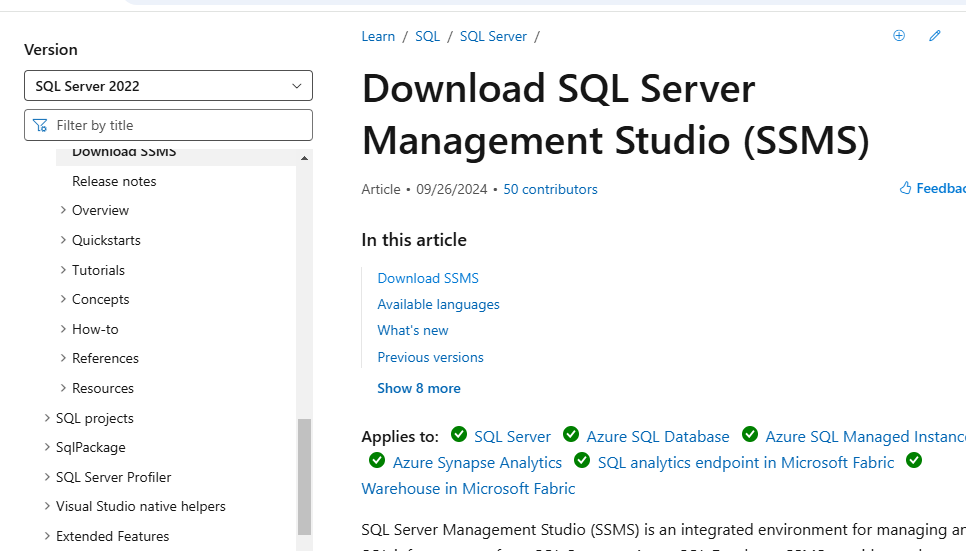
**SQL Server**

* Installation
* Got to the site to download [SQL Server](https://www.microsoft.com/en-in/sql-server/sql-server-downloads). Download developer edition.
* Open the downloaded exe extension and setup SQL. Select *BASIC* installation type.
* Need to restart the system to complete the setup.
* Go to the site to download [SSMS](https://learn.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms?view=sql-server-ver16&redirectedfrom=MSDN#download-ssms) setup.
* Click on download SSMS.



* Open the downloaded SSMS exe extension and complete SQL setup.
* Open the SQL app and set up the connection.

A screenshot of a computer

Description automatically generated

**Database Design:**

**(**[**https://support.microsoft.com/en-us/office/database-design-basics-eb2159cf-1e30-401a-8084-bd4f9c9ca1f5**](https://support.microsoft.com/en-us/office/database-design-basics-eb2159cf-1e30-401a-8084-bd4f9c9ca1f5) **)**

* Database terms:
  + Tables: List of rows and columns.
  + Record: Each row is called a record.
  + Field: Each column is called a field.
* What is a good database?
  + Principles of database design process:
    - Duplicate information (also called redundant data) is bad, because it wastes space and increases the likelihood of errors and inconsistencies.
    - The correctness and completeness of information is important. If the database contains incorrect information, any reports that pull information from the database will also contain incorrect information. As a result, any decisions made that are based on those reports will then be misinformed.

**The Design Process:** The design process consists of following steps:

* **Determine the purpose of the database.**
  + Identify the purpose of the database – its purpose, how you expect to use it, and who will use it.
* **Find and organize the information required**.
  + Identify the items (information) that need to be stored in the table.
  + Each item will be column of the table.
  + A key point to remember is that you should break each piece of information into its smallest useful parts.
  + Try to get the information from the table as required, so that you can add an extra column if any information is pending.
* **Divide the information into tables.**
  + To divide the information into tables, choose the major entities, or subjects.
  + When you design your database, always try to record each fact just once.
  + If there is any data that is repeatedly stored in the table, the create a separate table and store it only once to avoid duplication of data.
  + If we want to change the data, we can just change it at one place instead of changing many records and avoid chance of missing to change at some records.
  + Once you have chosen the subject that is represented by a table, columns in that table should store facts only about the subject.
* **Turn information into columns.**
  + To determine the columns in a table, decide what information you need to track about the subject recorded in the table.
  + Tips for determining the columns:
    - Don’t include calculated data.
    - Store information in its smallest logical parts.
* **Specify primary key.** 
  + The primary key is a column that is used to uniquely identify each row.
  + Each table should include a column or set of columns that uniquely identifies each row stored in the table.
  + Can’t have duplicate values in the primary key column.
  + Primary key can’t be null.
  + Always choose a primary key whose value will not change.
    - Using a primary key that will not change reduces the chance that the primary key might become out of sync with other tables that reference it.
  + A column set to the AutoNumber data type often makes a good primary key.
  + When a primary key employs more than one column, it is also called a ***composite key***.
* **Setup the table relationships.**
  + One-to-many relationship: (e.g.: products (many) and suppliers(one))
    - To represent a one-to-many relationship in your database design, take the primary key on the "one" side of the relationship and add it as an additional column or columns to the table on the "many" side of the relationship.
    - That column become foreign key in “many” side of the relationship table.
    - A foreign key is another table’s primary key.
  + Many-to-many relationship: (e.g.: products (many) and orders (many))
    - To detect many-to-many relationships between your tables, it is important to consider both sides of the relationship.
    - Create a third table, often called a ***junction table***, that breaks down the many-to-many relationship into two one-to-many relationships.
    - Insert the primary key from each of the two tables into the third table. As a result, the third table records each occurrence or instance of the relationship.
  + One-to-one relationship: (e.g.: products table and products supplement table with some empty fields)
  + When a one-to-one or one-to-many relationship exists, the tables involved need to share a common column or columns. When a many-to-many relationship exists, a third table is needed to represent the relationship.
* Refine the design.
* Apply the normalization rules.
  + Use these rules to see if the tables are structured correctly.
  + The process of applying the rules to the database design is called normalizing the database, or just normalization.
  + First normal form:
    - First normal form states that at every row and column intersection in the table there, exists a single value, and never a list of values.
  + Second normal form:
    - Second normal form requires that each non-key column be fully dependent on the entire primary key, not on just part of the key.
    - This rule applies when you have a primary key that consists of more than one column.
    - E.g. table having two primary keys as product id and order id. This table have a column with product name which is dependent only on product id and not on order id. The product name is not dependent on the entire primary key. So, the product name column has to be placed in products table.
  + Third normal form:
    - Third normal form requires that not only every non-key column be dependent on the entire primary key, but that non-key columns be independent of each other.

**Joins**

* JOIN is an SQL clause used to query and access data from multiple tables, based on logical relationships between those tables.
* Types of joins:
  + Inner join
  + Self join
  + Cross join
  + Outer join
    - Left outer join
    - Right outer join
    - Full outer join
* INNER JOIN:
  + Creates a result table by combining rows that have matching values in two or more tables.
  + Returns only those records or rows that have matching values and is used to retrieve data that appears in both tables.
* LEFT OUTER JOIN:
  + Includes in a result table unmatched rows from the table that is specified before the LEFT OUTER JOIN clause.
  + Gives the output of the matching rows between both tables. In case, no records match from the left table, it shows those records with null values.
* RIGHT OUTER JOIN:
  + Creates a result table and includes into it all the records from the right table and only matching rows from the left table.
  + Returns a result set that includes all rows in the right table, whether they have matching rows from the left table.
  + In case, a row in the right table does not have any matching rows in the left table, the column of the left table in the result set will have nulls.
* FULL OUTER JOIN:
  + Returns a result that includes rows from both left and right tables.
  + In case, no matching rows exist for the row in the left table, the columns of the right table will have nulls. Correspondingly, the column of the left table will have nulls if there are no matching rows for the row in the right table.
* SELF JOIN:
  + Joins the table to itself and allows comparing rows within the same table.
* CROSS JOIN:
  + Creates a result table containing paired combination of each row of the first table with each row of the second table.
  + CROSS JOIN joins every row from the first table with every row from the second table and its result comprises all combinations of records in two tables.

**Stored Procedures:**

* A stored procedure is a set of pre-compiled Structured Query Languages (SQL), so it can be reused and shared by multiple programs. It can access or modify data in a database.
* We can execute the stored procedures when required.
* Stored Procedures can't be called from a function because functions can be called from a select statement and Stored Procedures can't be called from. But you can call Store Procedure from Trigger.
* Stored Procedures can accept any type of parameter. Stored Procedures also accept out parameter.
* Stored Procedures may or may not return any values (Single or table) on execution.

***Syntax***:

Create proc Proc\_name

@permater

as begin

----Query here

end

(OR)

Create proc Proc\_name

@permeter datatype

@rslt output datatype

as begin

----Query here

select @rslt

end

**SQL Function:**

* A function is a database object in SQL Server. Basically, it is also a set of SQL statements that accept only input parameters and produce output in a single value form or tabular form.
* We can call a function whenever required. Function can't be executed because a function is not in pre-compiled form.
* Function can be called from Store Procedure or Trigger.
* Function can accept any type of parameter. But function can’t accept out parameter.
* Function must return any value.

***Syntax***:

create function funname(@parmeter datatype)

returns Returntype

as

begin Returntype

end

**Trigger:**

* A trigger is also a set of SQL statements in the database which automatically execute whenever any special event occurs in the database, like insert, delete, update, etc.
* Trigger can be executed automatically on specified action on a table like, update, delete, or update.
* Trigger can’t be called from Store Procedure or Function.
* We can’t pass a parameter to trigger.
* Trigger never return value on execution.

***Syntax***:

create trigger trigger\_name

before | after

{insert | update | delete}

on table\_name

for each row

---Query here

**Views:**

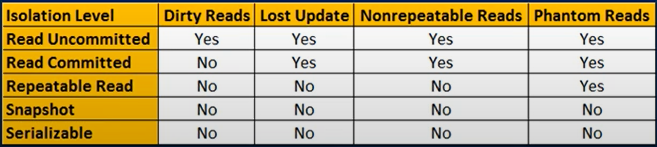
* A **view** is a virtual table based on the result of a SQL query.
* It does not store data itself but provides a simplified way to interact with complex queries.

**Indexes:**

* An **index** improves the speed of data retrieval by creating a structured lookup for data in a table or view.
* **Creating indexes on table:**
  + Clustered Index: A table can have only one clustered index. It determines the physical order of data in the table.
  + Non-Clustered Index: Can create multiple non-clustered indexes on a table.
* **Indexed views:**
  + An **indexed view** (materialized view) physically stores data and is optimized for frequent querying.
  + Requirements for indexed views:
    - All columns must have deterministic expressions.
    - The *SCHEMABINDING* option must be used in the view definition.

**Transaction:**

* A transaction is a group of commands that change the data stores in a database. A transaction is treated as a single unit of work.
* A transaction ensures that either all the commands succeed or none of them.
* **Concurrency problems:**
  + Dirty reads
  + Lost updates
  + Nonrepeatable reads
  + Phantom reads
* **SQL server transaction isolation levels:**
  + Read uncommitted
  + Read committed
  + Repeatable read
  + Snapshot
  + Serializable



* + Based on the isolation level, the performance and concurrency problems differ.
* **Dirty Read Concurrency Problem**:
  + A dirty read happens when one transaction is permitted to read data that has been modified by another transaction that has not yet been committed.

In most cases this wouldn’t cause a problem. However, if the first transaction is rolled back after the second reads the data, the second transaction has dirty data that doesn’t exist anymore.

* + ***Read uncommitted transactions isolation level is the only isolation level that has dirty read side effect.***

To read dirty data, isolation level can be set as read uncommitted or by using ***NOLOCK*** table hint***.***

* **Lost Update Problem:**
  + Lost update problem happens when 2 transactions read and update the same data.
  + Both read uncommitted and read committed transaction isolation levels have the lost update effect.
  + Repeated Read, Snapshot, and Serializable isolation levels don’t have this side effect.
  + The repeated read isolation level uses additional locking on rows that are read by the current transaction and prevents them from being updated or deleted elsewhere. This solves the lost update problem.
* **Non repeatable Read Problem:**
  + Non repeatable read happens when one transaction reads the same data twice and another transaction updates the data in between the first and second read of transaction one.
  + Repeatable read or any other higher isolation level solves the non-repeatable read problem.
* **Phantom Read Problem:**
  + Phantom read happens when one transaction executes a query, and it gets a different number of rows in the result set each time.

This happens when a second transaction inserts a new row that matches the WHERE clause of the query executed by the first transaction.

* + To fix the phantom read problem, set transaction isolation level of transaction 1 to serializable. This will place a range lock on the rows between the rows involved in transaction, which prevents any other transaction from inserting new rows within that range.
  + We don’t have phantom read problem in snapshot and serializable isolation levels.