database engine.
- Ensures separation between data owners and data managers.
- Prevents database administrators from accessing sensitive data.
- Encrypts data at rest and can be configured for individual database columns.
- Requires a special driver on client computers.
- Available in SQL Server 2016 and later, but only in Enterprise editions.

## Transparent Data Encryption (TDE):

- TDE is a built-in feature introduced with SQL Server 2008 for encrypting data at rest.
- It protects the physical media that stores data associated with a user database, including data and log files, backups, and snapshots.
- Encrypting data at rest prevents unauthorized access to the data even if the files are accessed.
- TDE encrypts all data in the database, taking an all-or-nothing approach, unlike column-level encryption.
- Enabling TDE is relatively straightforward and does not require changes to the database structure.

TDE Encryption Hierarchy:

- The encryption hierarchy includes Windows Data Protection API (DPAPI) at the top and SQL Server data at the bottom.
- The next level is a certificate, which is created in the master database.
- The Database Master Key (DMK) protects the certificate, and the certificate protects the Database Encryption Key (DEK) in the user database.
- The DEK is specific to TDE and is used to encrypt the data in the user database.

Steps to Implement TDE on a User Database:

1. Create the DMK in the master database if it doesn't already exist.
2. Create a certificate in the master database to secure the DEK.
3. Create the DEK in the user database that needs encryption.
4. Enable TDE on the user database.

TDE Keys Management:

- It is crucial to back up the Service Master Key (SMK), DMK, and certificate used in TDE.
- These keys and certificates should be securely stored in a separate location.
- Backing up certificates and keys ensures their availability for recovery, restoration, or moving encrypted databases.
- It is recommended to back up certificates and keys right after their creation.
- This also applies to the SMK before relying on it to protect the DMKs.

By following proper key management practices and regular backups, the TDE implementation can be effectively maintained and recovered if needed.

### *TDE with Windows Account*

1. Open SQL Server Management Studio and connect to the SQL Server instance.
2. Expand the "Databases" folder, right-click on the database you want to encrypt, and select "Tasks" -> "Encrypt Connections".
3. In the "Encrypt Database" dialog box, click on the "Next" button.
4. Select the option "Use my Windows user account to encrypt the database" and click on the "Next" button. This option allows you to use the Windows account credentials for encryption.
5. Review the summary of the settings and click on the "Next" button.
6. In the "Complete the Wizard" dialog box, review the options and click on the "Finish" button.
7. The encryption process will begin, and you will see the progress in the "Encrypt Database" wizard.
8. Once the encryption process is complete, click on the "Close" button to exit the wizard.
9. To verify that TDE is enabled, right-click on the encrypted database, select "Properties", and navigate to the "Options" page.
10. On the "Options" page, you should see the "Encryption Enabled" property set to "True".

Congratulations! You have successfully enabled Transparent Data Encryption (TDE) for the selected database using SQL Server Management Studio (SSMS). The database and its associated files are now encrypted at rest.

### *TDE with Certificates*

1. Open SQL Server Management Studio and connect to the SQL Server instance.
2. Expand the "Databases" folder, right-click on the database you want to encrypt, and select "Tasks" -> "Encrypt Connections".
3. In the "Encrypt Database" dialog box, click on the "Next" button.
4. Select the option "Create a database master key and a certificate" and click on the "Next" button.
5. In the "Master Key Configuration" dialog box, select the option "Automatically generate a master key" and click on the "Next" button.

6. In the "Certificate Configuration" dialog box, select the option "Create a new certificate" and enter a name for the certificate.
7. Choose a strong password for the certificate and confirm it. Make sure to remember this password as it will be needed for future operations.
8. Specify the certificate expiration date, if desired, or leave it blank for the certificate to never expire.
9. Click on the "Next" button to continue.
10. Review the summary of the settings and click on the "Next" button.
11. In the "Complete the Wizard" dialog box, review the options and click on the "Finish" button.
12. The encryption process will begin, and you will see the progress in the "Encrypt Database" wizard.
13. Once the encryption process is complete, click on the "Close" button to exit the wizard.
14. To verify that TDE is enabled, right-click on the encrypted database, select "Properties", and navigate to the "Options" page.
15. On the "Options" page, you should see the "Encryption Enabled" property set to "True".

## Congratulations!!

You made it till the end☺