System Development Life cycle

1. Problem identification
2. Analysis
3. Logical Design
4. Physical Design Mapping
5. Implementation
6. Testing & maintainance

Database System = Database(collection of related data) + DBMS(software)

Three Level /Schema Architecture

External Schema : what the user sees , They are concerned with what data the user will see and how the data will be presented to the user.(Different Views).

Conceptual Schema (The logical Model) : define database structures such as tables and constraints.

Internal Schema (The physical Model): *how* the data structures are implemented.

ERD

Basic constructs of the E-R model:

Entity : object. (Rectangle)

Attribute: property or characteristic. (Diamond)

Relationship: link between entities. (Eclipse)

Entities

Strong Entity: has primary key (solid underline)

Weak Entity (Dependant Entity): has composite primary key [partial key of weak entity (dashed underline) + primary key of its string entity].

Attributes

Simple Attribute : (solid eclipse)

Composite Attribute : ex. Address consist of steet and city

Derived Attribute: (dashed eclipse)

Multivalued Attribute (double eclipse): ex. You have multiple phone numbers.

Complex Attribute(multivalued + complex): ex. A company have mutiple branches in different addresses and each address is a composite attribute that consists of street and city.

Relationship

It's association between two ro more entities and may also has attributes.

Relationship properties

Degree of Realtionship : number of entity types that participate in a relationship

- **Unary(recursive):** between two instances of one entity type.
- **Binary :** between the instances of two entity types.
- **Ternary:** among the instances of three entity types.

Cardinality Constraint: How many instances of one entity will or must be connected to a single instance from the other entities.

- **One-One Relationship**
- **One-Many Relationship**
- **Many- Many Relationship**

Participation Constraint:

Total participation(0,N): ex. An employee must be working in a department.

Partial Perticipation(1, Max): ex. Some employees are managers.

Keys (chatgpt)

In a database management system, there are several types of keys that are used to identify and organize data. Some of the commonly used keys are:

Primary key: A primary key is a unique identifier for each record in a table. It cannot contain null values, and it must be unique across all records in the table. The primary key is used to enforce data integrity and to ensure that there are no duplicates in the table.

Foreign key: A foreign key is a field in a table that refers to the primary key of another table. It is used to establish relationships between tables, and to ensure that the data in the table is consistent with the data in the related table.

Candidate key: A candidate key is a set of one or more fields that can be used as a primary key. It must be unique across all records in the table, and it cannot contain null values.

Superkey: A superkey is a set of one or more fields that uniquely identify each record in a table. It can include the primary key as well as other fields.

Alternate key: An alternate key is a candidate key that is not selected as the primary key. It can also be used to identify records in the table.

Composite key: A composite key is a primary key that consists of two or more fields. It is used when a single field cannot uniquely identify a record in the table.

Partial key: A partial key, also known as a "partial candidate key," is a key that does not uniquely identify each record in a table, but when combined with one or more other fields, can be used to create a unique identifier (used with weak enetities).

These keys play a crucial role in the structure and organization of a database, ensuring data integrity, enforcing relationships between tables, and facilitating efficient data retrieval.

Relational Database Definitions

- **Table or entity:** a collection of records
- **Attribute or Column or field:** a Characteristic of an entity
- **Row or Record or tuble:** the specific characteristics of one entity
- **Database:** a collection of tables

Categories of T-SQL Statements

Data Definition Language(DDL): create, drop, alter.

Data Manipulation Language(DML): Insert, Update, Delete,Merge.

Data Control Language(DCL): Grant, Deny, Revoke.

Data Query Language(DQL): Select, orderby, groupby, aggregation functions, union.

Transactional Control Language(TCL): begin Transact, Commit, Rollback.

Batch: is a series of one or more statements submitted and executed at the same time.

Trucate VS Delete

The DELETE command is transactional, meaning that it can be rolled back if needed.

TRUNCATE TABLE is similar to the DELETE statement with no WHERE clause; however, TRUNCATE TABLE is faster and uses fewer system and transaction log resources.

Compared to the DELETE statement, TRUNCATE TABLE has the following advantages:

Less transaction log space is used
The DELETE statement removes rows one at a time and records an entry in the transaction log for each deleted row. TRUNCATE TABLE removes the data by deallocating the data pages used to store the table data and records only the page deallocations in the transaction log.

Fewer locks are typically used
When the DELETE statement is executed using a row lock, each row in the table is locked for deletion. TRUNCATE TABLE always locks the table and page but not each row.

Without exception, zero pages are left in the table
After a DELETE statement is executed, the table can still contain empty pages. For indexes, the delete operation can leave empty pages behind.

TRUNCATE TABLE removes all rows from a table, but the table structure and its columns, constraints, indexes, and so on remain. To remove the table definition in addition to its data, use the DROP TABLE statement.

Note to students that if the table to be truncated contains an identity column, the counter for that column is reset to the seed value defined for the column. If no seed was defined, the default value 1 is used. If they want to retain the identity counter for any reason, use DELETE instead.

Trucate VS Delete (chatgpt)

In SQL, "DELETE" and "TRUNCATE" are two different commands that can be used to remove data from a table, but they function differently.

"DELETE" is a DML (Data Manipulation Language) command that is used to remove specific rows from a table based on a condition. The syntax for DELETE is as follows:

```
DELETE FROM table_name
WHERE condition;
```

This command will remove all the rows from the table that satisfy the condition. If no condition is specified, all rows in the table will be deleted. The DELETE command is transactional, meaning that it can be rolled back if needed.

On the other hand, "TRUNCATE" is a DDL (Data Definition Language) command that is used to remove all the rows from a table. The syntax for TRUNCATE is as follows:

```
TRUNCATE TABLE table_name;
```

TRUNCATE removes all the rows from the table, but it does not record individual row deletions and is non-transactional. It also resets the identity seed of the table and deallocates all the data pages associated with the table, which makes it faster than DELETE for large tables.

In summary, DELETE is used to remove specific rows from a table based on a condition, whereas TRUNCATE is used to remove all rows from a table without any conditions. Use DELETE when you need to selectively remove data from a table, and use TRUNCATE when you need to remove all data from a table quickly and efficiently.

SQL statement processing order

1. From
2. Join
3. on
4. Where
5. Groupby
6. Having
7. Select
8. Orderby
9. Top
10. union
11. Result

Where Clause

The WHERE clause uses search arguments to filter the data requested in the SELECT statement. Search arguments contain the comparisons and criteria for selecting data. Search arguments are expressed using conditional statements and predicates.

- Conditional statements include operators such as: **=, <, >, <>, <=, >=.**
- Predicates are statements that return a TRUE, FALSE, or UNKNOWN result. These statements include: **BETWEEN, CONTAINS, EXISTS, IN, IS [NOT] NULL,** and **LIKE**.
- Criteria in the WHERE clause can be combined using the **NOT, AND,** and **OR** operators.

Explain what a search argument is. Make sure to introduce the word predicate.

Mention that by using search conditions, performance can be improved by limiting the numbers of rows to be returned.

String Comparison

Explain the LIKE clause. Explain different types of wildcard characters in a table. Describe different comparison operators in a table using the SELECT, the AND and LIKE operators

String comparison operators search for strings and substrings within a text, ntext, char, nchar, varchar, or nvarchar data type.

The **=** operator checks for an exact match between two strings.

LIKE combined with a wildcard searches for the search condition argument in the left value string. Note that placing the **%** sign at the beginning of the criteria may adversely affect query performance by eliminating the possible use of indexes to help with the search.

FREETEXT searches for the meaning rather than exact words.

If a phrase is passed into FREETEXT, it will break the phrase into the component words to perform the search.

Uses "inflectional" forms of the words to search. For instance "drive" is the "inflectional stem" of drives, drove, driving and driven.

Uses the thesaurus to search for additional forms and "replacement" words.

CONTAINS performs a "fuzzy" search. This is a special predicate specific to full text searches.

Looks for a word or phrase.

Also uses inflectional forms

Also looks for synonyms by using a thesaurus.

Can look for a word or phrase within a certain proximity of another word or phrase.

When multiple words are used for the search, you can specify a value between 0.0 and 1.0 to give specific words more weight in the search criteria.

Emphasize the **FREETEXT** and **CONTAINS** both operate only on full-text indexed columns.

Mention that there are four different wildcard symbols that can be used:

% replaces any string of zero or more characters

_ (underscore) replaces any single character

[] allows you to specify a list of characters, any of which can be matched

- **[a-f]** looks for any of the letters in the sequence a – f
- **[abcdef]** also looks for any of the letters in the sequence a – f

[^] looks for all characters except those given in the range or set

Working with NULL Values

- **ISNULL() –** queries with data that you want to check for NULL values.

- **NULLIF() –** queries that you want to offer a more meaningful value in place of the NULL keyword being displayed in the result.
- **COALESCE() –** Queries where NULL values may exist and you wish to substitute one of several possibilities into a column of the result set.

Working with NULL Values (chatgpt)

"ISNULL": The "ISNULL" function is used to check if a value is null and replace it with a specified value if it is. The syntax for ISNULL is as follows:

ISNULL(expression, replacement_value)

"NULLIF": The "NULLIF" function is used to compare two expressions and return null if they are equal. The syntax for NULLIF is as follows:

NULLIF(expression1, expression2)

"COALESCE": The "COALESCE" function is used to return the first non-null value in a list of expressions. The syntax for COALESCE is as follows:

COALESCE(expression1, expression2, ..., expressionn)

Examples

```
SELECT ISNULL(quantity, 0) FROM orders;
```

This query will return the quantity of each order, but if the quantity is null, it will be replaced with the value 0.

```
SELECT NULLIF(price, 0) FROM products;
```

This query will return the price of each product, but if the price is 0, it will be replaced with null.

```
SELECT COALESCE(name, email, phone) FROM customers;
```

This query will return the first non-null value among the name, email, and phone fields for each customer. If all three fields are null, the function will return null.

Sorting Data

Sort order can be specified as ASC (ascending) or DESC (descending) for each column.

Mention that ASC (ascending) order is the assumed order if no order is specified.

Mention that sorting data can slow down performance as SQL server needs to perform a distinct step. However, indexes will improve performance.

Labeling Columns in Result Sets

- Aliases are used to create custom column headers in the result set display.
- The optional **AS** clause can be added to make the statement more readable.

```
SELECT salary*12 as [annual salary]

From instructor
```

Substring Method (chatgpt)

In SQL, the "SUBSTRING" function is used to extract a substring from a string. The syntax for the SUBSTRING function is as follows:

```
SUBSTRING(string, start_position, length)
```

The first argument is the string that you want to extract the substring from. The second argument is the position in the string where the substring should begin, and the third argument is the length of the substring.

Here is an example of using the SUBSTRING function in SQL:

```
SELECT SUBSTRING('Hello, world!', 1, 5);
```

This query will return the substring "Hello" from the string "Hello, world!" because we specified a start position of 1 and a length of 5.

You can also use the SUBSTRING function to extract a substring from a column in a table. Here is an example:

```
SELECT SUBSTRING(name, 1, 3) FROM employees;
```

This query will return the first three characters of the "name" column for each row in the "employees" table.

Note that the position argument starts from 1, not 0. If the start position is negative, the SUBSTRING function will count backward from the end of the string. If the length argument is not specified, the SUBSTRING function will return all the characters from the start position to the end of the string.

Querying Multiple Tables by Using Joins

- joins is faster than subqueries if the number of tables is smaller and if there is indexes

  In SQL, an "outer join" is used to combine rows from two or more tables based on a related column, and it includes all the rows from one table and matching rows from another table, or null values where there is no match.

  There are two types of outer joins:

  1. "Left outer join" (or "Left join"): This type of join returns all the rows from the left table and matching rows from the right table. If there are no matching rows in the right table, the result will contain null values for those columns.

  2. "Right outer join" (or "Right join"): This type of join returns all the rows from the right table and matching rows from the left table. If there are no matching rows in the left table, the result will contain null values for those columns.

  Here is an example of using a left outer join in SQL:

```
SELECT customers.name, orders.order_date
FROM customers
LEFT JOIN orders ON customers.id = orders.customer_id;
```

  In this query, we are selecting the name of each customer and the date of their orders. We are using a left join to include all the customers, even if they have no orders yet. If a customer has no orders, the order_date column will contain a null value.

  Here is an example of using a right outer join in SQL:

```
SELECT orders.order_date, customers.name
FROM orders
RIGHT JOIN customers ON orders.customer_id = customers.id;
```

  In this query, we are selecting the date of each order and the name of the customer who placed the order. We are using a right join to include all the orders, even if there is no matching customer yet. If an order has no matching customer, the name column will contain a null value.

  Note that some databases, such as MySQL, support a "full outer join" that includes all the rows from both tables. However, other databases, such as Oracle and SQL Server, do not support full outer joins directly, but they can be emulated using a combination of left and right outer joins.

```
select st_fname,dept_name
from student s full outer join department d
on s.dept_id=d.dept_id
```

Categorizing Statements by Types of Joins

- **Inner Join**

- Includes equi-joins and natural joins
- Use comparison operators to match rows

- **Outer Join**

- Includes left, right, or full outer joins

- **Cross Join**

- Also called Cartesian products

```
1.select st_fname,dept_name
from student s cross join department d
2.select st_fname,dept_name
from student s , department d
```

- Is defined as all possible combinations of rows in all tables
- Results in a rowset containing the number of rows in the first table times the number of rows in the second
- Can result in huge result sets that take several hours to complete!

- **Self Join**

- Refers to any join used to join a table to itself

```
select stud.st_fname,super.st_fname as "supervisor Name"
from student super,student stud
where super.st_id=stud.st_super
```

Not equal join <> (chatgpt)

In SQL, you can use a "not equal join" to combine rows from two or more tables based on a related column, but only include rows where the values in that column do not match. This type of join is achieved by using the "<>" (not equal) operator in the join condition.

Here is an example of using a not equal join in SQL:

```
SELECT customers.name, orders.order_date

FROM customers

INNER JOIN orders ON customers.id <> orders.customer_id;
```

In this query, we are selecting the name of each customer and the date of their orders. We are using a not equal join to include only the customers who have no orders yet or the orders that have no matching customer. If a customer has an order, but the order is placed by a different customer, it will be included in the result.

Note that not all databases support the "<>" operator in the join condition. In some databases, you can use the "!=" operator instead. Also, be careful when using not equal joins, as they can result in unexpected or misleading results if the data is not properly filtered or if the join condition is not well defined.

Union & Union all (chatgpt)

In SQL, you can use the "UNION" and "UNION ALL" operators to combine the results of two or more SELECT statements into a single result set.

The "UNION" operator is used to combine the results of two SELECT statements and remove any duplicate rows. Here is the syntax for using the UNION operator:

```
SELECT column1, column2, ...

FROM table1

UNION

SELECT column1, column2, ...

FROM table2;
```

In this query, we are selecting the same columns from two tables and combining the results using the UNION operator. The UNION operator will remove any duplicate rows from the result set.

The "UNION ALL" operator is used to combine the results of two SELECT statements and include all the rows, even if they are duplicates. Here is the syntax for using the UNION ALL operator:

```
SELECT column1, column2, ...

FROM table1

UNION ALL

SELECT column1, column2, ...

FROM table2;
```

In this query, we are selecting the same columns from two tables and combining the results using the UNION ALL operator. The UNION ALL operator will include all the rows from both tables, even if they are duplicates.

Note that when using the UNION and UNION ALL operators, the number and data types of the columns in the SELECT statements must match, or the query will fail. Also, the columns are ordered based on the order they appear in the first SELECT statement. If the order of the columns in the other SELECT statements is different, you can use aliases to ensure that they are ordered correctly.

Limiting Result Sets by Using the EXCEPT and INTERSECT Operators

EXCEPT returns any distinct values from the left query that are not also found on the right query, and is used to except members from a result set.

INTERSECT returns any distinct values that are returned by both the query on the left and right sides of the INTERSECT operand, and is used to determine which tables share similar data.

The basic rules for combining the result sets of two queries that use EXCEPT or INTERSECT are the following:

- The number and the order of the columns must be the same in all queries
- The data types must be compatible

Except Example

```
SELECT ProductID
FROM Production.Product
EXCEPT
SELECT ProductID
FROM Production.WorkOrder
```

Intersect Example

```
SELECT ProductID
FROM Production.Product
INTERSECT
SELECT ProductID
FROM Production.WorkOrder
```

All Operator

All operator Example :

We don’t know the values in list

```
Select Lname , Fname

From employee

Where salary > All ( select salary

from employee where Dno=5)
```

We can replace ALL by max aggregate function:

```
Select last_name, salary

from employees

where salary > (select max ( salary ) from employee where dno=5);
```

Exists Condition

The EXISTS condition is considered "to be met" if the subquery returns at least one row.

The syntax for the EXISTS condition is:

```
SELECT columns
FROM tables
WHERE EXISTS ( subquery );
```

Exists Example

```
SELECT *
FROM suppliers
WHERE EXISTS
  (select *
    from orders
    where suppliers.supplier_id = orders.supplier_id);
```

Not Exists Example

Retrieve the name of employees who have no dependents

```
Select  name

From employee

Where Not Exists ( select * from dependent where ssn=Essn)
```

Aggregate Functions (COUNT , SUM , MAX, MIN and AVG)

Find the total number of employees in the company?

Select count(*) from employees

Find the number of employees in the research department ?

Select count(*) from employee , department

Where dno=dnumber and dname ='Research'

Note : Group Functions ignore Null values in the columns

Grouping

Apply aggregate functions to a subgroups of tuples

For each department retrieve the department number , the number of employees in the department, and their average salary

```
Select dno , count(*) , avg(salary)
From employee Group by dno
```

Having Example

For each project on which more than two employees work, retrieve the project number, the project name , and the number of employees who work on the project

Select pnumber, pname ,count(works_on.pno)

From project , Works_on

Where Pnumber = Pno

Group by pnumber, pname

Having count(*) > 2

Data Control Language (DCL)

Grant Examples

- Grant Select on table employees to Ahmed;

- Grant All on Table department to Mary, Ahmed;

- Grant Select on table employees to Ahmed with grant Option;

Revoke Examples

-Revoke update on Table department From Mary;

-Revoke All on Table department From Mary, Ahmed;

Normalization

Normalization: The process of structuring data to minimize duplication and inconsistencies.

- Normalization is a bottom-up Analysis
- Normalization is used to reduce Null Values
- Normalization is used to improve performance

Functional Dependency

In SQL, a "functional dependency" is a relationship between two or more columns in a table, where the value of one column (the "dependent" column) is determined by the value of another column (the "determinant" column). This relationship is often expressed using the notation:

determinant -> dependent

For example, in a table of customer orders, the order total might be dependent on the quantity and price of each item, which are the determinants. We can express this as:

(quantity, price) -> order_total

Functional dependencies are important in database design because they help ensure data integrity and minimize redundancy. By identifying the dependencies between columns, we can normalize the table to eliminate redundant data and reduce the risk of inconsistencies or errors.

some examples

- social security number determines employee name

  SSN -> ENAME

- project number determines project name and location

  PNUMBER -> {PNAME, PLOCATION}

Types of functional dependency

- **Full Functional Dependency**

  Attribute is fully Functional Dependency on a PK if its value is determined by the whole PK

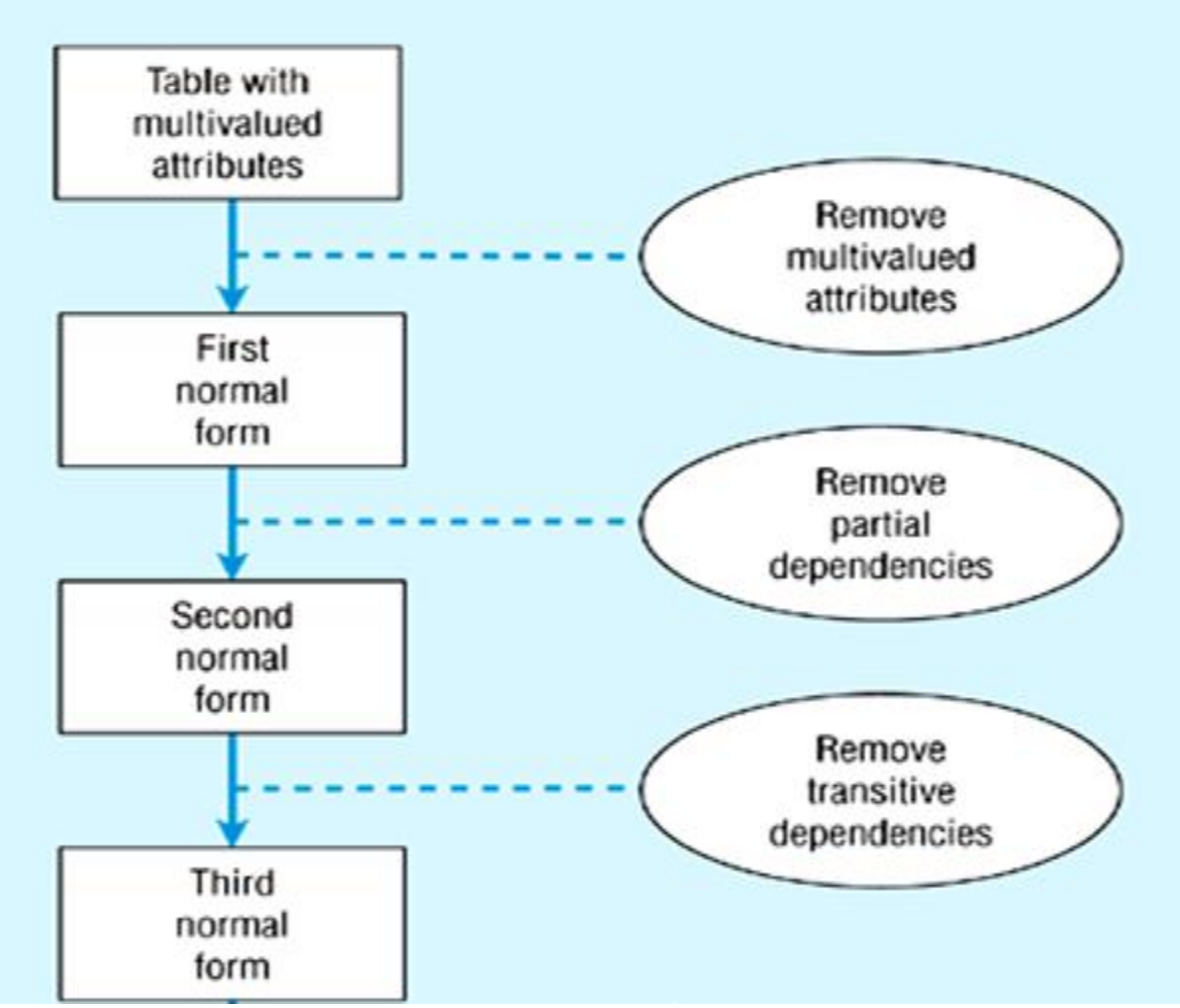
- **Partial Functional Dependency**

  Attribute if has a Partially Functional Dependency on a PK if its value is determined by part of the PK**(Composite Key)**

- **Transitive Functional Dependency**

  Attribute is Transitively Functional Dependency on a table if its value is determined by anther non-key attribute which it self determined by PK

Steps of Normalization



Normalization (chatgpt)

Normalization is the process of organizing data in a database to minimize redundancy and dependency. The goal of normalization is to design a database schema that is flexible, efficient, and easy to maintain.

In SQL, normalization is typically achieved by decomposing a table into two or more tables based on functional dependencies. There are several levels of normalization, each with its own set of rules for decomposing a table:

1. First Normal Form (1NF): A table is in 1NF if all the values in each column are atomic (indivisible). This means that each column should contain only one piece of information, and there should be no repeating groups or arrays. To convert a table to 1NF, we typically create a new table for each repeating group(several multivalued columns), with a foreign key linking it to the original table.

2. Second Normal Form (2NF): A table is in 2NF if it is in 1NF and every non-key column is fully dependent on the primary key. This means that each column should be dependent on the entire primary key, not just part of it. To convert a table to 2NF, we typically create a new table for each partial dependency, with a foreign key linking it to the original table.

3. Third Normal Form (3NF): A table is in 3NF if it is in 2NF and every non-key column is dependent only on the primary key, and not on any other non-key column. This means that each column should be independent of other columns in the same table. To convert a table to 3NF, we typically create a new table for each transitive dependency, with a foreign key linking it to the original table.

There are also higher levels of normalization, such as Boyce-Codd Normal Form (BCNF) and Fourth Normal Form (4NF), which further refine the rules for decomposing a table.

The process of normalization can be complex and time-consuming, but it is important for creating a well-designed database schema that is efficient, flexible, and easy to maintain. It can also help ensure data integrity and prevent errors or inconsistencies.

Top & Top with Ties (chatgpt)

In SQL, the `TOP` and `TOP WITH TIES` clauses are used to limit the number of rows returned in a query result. However, they differ in how they handle ties, which occur when multiple rows have the same value for the column being sorted.

Whats is Ties: In the context of SQL and query results, "ties" refer to rows that have the same value for the column(s) being sorted or ordered. Ties occur when multiple rows have identical values for the sorting criteria.

For example, consider a table of employees with columns "EmployeeID", "FirstName", and "Salary". If you are sorting the employees based on their salary in descending order, ties would occur when multiple employees have the same salary.

1. `TOP`: The `TOP` clause is used to specify the number of rows to be returned from the query result, regardless of ties. For example, if you use `TOP 5` in your query, it will return the top 5 rows based on the specified sorting criteria, even if there are ties. If there are ties, some rows with the same value may be excluded from the result.

2. `TOP WITH TIES`: The `TOP WITH TIES` clause is used to include additional rows with the same value as the last row in the specified number of rows. In other words, it includes tied rows in the result. For example, if you use `TOP 5 WITH TIES`, it will return the top 5 rows based on the specified sorting criteria, and if there are ties for the last row, it will include those tied rows as well.

Here's an example to illustrate the difference:

Assume you have a table named "Employees" with columns "EmployeeID" and "Salary". You want to retrieve the top 3 employees based on their salary.

Using `TOP`:

```
SELECT TOP 3 EmployeeID, Salary
FROM Employees
ORDER BY Salary DESC;
```

This query will return the top 3 employees with the highest salaries, regardless of ties.

Using `TOP WITH TIES`:

```
SELECT TOP 3 WITH TIES EmployeeID, Salary
FROM Employees
ORDER BY Salary DESC;
```

This query will return the top 3 employees with the highest salaries, and if there are ties for the last salary, it will include those tied rows in the result.

By using `TOP WITH TIES`, you ensure that tied rows are included in the result, providing a more comprehensive view of the data when ties exist.

Remember to adjust the column names and table name according to your specific scenario.

Database Object Fullpath

```
--fullpath object
-----> Servername.DBName.SchemaName.ObjectName
```

Notice we can use this path to access objects from different databases or even different SQL servers

Example:

```
select dept_name from ITI.dbo.Department
union all
select dname from Company_SD.dbo.Departments
```

Select Into

Is a DDL query which creates a new table and copy the data of the first table into the new table.

```
select * into tab2
from instructor
```

```
//this query creates a new table calles tab2 and copy all the data of table instructor into the new created table. Hence we can't run this query twice because we can't have two table with the same name.
```

```
select * into tab7
from student
where 1=2  //false condition
```

```
//this query will create a new empty table (tab7) which have the same structure as Student table
```

Because this condition won't ever happen.

Insert based on select

It's a DML query used to copy data of a table to another table but both tables should have the same structure

```
insert into tab5
select st_id,st_fname from student where st_address='alex'
```

```
//in this example table tab5 have only two columns so we can't copy all the data of table Student and we can only copy data of two columns which has the same datatype.
```

NewID() (chatgpt)

`NEWID()` is a built-in function in SQL Server that is used to generate a new uniqueidentifier (GUID) value. A GUID (Globally Unique Identifier) is a 128-bit identifier that is globally unique across all computers and networks.

The `NEWID()` function returns a new GUID value each time it is called. It can be used in various scenarios where a unique identifier is required, such as generating primary keys for tables, creating random identifiers, or generating unique values for temporary tables or session-specific data.

Here's an example of how to use the `NEWID()` function to generate a new GUID value:

```
SELECT NEWID() AS NewGuid;
```

This query will return a new GUID value as the result, represented as a string in the format "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx", where each "x" represents a hexadecimal digit.

The `NEWID()` function is commonly used in conjunction with the `INSERT` statement to generate unique identifiers for new records in a table. For example:

```
INSERT INTO TableName (Id, Column1, Column2)

VALUES (NEWID(), 'Value1', 'Value2');
```

In this example, a new row is inserted into the "TableName" table with a new GUID value generated by `NEWID()` for the "Id" column.

Please note that the `NEWID()` function is specific to SQL Server and may not be available in other database systems.

//Notice: NewID() can be used to display data randomly

```
select * ,newid() as X
from student
order by X

//this query is used to display student table data randomly each time you run it


select top(1)*
from student
order by newid()

//this query is used to get a random student each time you run it
```

Notes :

```
//this query will run properly because order by clause is executed after select clause

select st_fname+' '+st_lname as fullname
from student
order by fullname


//but this query will generate an error because select clause is executed after where clause

So at where clause the compiler doesn’y know which column is fullname as it’s defined later in the
select clause

select st_fname+' '+st_lname as fullname
from student
where fullname='ahmed hassan'

//To solve this problem we have two options

select st_fname+' '+st_lname as fullname
from student
where st_fname+' '+st_lname='ahmed hassan'

//anything after from clause must be a table

select *
from (select st_fname+' '+st_lname as fullname
         from student) as newtable
where fullname='ahmed hassan'
```

Having Special case

By default Having clause must come after group by

But there is a specual case where having can be used without group by

```
Select Sum(Salary)

From Instructor

Having count(Ins_Id)<100

//this query calculates the sum of salaries of instructors if the number of the instructors is less
than 100

//in this query we were able to use having without group by because there was no other columns in the
select clause that require using group by

Select Sum(Salary), Dept_Id

From Instructor

Group by Dept_ID

Having count(Ins_Id)<100

//in this query we must use group by because there is another column mentioned in the select clause
```

Ranking Functions(chatgpt)

Ranking functions in SQL are used to assign a rank or position to each row in the result set based on a specific criteria. There are several ranking functions available in SQL, including:

1. ROW_NUMBER(): Assigns a unique number to each row in the result set, without any gaps. The numbering starts from 1 and increments by 1 for each subsequent row.

2. RANK(): Assigns a unique rank to each row, with equal values receiving the same rank. The next rank is skipped if there are ties.

3. DENSE_RANK(): Similar to the RANK() function, but there are no gaps in the rank values, even if there are ties. The next rank is not skipped.

4. NTILE(): Divides the result set into a specified number of equal-sized groups or buckets and assigns a group number to each row.

Notice: if the table rows can't be divided into equal-sized groups then it will be grouped so that last groups are less by one (maximum)

Ex. if we have 14 rows ➔ 5,5,4 last group is 4

Ex. if we have 13 rows ➔ 5,4,4 and so on

These ranking functions are typically used in conjunction with the ORDER BY clause to determine the order or criteria for ranking.

Here's an example of how to use the ROW_NUMBER() function:

```
SELECT
  ROW_NUMBER() OVER (ORDER BY Salary DESC) AS RowNumber,
  EmployeeName,
  Salary
FROM
  Employees;
```

In this example, the ROW_NUMBER() function is used to assign a unique row number to each employee based on their salary in descending order. The result set will include the row number, employee name, and salary.

Ranking Functions

Row_Number()
Dense_rank()
NTiles(Group)
Rank()

```
Select *
From (
        Select *, Row_Number() over(order by esal desc) as RN
                , Dense_rank()   over(order by esal desc) as DR
                ,NTile(3)        over(order by esal desc) as G
        From employee ) as Newtable
where  RN=1
       RN=3
       RN<=2
       DR=1
       DR<=2
       G=1
```

eid	ename	esal	did	RN	DR	G
15	ahmed	10000	10	1	1	1
14	ali	10000	10	2	1	1
12	eman	9000	10	3	2	1
1	nada	9000	10	4	2	1
2	reem	9000	10	5	2	1
3	khalid	8000	10	6	3	2
7	mohamed	7000	20	7	4	2
8	sayed	7000	20	8	4	2
6	hassan	6000	20	9	5	2
5	omar	6000	20	10	5	2
9	sally	5000	30	11	6	3
10	shimaa	4000	30	12	7	3
11	hana	4000	30	13	7	3
12	lama	3000	30	14	8	3

```sql
select *,Row_number() over(order by st_Age desc) as RN
from Student

select *,Dense_rank() over(order by st_Age desc) as DR
from Student
```

200 %

Results | Messages

	St_Id	St_Fname	St_Lname	St_Address	St_Age	Dept_Id	St_super	RN
1	13	Said	NULL	NULL	30	30	12	1
2	14	NULL	Saleh	Tanta	30	30	NULL	2
3	8	Mohamed	Fars	Alex	28	20	6	3
4	6	Heba	Farouk	Mansoura	25	20	NULL	4
5	7	Ali	Hussien	Cairo	25	20	6	5
6	9	Saly	Ahmed	Mansoura	24	20	NULL	6
7	10	NULL	NULL	Alex	24	30	9	7
8	11	Marwa	Ahmed	Cairo	24	30	9	8
9	4	Ahmed	Mohamed	Alex	24	10	1	9
10	5	NULL	Mohamed	Alex	24	10	1	10
11	3	Mona	Saleh	Cairo	22	10	1	11
12	2	Amr	Magdy	Cairo	21	10	1	12
13	12	Noha	Omar	Cairo	21	30	NULL	13

DESKTOP-VF50P25.ITI - dbo.Student | Day5.sql - (local)....P-VF50P25\Rami (63))*

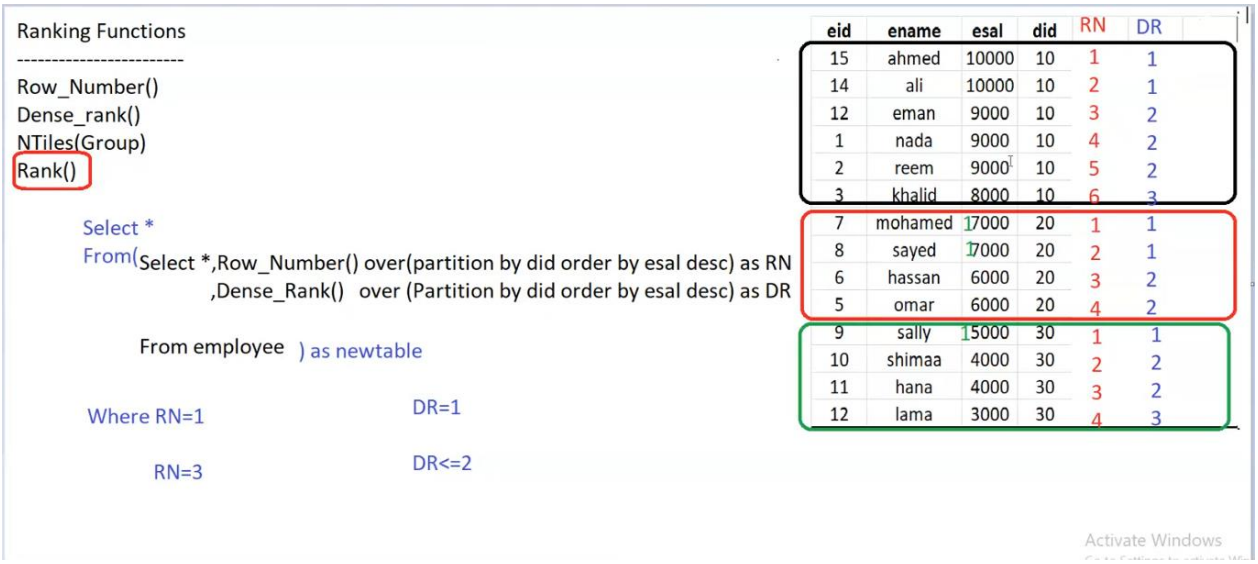

```sql
select *,Row_number() over(order by st_Age desc) as RN
from Student

select *,Dense_rank() over(order by st_Age desc) as DR
from Student
```

200 %

Results | Messages

	St_Id	St_Fname	St_Lname	St_Address	St_Age	Dept_Id	St_super	DR
1	13	Said	NULL	NULL	30	30	12	1
2	14	NULL	Saleh	Tanta	30	30	NULL	1
3	8	Mohamed	Fars	Alex	28	20	6	2
4	6	Heba	Farouk	Mansoura	25	20	NULL	3
5	7	Ali	Hussien	Cairo	25	20	6	3
6	9	Saly	Ahmed	Mansoura	24	20	NULL	4
7	10	NULL	NULL	Alex	24	30	9	4
8	11	Marwa	Ahmed	Cairo	24	30	9	4
9	4	Ahmed	Mohamed	Alex	24	10	1	4
10	5	NULL	Mohamed	Alex	24	10	1	4
11	3	Mona	Saleh	Cairo	22	10	1	5
12	2	Amr	Magdy	Cairo	21	10	1	6
13	12	Noha	Omar	Cairo	21	30	NULL	6

Ranking Functions with partition



eid	ename	esal	did	RN	DR
15	ahmed	10000	10	1	1
14	ali	10000	10	2	1
12	eman	9000	10	3	2
1	nada	9000	10	4	2
2	reem	9000	10	5	2
3	khalid	8000	10	6	3
7	mohamed	17000	20	1	1
8	sayed	17000	20	2	1
6	hassan	6000	20	3	2
5	omar	6000	20	4	2
9	sally	15000	30	1	1
10	shimaa	4000	30	2	2
11	hana	4000	30	3	2
12	lama	3000	30	4	3

Ranking functions Examples

```
Select *
From ( Select *, Row_number() over(order by st_age desc) as RN
       from Student) as newtable
where RN=1

//This query will get the first student who has the oldest age

select *
From (Select *, Dense_rank() over(order by st_age desc) as DR
      from Student)as newtable
Where DR=1

//This query will get all the students who has the oldest age

Select *
From ( Select *, Row_number() over(Partition by dept_id order by st_age desc) as RN
       from Student) as newtable
where RN=1

//This query will get the first student who has the oldest age in each department

select *
From (Select *, Dense_rank() over(Partition by dept_id order by st_age desc) as DR
      from Student)as newtable
where DR=1

//This query will get all the students who has the oldest age in each department

select *
From (Select *, Ntile(2) over(Partition by dept_id order by st_age desc) as G
      from Student)as newtable
where G=1

//this query will devide each departmet in student table into two equal-sized groups if possible and
get the students of the fisrt group
```

IIF(chatgpt)

The `IIF()` function is a logical function (like trenary operator) in SQL that returns one of two values based on a specified condition. It is available in some database systems, such as SQL Server, and provides a shorthand way to write conditional expressions.

The syntax of the `IIF()` function is as follows:

```
IIF(condition, value_if_true, value_if_false)
```

Let's break down the components of the `IIF()` function:

- **`condition`**: This is the condition that is evaluated. It can be any expression that returns a Boolean (TRUE or FALSE) value.

- **`value_if_true`**: This is the value that is returned if the condition evaluates to TRUE.

- **`value_if_false`**: This is the value that is returned if the condition evaluates to FALSE.

Here's an example to illustrate the usage of the `IIF()` function:

```
SELECT EmployeeName, Salary,
       IIF(Salary > 5000, 'High', 'Low') AS SalaryCategory
FROM Employees;
```

In this example, the `IIF()` function is used to categorize employees based on their salary. If the salary is greater than 5000, the value "High" is returned; otherwise, the value "Low" is returned. The result set includes the employee name, salary, and the salary category.

The `IIF()` function provides a concise way to write conditional expressions in SQL queries, especially for simple conditions with only two possible outcomes. However, it's important to note that the availability and syntax of the `IIF()` function may vary depending on the database system you are using.

Some built-in functions

```
select convert(varchar(20),getdate())

select cast(getdate() as varchar(20))

//The third parameter passed to convert() function is used to specify how this date will be displayed
select convert(varchar(20),getdate(),101)
select convert(varchar(20),getdate(),103) //  25/05/2023
select convert(varchar(20),getdate(),111)
select convert(varchar(20),getdate(),110)
select convert(varchar(20),getdate(),107) // May 25, 2023


select format(getdate(),'dd-MM-yyyy')
select format(getdate(),'dddd MMMM yyyy')
select format(getdate(),'ddd MMM yy')
select format(getdate(),'dddd')
select format(getdate(),'MMMM')
select format(getdate(),'hh:mm:ss')
select format(getdate(),'hh tt')
select format(getdate(),'HH')
select format(getdate(),'dd-MM-yyyy hh:mm:ss')
select format(getdate(),'dd-MM-yyyy hh:mm:ss tt')

select format(getdate(),'dd') // return nvarchar
select day(getdate()) // returns int

select eomonth(getdate()) //returns the last day of the month [ex. 2023-05-31]

select format(eomonth(getdate()),'dd') //returns the last day of the month [ex. 31]

select format(eomonth(getdate()),'dddd') //returns the last day of the month [ex. Wendesday]


select upper(st_fname),lower(st_lname)
from student

select len(st_fname),st_fname
from student

select substring(st_fname,1,3)
from student

select db_name()

select suser_name()
```

Advanced SQL

File Group

Microsoft save database on harddisk (HD) in two representations

Physical Representation (physical files) ➔ [mdf, ldf]

Logical Representation (logical file groups) ➔[primary File group]

By default there is a primary file group points to the MDF file, so when we create a new table, this table will be saved in the mdf file.

But we create secondary file groups that points to other physical files that has **.ndf** extension.

So It's better to keep the mdf file only for metadata, and add the newly created tables to other physical files (.ndf).

If we have to do many join quieries between two tables, it's better to save those tables in different file groups (that points at different ndf files which may be saved on different harddisks) so that when the query runs, data of these tables will be retrieved parallely not sequentialy and hence we can enhance the performance.

We can add new file groups and new physical files(.ndf) when creating the database.

Note: when you create a table by default it's added to the primary file group, but you can change this behavior from table properties and choose any other file group.

Column Properties:

Default Value or bining : allows you to add default value or method (ex. getdate()) to a column.

Computed Column Specification:

Formula : is used to provied equation to get drived attribute .

Is persisted: specify whether to save value of the drived attribute to the harddisk or not (by default only the equation is saved) but we can make the drived attribute persisted if it's calculation would be complex or if it's used to calculate the value of other attributes.

Is Sparse: is used to specify whether to keep Null values on the harddisk or not (by default it's value is NO which means Null values will be saved on the harddisk), but if you're pretty sure that a column will have many Null values, then it's better to set is sparse to YES.

Relationship Properties:

Insert And Upadate Specification : is used to specify what happens to the child when we delete or update the parent

Options : [no Action, set Null, set Default, Cascade]

On delete:

On Update:

Add users to sql server

Allow mixed mode :

On server name press right click and go to properties then choose secuirty and then choose SQL server and windows authentication mode.

Restart the server : On server name press right click and click restart.

Add new Login:

Security ➔ Logins ➔ new Login ➔ enter login name and choose SQL server authentication then enetr password and confirm it.

Add Login as a user to a specific database:

Click on the desired database ➔ security ➔ users ➔ right click and choose new user ➔ enetr the Login name in the username and loginname fields.

Till now the user won't be able to see any tables or schemas of this database.

give permissions to a user on a specific schema:

Click on the desired database ➔ security ➔ schemas ➔ Right Click on the desired database ➔ choose properties ➔ go to permissions ➔ on users or roles sections click on search and choose the user ➔ specifiy the desired permissions to this user.

Schema

a schema is a logical container or namespace that provides a way to organize and manage database objects, to define the structure of the database, and to control access to the database objects.

```
create schema HR
```

```
alter schema HR transfer Student //transfer student table from default schema dbo to HR schema
```

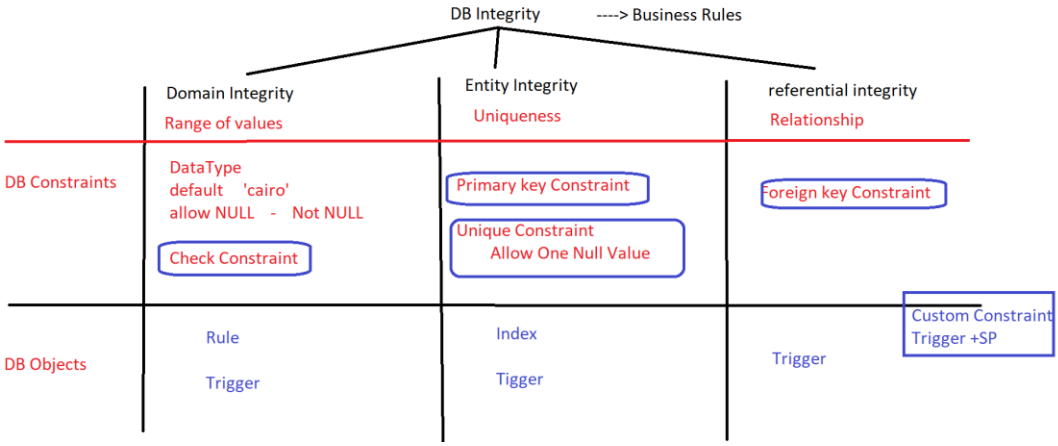
Synonym

a synonym is an alias or alternate name for a database object, such as a table, view, stored procedure, or function. Synonyms are used to simplify the naming and management of database objects, and to provide an additional layer of security and abstraction.

```
create synonym EmpDH
for  HumanResources.EmployeeDepartmentHistory //give shortcut to the table name to make it easier.
```

```
select * from EmpDH
```

DB Integrity



Main Constraints:

- Primary key Constraint
- Forieng key Constraint
- Check Constraint
- Unique Constraint
- Custome constraint (Trigger or SP)

Constraints Example :

```
create table emps
(
 eid int identity(1,1),
 ename varchar(20),
 eadd varchar(20) default 'cairo',
 hiredate date default getdate(),
 sal int,
 overtime int,
 netsal as(isnull(sal,0)+isnull(overtime,0)) persisted, //equation for drived attribute
 BD date,
 age as year(getdate())-year(BD), //can't be persisted because getdate() is changeable
 hour_rate int not null,
 gender varchar(1),
 dnum int,
 constraint c1 primary key(eid,ename),//composite primary key
 constraint c2 unique(sal),
 constraint c3 unique(overtime),
 constraint c4 check(sal>1000),
 constraint c5 check(overtime between 100 and 500),
 constraint c6 check(eadd in('alex','mansoura','cairo')), //default value must be included
 constraint c7 check(gender='f' or gender='M'),
 constraint c8 foreign key(dnum) references depts(did)
      on delete set NULL on update cascade
)
```

We can add consraints on table columns after creating the table

```
alter table emps add constraint c9 check(hour_rate>1000)
```

Notice:

Constraints are applied to all columns data, so when we add a new constraints we must make sure there's no conflict between column data and the constraint.

One column can have multiple constraints.

Rules

Rules are special constraints applied on the database level (programability ➔ Rules)

Rules are only applied to the new added data unlike constraints.

One column can only have one Rule

Rules can be shared between multiple columns on different tables

```
create rule r1 as @x>1000 //@x is a variable that will be replaced by column name

sp_bindrule r1,'instructor.salary' // 'instructor.salary' can only have one rule
sp_bindrule r1,'emps.sal' //rule r1 is shared between multiple columns
```

we can't drop a rule unless we unbind it from all columns

```
sp_unbindrule 'instructor.salary'
sp_unbindrule 'emps.sal'

drop rule r1
```

Default

Default constraints are used to provide a default value for a column in a table if no value is specified when a new row is inserted.

```
create default def1 as 5000

sp_bindefault def1,'instructor.salary'

sp_unbindefault 'instructor.salary'

drop default def1
```

Rules are usually used to create custom datatypes. //int value>1000 default 5000

```
sp_addtype newdt,'int'   --4 bytes

create rule r1 as @x>1000

create default def1 as 5000

sp_bindrule r1,newdt

sp_bindefault def1,newdt

create table mystaff
(
 sid int,
 sname varchar(20),
 salary newdt
)
```

Variables

Local variables (local through [batch, function, stored procedure]).

Global Variables (can't be declared or assigned).

A)Local Variables:

Declaring variable

Declare @x int //@x holds Null

Decalre @x int=100 //decleration and initialization

Initialzing variable

Set @x= 20

Or: Select @x=20

```
Or: select @y=st_age,@name=st_fname from Student
where st_id=8
```

```
or: update Student set fname='Ali', @x=age where id=10//this will get the age of the updated row so
this query is considered update and select statement combined.
```

Display variable value

Select @x

Initializing variable from subquery

```
declare @x int=(select avg(st_age) from student)
```

Variables Notes

```
declare @y int=100
select @y=st_age from Student
where st_id=5000
select @y //if there is no student with id=5000 , the variable will hold its latest value (100) or Null
if the variable wasn't initialized.
```

```
declare @y int=100
select @y=st_age from Student
where st_address='alex'
select @y //if the query returns array of values(it's expected that there are many students in alex),
the variable will hold the last value in the array.
```

Array in SQL : it's represented using variable of type table

```
declare @t table(x int,y varchar(20))
insert into @t
select st_age,st_fname from Student
where st_address='alex' //runs first
select * from @t //display variable table

//this query will select age and name from student table then insert these values into the variabl
table @t in columns x(holds age), y(holds fname)
```

B) Global Variables

```
select @@SERVERNAME //display your server name

select @@version    //display your version of sql server


update student
set st_age+=1
select @@rowcount //display number of raws affected by the last query

select @@error //display number of error message caused by last query

select @@identity // returns the last identity value generated for any table in the current session and scope. Identity values are auto-generated values that are used to uniquely identify each row in a table.
```

Execute

```
execute( 'select * from Student')//execute converts this string into executable query

declare @col varchar(20)='ins_name'
       ,@t varchar(30)='instructor'
execute('select '+@col+' from '+@t)//this is equavilant to select int_name from instructor
```

Control of flow statements

If Statement

```
declare @x int=10
if @x>0
	begin
	select 'x>0'
	end
else
select 'x<0'
```

If exists

```
if exists(select name from sys.tables where name='student')
select * from student
else
select 'invalid table'
```

```
///(select name from sys.tables where name='student')this will check if the system contains a table called Student (returns true or false)
So if the table exist it will display its content, and if it doesn’t exist “invalid table” will be displayed.
```

Try & Catch

```
begin try
		delete from Department
		where dept_id=10
end try
begin catch
	select 'Dept has relationship'
	select ERROR_LINE(),ERROR_MESSAGE(),ERROR_NUMBER()
end catch
```

While Loop

```
declare @x int=10
while @x<20
	begin
		select @x+=1
		if @x=14
			continue
		if @x=16
			break
			select @x
	end
//11,12,13,15 will be printed
```

Case When

```
select ins_name,
    case gender
		when 'm' then 'Male'
		when 'f' then 'female'

    end
from Instructor
```

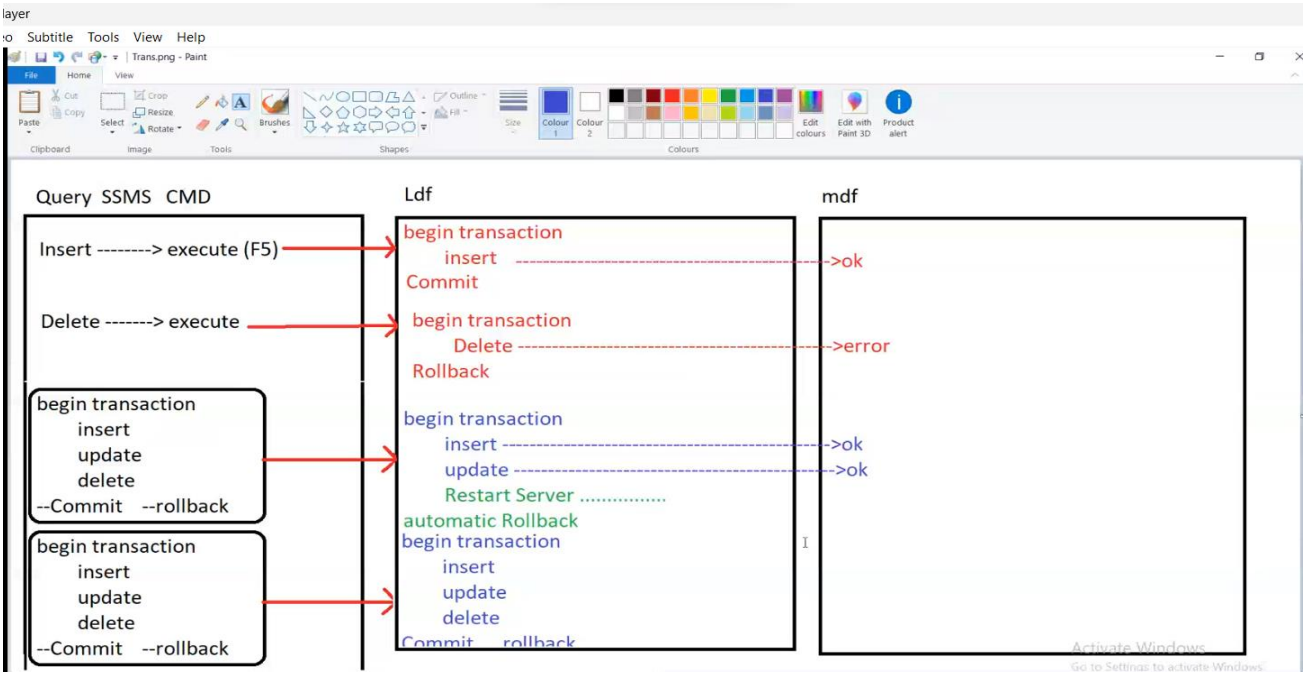
--iif
--choose
--waitfor

Batch vs Script

a)Batch
a series of one or more statements submitted and executed at the same time and don't depend on each others.

b)Script
script set of batches separated with go statement
//sometimes there are queries can’t be run together in one batch so we can separate them by go and this batch will be converted into script then we can run those queries

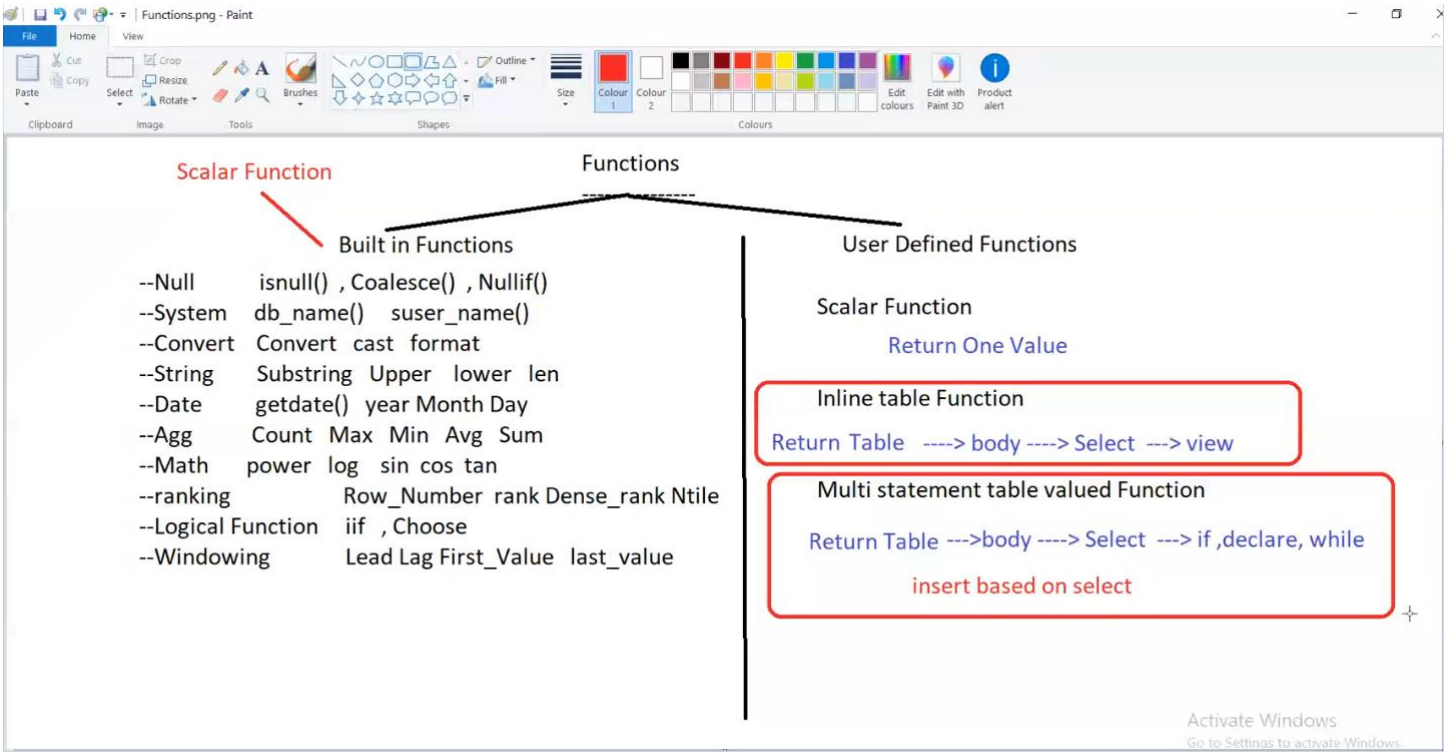
Transaction




```
--Transaction (check function sql file in day7 )
--a series of one or more data statements that executed as a unit or are aborted as a unit
--A transaction is a logical unit of work
--      Defined from business rules
--      Typically includes at least one data modification
--      Maintains DB consistency.
--What is:
--      Commit: is used when all queries in the transaction will be executed with no errors
--      Rolled back: is used when we daught that a query will cause an error
--      Save point    savepoint allows for partial rollbacks


//it’s better to use transaction with try catch
In try we will end transaction with commit so we make sure it will be executed with no runtime errors
In catch we will use rollback so it will rollback all statments if there is error
```

```
begin try
	begin tran
		insert into child values(1)
		insert into child values(5)
		insert into child values(2)
	commit tran
	print 'transaction commited'
end try
begin catch
	rollback
	print 'transation rolled back'
	select error_number() as "number",
	error_message() as "message",
	error_line() as "line"
end catch
```

Functions



a)Scalar function (return one value)

```
create function getsname(@id int)
 returns varchar(30) //function signature
 begin
	declare @name varchar(30)
	select @name=st_fname
	from Student
	where st_id=@id
	return @name
 end //function body

 select dbo.getsname(1) //function call

select *
 from Instructor
 where ins_name=dbo.getsname(1) //another function call
//notice: scalar functions must be called with schema name
```

b)inline function (return Table)[contain select statement only]

```
create function getins(@did int)
 returns table //function signature
 as
 return
 (
  select ins_name,salary*12 as total
  from Instructor
  where dept_id=@did
 ) //function body

 select ins_name from getins(10)
where total>9000 //function call
```

C) Multistatement table valued function

```
create function getstuds(@format varchar(10))
 returns @t table
                     (
                      id int,
                      name varchar(30)
                     )
AS
BEGIN
       if @format='first'
              insert into @t
              select st_id,st_fname from Student
       else if @format='last'
              insert into @t
              select st_id,st_lname from Student
       else if @format='full'
              insert into @t
              select st_id,st_fname+' '+st_lname from Student
return
END

select * from getstuds('FULL')
```

Windowing functions

```
SELECT st_fname,dept_id,st_age,
          lowest=FIRST_VALUE(st_age) OVER(PARTITION BY dept_id ORDER BY st_age ),
          Highest=LAST_VALUE(st_age) OVER(PARTITION BY dept_id ORDER BY st_age ROWS BETWEEN unbounded
preceding AND unbounded following),
          stud_prev=lAG(st_age) OVER(PARTITION BY dept_id ORDER BY st_age),
          stud_Next=LEAD(st_age) OVER(PARTITION BY dept_id ORDER BY st_age )
FROM student
```

Explaination

```
// lowest=FIRST_VALUE(st_age) OVER(PARTITION BY dept_id ORDER BY st_age ), this will order students
ages in each department and display the youngest student

// Highest=LAST_VALUE(st_age) OVER(PARTITION BY dept_id ORDER BY st_age), this will order students ages
in each department and display the oldest student



   //stud_prev=lAG(st_age) OVER(PARTITION BY dept_id ORDER BY st_age),this will order students ages in
each department and display younger student

//stud_Next=LEAD(st_age) OVER(PARTITION BY dept_id ORDER BY st_age ), this will order students ages in
each department and display older student
```

Lead & Lag (chatgpt)

The LAG function retrieves the value of a column from the previous row in the result set, while the LEAD function retrieves the value of a column from the next row in the result set. Both functions take two arguments: the first argument is the column to retrieve the value from, and the second argument is the offset, which specifies the number of rows to look ahead or behind.

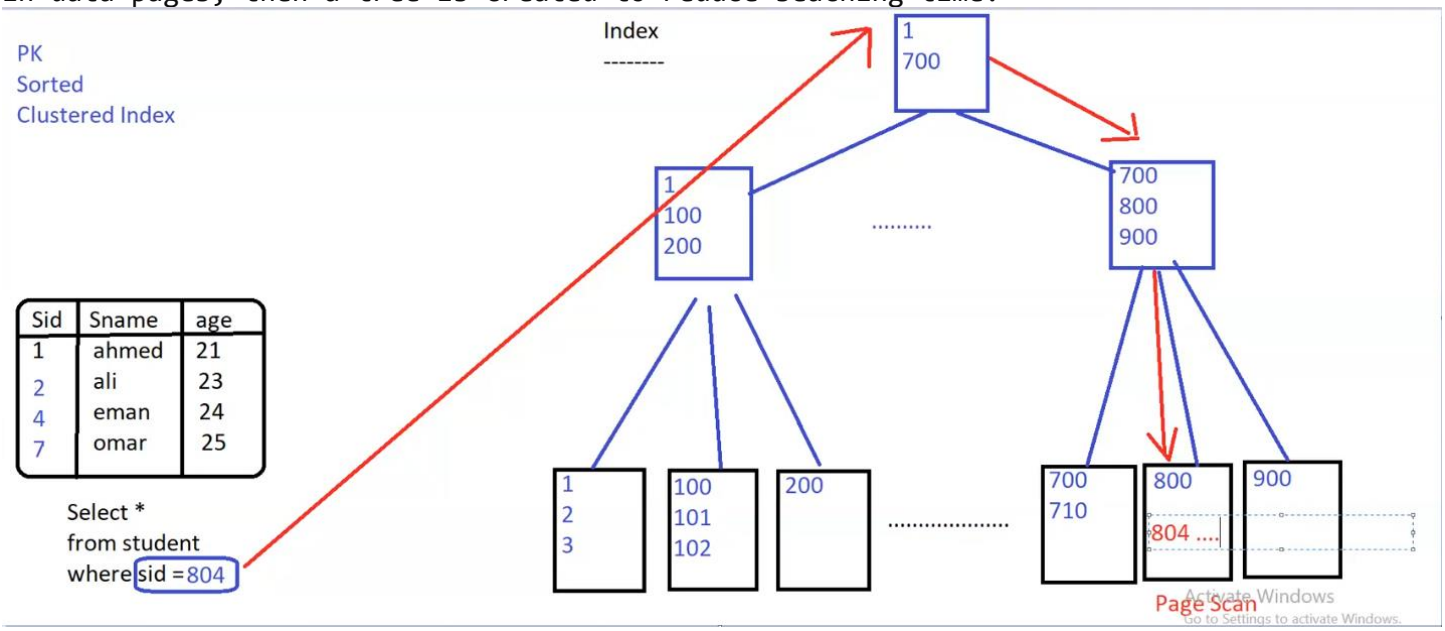
```
SELECT MyColumn, LAG(MyColumn, 1) OVER (ORDER BY MyDate) AS PreviousValue
FROM MyTable;
```

Index

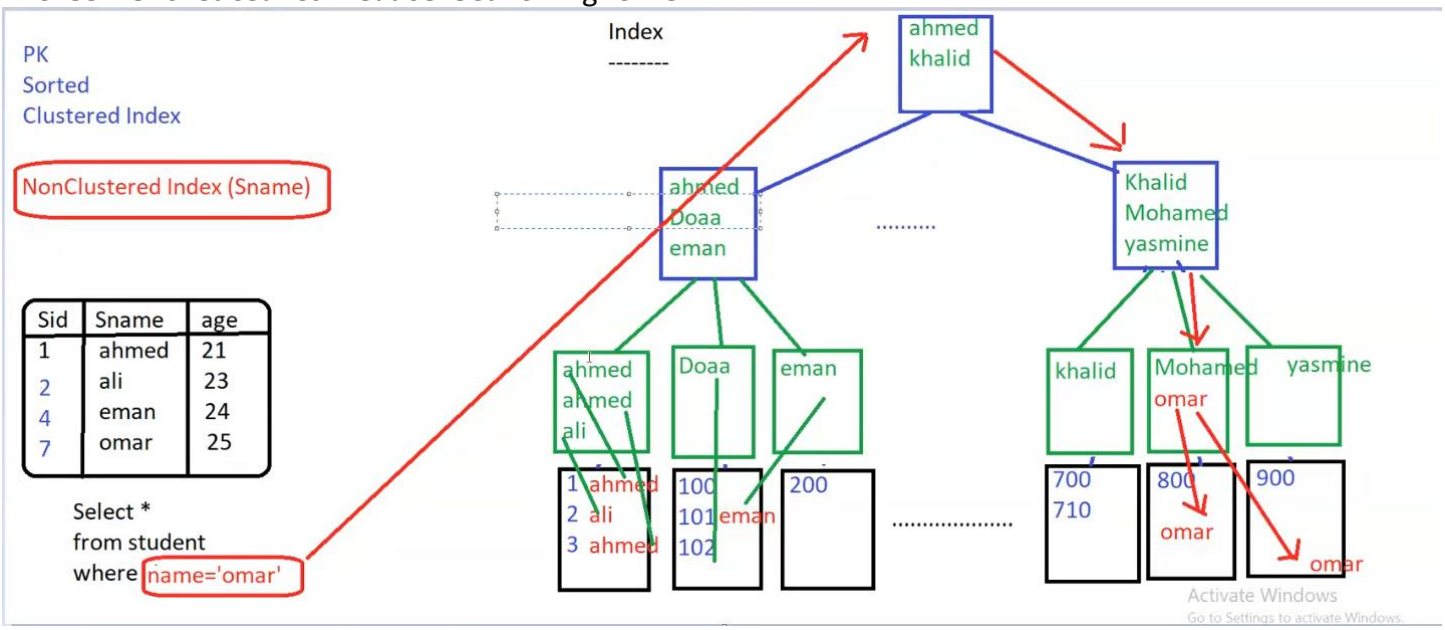
When we create a table without primary key, table data will be stored in heap (not sorted).
Se when we run a select query, table scan will be done(bad performance).

When we set a **primary key** to the table, then a **clustered index** will be generated automatically.

This Clustered index helps in enhancing select query performance by storing data sorted in data pages, then a tree is created to reduce seaching time.



We can create other non clustered indexes on other columns in the table
So this column data pages will be copied to the server memory with pointers to the actual data location.
A tree is created to reduce searching time



Notes:

- If a table doesn’t have a primary key, then the clustered index can be created on any column. After that if we set other column as primary key it will have non clustered index. Because there can be only one clustered index per table but there can be serveral nonclustered indexes in a table.

- Indexes enhance database performance with searching (select query) but not the best choice with DML queries, so be careful with using indexes.

- When we create a Unique constraint on a column, it automatically generate nonclustered index.

```
//creating nonclustered index
create nonclustered index i2
on student(st_fname)

---> Primary key constraint ===> clustered index
---> unique constraint ==== ===>nonclustered index

create unique index i4   ---->unique constraint + nonclustered index
on student(st_age) //notice age values must be unique
```

SQL Server System Databases

- **Master :** contain the server configuration, such as users and their username a password, if it’s corrupted you won’t be able to connect to the server.

- **Model:** Template database saved in the server, each newly created database, take an image of this Model database.
  --Notice: if you modify anything in this Model database content or configuration, this modifications will be applied only to the newly created databases not the existing ones.

- **Msdb:** responsible for jobs and alerts.

- **Tempdb:** contain temporary tables.

Table types in sql server

Physical table: this table is saved on the harddisk and won't be deleted unless you drop it

```
create table exam
(
 eid int,
 numofQ int,
 Edate date
)

drop table exam
```

table variable : is valid through the batch or the function

```
declare @exam table
(
 eid int,
 numofQ int,
 Edate date
)
insert into @exam values(1,7,'1/1/2023')
select * from @exam
```

Local Table : is saved in tempdb during the session and will be droped automatically if you run new query or disconnect with the server.

```
create table #exam
(
 eid int Primary key,
 numofQ int,
 Edate date
)
```

Global tables(shared tables): saved in tempdb during the session and can be accessed by multiple users during the session. Will be droped automaticaly if all the user disconnect from the server.

```
create table ##exam
(
 eid int,
 numofQ int,
 Edate date
)
```

Rollup & Cube (chatgpt)

ROLLUP and CUBE are two SQL grouping functions that allow you to group data based on multiple dimensions or attributes.

The ROLLUP function generates subtotals for each level of a hierarchy, and includes grand totals as the last row. For example, suppose you have a sales table that includes columns for region, product, and sales amount. You could use the ROLLUP function to generate subtotals for each region, each product, and each combination of region and product, as well as a grand total for all regions and products combined. Here's an example of how to use the ROLLUP function in SQL:

```
SELECT Region, Product, SUM(SalesAmount) AS TotalSales
FROM SalesTable
GROUP BY ROLLUP(Region, Product);
```

```
//Mine: in this example Rollup function will give the sum of Sales amount for each Region with each product
And it will add rows that give the subtotal sales amount for each region (because region is the fisrt attribute in Rollup function)
Notice: rollup function will add rows that give the subtotal sales amount for each product (if product was the fisrt attribute in Rollup function)
Finally it will add a row that gives the total SalesAmount over all the regions for all products
```

In this example, the ROLLUP function is used to generate subtotals for each combination of Region and Product, as well as a grand total for all regions and products combined.

The CUBE function generates subtotals for each combination of attributes, and includes a grand total for all attributes combined. For example, suppose you have a sales table that includes columns for region, product, and sales amount. You could use the CUBE function to generate subtotals for each region, each product, each combination of region and product, and a grand total for all regions, products, and combinations combined. Here's an example of how to use the CUBE function in SQL:

```
SELECT Region, Product, SUM(SalesAmount) AS TotalSales
FROM SalesTable
GROUP BY CUBE(Region, Product);
```

```
//Mine: in this example Cube function will give the sum of Sales amount for each Region with each product
And it will add rows that give the subtotal sales amount for each region and subtotal amount for each product (no matter which one is the first attribute as it gives the subtotal for both attributes)
Finally it will add a row that gives the total SalesAmount over all the regions for all
```

In this example, the CUBE function is used to generate subtotals for each combination of Region and Product, as well as subtotals for each Region and each Product, and a grand total for all regions and products combined.

Overall, the ROLLUP and CUBE functions are useful for generating complex summary reports that include multiple levels of aggregation and subtotals, and can help you gain insights into large datasets with many dimensions and attributes.

Pivot & unpivot (chatgpt)

PIVOT and UNPIVOT are two SQL operations that allow you to transform data from rows into columns and vice versa. These operations are commonly used in data analysis and reporting.

PIVOT is used to transform rows of data into columns, based on a specified pivot column or set of columns. For example, suppose you have a sales table that includes columns for region, product, and sales amount. You could use the PIVOT operation to transform the data so that each region is listed as a separate column, with the sales amount for each product listed under each region. Here's an example of how to use the PIVOT operation in SQL:

```
SELECT *
FROM SalesTable
PIVOT (
  SUM(SalesAmount)
  FOR Region IN ('East', 'West', 'North', 'South')
);
```

In this example, the PIVOT operation is used to transform the sales data so that each region is listed as a separate column, with the sales amount for each product listed under each region.

UNPIVOT is used to transform columns of data into rows, based on a specified set of columns. For example, suppose you have a sales table that includes columns for region, product, and sales amount, with each region listed as a separate column. You could use the UNPIVOT operation to transform the data so that each region is listed as a separate row, with the sales amount for each product listed in a separate column. Here's an example of how to use the UNPIVOT operation in SQL:

```
SELECT Region, Product, SalesAmount
FROM SalesTable
UNPIVOT (
  SalesAmount FOR Region IN ('East', 'West', 'North', 'South')
);
```

In this example, the UNPIVOT operation is used to transform the sales data so that each region is listed as a separate row, with the sales amount for each product listed in a separate column.

Overall, the PIVOT and UNPIVOT operations are useful for transforming data between rows and columns, and can help you generate reports and analyses that are more organized and easier to read.

View

- A view is a virtual table used to provide a layer of security.
- It's used to specify user view of data and hide DB objects(Hide names of database objects).
- It can be used to simplify construction of complex queries
- Unlike functions, views don't have parameters
- We can't write DML queries in views body.

View Types:

- **Standard view:**is a virtual table
- **Partitioned view:** used to select data from different servers
- **Indexed view:** the only view that is used to enhance performance.

Standard view example

```
create view v2(studept_name,dept_name)
as
	select st_fname,dept_name
	from student s,department d
	where  s.dept_id=d.dept_id
	and s.st_address='cairo'

//if we run sp_helptext v2 we can get the code of the view, so it's better to encrypt view

create view v2(studept_name,dept_name)
WITH ENCRYPTION
as
	select st_fname,dept_name
	from student s,department d
	where  s.dept_id=d.dept_id
	and s.st_address='cairo'
```

We can't write DML queries in View body but we can run DML queries on View as if it was a table

Runnig DML query on View that get data from one table

We can run delete statement
We can run insert or update statement on columns that are included in the view, but you must make sure that other columns that aren't included in the view have (allow Null, default value, identity,) .

```
Create view v13
as
	select st_id,st_fname,st_age
	from student
	where st_age=20
	with check option //this means we can only insert ages that match the condition (st_age=20)

insert into v13
values(111,'ali',33)//this query won't run because we're only allowed to insert student with age 20
```

Runnig DML query on View that get data from multiple table (join)

We can't run delete statement (because we can't delete two rows from different tables at the same time).
We can run insert or update statement only if data is inserted or updated into one table.

Notice:
We must write schema name before table name in the following cases:

- Scaler function
- Indexed view
- Running query on data from different server.

Merge Statement (chatgpt)

The `MERGE` statement in SQL is used to perform an operation that combines an `INSERT`, `UPDATE`, or `DELETE` statement into one statement. It provides a way to conditionally insert or update data into a table based on a matching condition.(compare two tables)

The syntax for the `MERGE` statement is as follows:

```
MERGE INTO target_table AS T
USING source_table AS S
ON T.column_name = S.column_name
WHEN MATCHED THEN
    UPDATE SET T.column1 = S.column1, T.column2 = S.column2
WHEN NOT MATCHED THEN
    INSERT (column1, column2) VALUES (S.column1, S.column2);
```

Here's how the `MERGE` statement works:

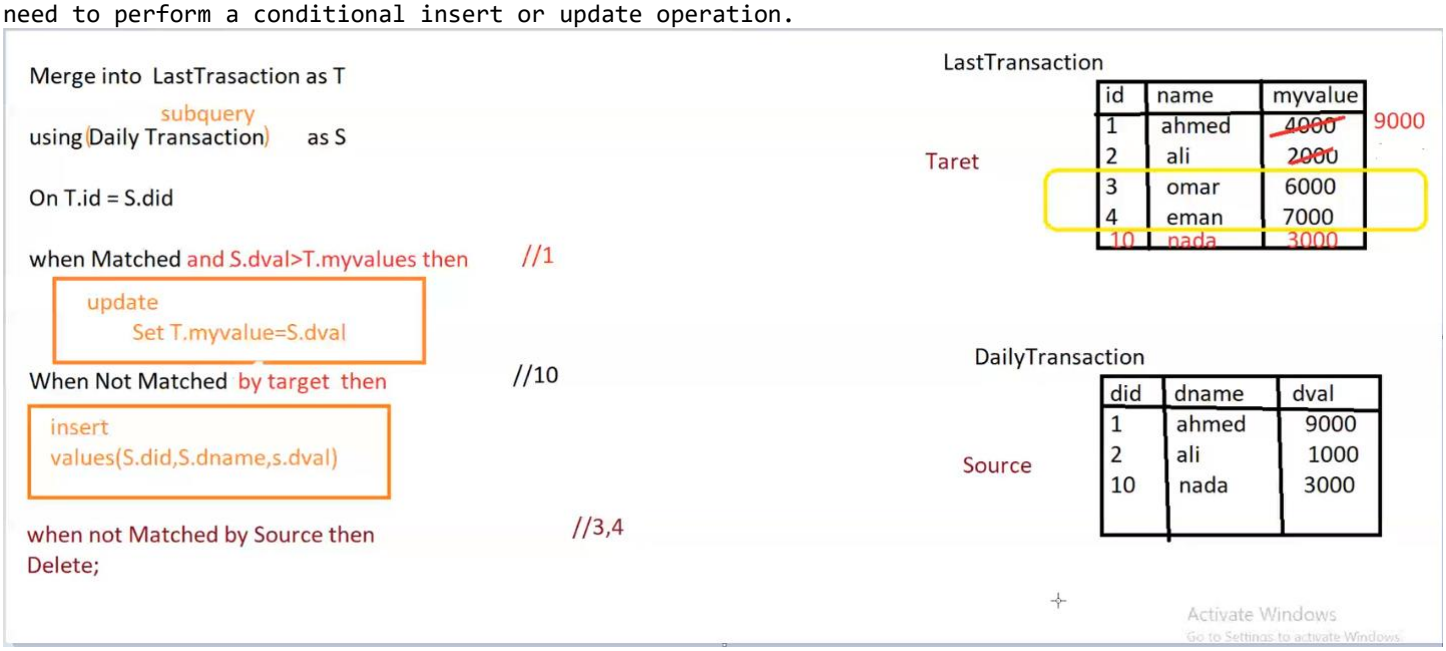
1. The `target_table` is the table you want to merge data into, and `source_table` is the table that contains the data you want to merge.

2. The `ON` clause specifies the matching condition to be used for the merge. This condition compares the values of the columns in the target table and the source table.

3. The `WHEN MATCHED` clause specifies the action to be taken when a match is found between the target and source tables. In this case, the statement updates the values of the target table using the values from the source table.

4. The `WHEN NOT MATCHED` clause specifies the action to be taken when no match is found between the target and source tables. In this case, the statement inserts the values from the source table into the target table.

The `MERGE` statement is often used in data warehousing scenarios where you want to update or insert data in a table based on the data in another table. It can also be used in other scenarios where you need to perform a conditional insert or update operation.



Stored Procedure

Types of stored procedure:

- Built-in SP
- User defined SP
- Triggers

In SQL, a stored procedure is a pre-written set of SQL statements that is compiled and stored in the database for later execution. It is a programming block that can be called by name to perform a specific task or set of tasks. Stored procedures can be created using SQL or a procedural programming language such as PL/SQL, T-SQL, or PL/pgSQL.

Stored procedures provide a number of benefits, including:

1. Reusability: Once created, a stored procedure can be called multiple times by different applications, making it a reusable code block.

2. Performance: Stored procedures can improve performance by reducing network traffic between the application and the database server, since the SQL code is executed on the server side.

3. Security: Stored procedures can help improve security by allowing database administrators to restrict access to specific procedures instead of entire tables or databases.
--also hide table names and other database objects
-- it hide the business logic.

4. Modularity: Stored procedures can be modularized, making it easier to update and maintain code.

5. Transaction management: Stored procedures can be used to group multiple SQL statements into a single transaction, ensuring data consistency and integrity.

6. We can handle errors in stored procedures using if statements or try & catch for example so we can prevent runtime errors in the applications.

Stored procedures can accept input parameters, return output parameters, or both. They can also be used to create temporary tables, control program flow with conditional logic and loops, and call other stored procedures.

Normal query lifecycle

Query ➔ Parsing (making sure query syntax is correct) ➔ Optimizing (making sure query metadata is provided correctly) ➔ Query Tree (From Where Select) ➔ Execution Plan (execute query in memory).

When we call a stored procedure for the first time, it goes through the entire query lifecycle but when we call it againg it gets executed directly because query tree is saved in memory which enhances performance.

Stored Procedure vs Function

Return statement is mandatory in function and we can return any data type.
Return is optional in SP, Return type in SP is only integer and it’s used to indicate SP state.

Fo example Database developer and application developer will have an agreement that if the SP returns 10 this means SP run successfully
But if it returns 15 it means there is an error
It enhances appliaction security.

Notice :

We can return values from SP in form of varaibles using **Output** keyword

```
create Proc getvalues @id int,@age int output,@name varchar(20) output
as
              select @age=st_age,@name=st_fname from student
              where st_id=@id

declare @x int,@y varchar(20)
execute getvalues 3,@x output,@y output
select @x,@y
```

Parameters Types in SP:

- Input parameter
- Output parameter
- Input Output parameter
- Return parameter

We can encrypt SP code using **with Encryption** keyword

Trigger (special Stored Procedure)

```
--special type of SP
--implicit code
--can't call
--can't take paramters
--run according actions
-----> triggers    table    insert update  delete     ok       select,truncate  XXXx


create trigger tr_1
on student
after insert
as
       select 'welcome to ITI'



create trigger tr_3
on student
instead of delete
as
       select 'not allowed'
```

Each fire of a trigger creates two tables inserted, deleted both of them have the same signature as the table which the trigger is created on.

If we run insert statement:
The inserted values will be stored in the inserted table and the deleted table will be empty

If we run delete statement
The deleted values will be stored in the deleted table and the inserted table will be empty

If we run update statement
The new values will be stored in the inserted table and the old values will be stored in the deleted table.

We can access inserted and deleted tables with triggers and output keyword(create runtime trigger).

```
//triggers can be enabled or disabled or droped
alter table department disable trigger tr_4
alter table department enable trigger tr_4
```

output keyword

```
update instructor
       set ins_id=500
output suser_name(),getdate(),deleted.ins_id,inserted.ins_id  into history
where ins_id=777


//this query will insert username, date, old id (from deleted), new id (from inserted table) into
history table to audit this process.
```

Trigger (chatgpt)

In SQL, a trigger is a set of SQL statements that are automatically executed in response to certain events or actions performed on a table, such as an `INSERT`, `UPDATE`, or `DELETE` operation. A trigger is a database object that is associated with a table and is activated when a specified event occurs for that table.

Triggers can be used for a variety of purposes, including:

1. Auditing: Triggers can be used to track changes to the data in a table by recording information such as the user who made the change and the time of the change.

2. Enforcing business rules: Triggers can be used to enforce business rules by preventing changes to a table that violate those rules.

3. Replication: Triggers can be used to replicate data to other databases or tables.

4. Complex calculations: Triggers can be used to perform complex calculations or data transformations on the data in a table.

The syntax for creating a trigger is as follows:

```
CREATE TRIGGER trigger_name
{BEFORE | AFTER | instead of} {INSERT | UPDATE | DELETE}
ON table_name
[FOR EACH ROW]
BEGIN
   -- SQL statements
END;
```

Here's how a trigger works:

1. When an event specified in the trigger occurs on a table, the trigger is automatically activated.

2. The SQL statements specified in the trigger are executed, typically manipulating data in the table or performing other actions.

3. The trigger can also include conditional logic to determine which actions to take based on the data being modified.

Triggers can be created using SQL or a procedural programming language such as PL/SQL, T-SQL, or PL/pgSQL.

Output keyword (runtime trigger)

In SQL, the `OUTPUT` keyword is used in conjunction with `INSERT`, `UPDATE`, and `DELETE` statements to return the values that were affected by the statement.

When you use the `OUTPUT` keyword with an `INSERT`, `UPDATE`, or `DELETE` statement, the data that was inserted, updated, or deleted is stored in a temporary table that is created in memory. You can then use the `SELECT` statement to query the temporary table and return the data that was affected.

Here's an example of how to use the `OUTPUT` keyword with an `INSERT` statement:

```
INSERT INTO MyTable (Column1, Column2, Column3)
OUTPUT INSERTED.Column1, INSERTED.Column2
VALUES ('Value1', 'Value2', 'Value3');
```

In this example, an `INSERT` statement is used to insert a new row into the `MyTable` table. The `OUTPUT` keyword is used to return the values of `Column1` and `Column2` for the row that was just inserted.

The result set will contain the values that were inserted into `Column1` and `Column2`. The `OUTPUT` keyword can also be used with `UPDATE` and `DELETE` statements in a similar way to return the values that were updated or deleted.

The `OUTPUT` keyword can be useful for auditing purposes, or for returning data to an application after a data modification operation has been performed.

Cursor

In SQL, a cursor is a database object **used to retrieve and manipulate data row by row** from a result set. It provides a way to iterate over the rows returned by a query and perform operations on each row individually. Cursors are commonly used in situations where you need to process or manipulate data row by row rather than as a whole result set.

Here's a general process of working with a cursor in SQL:

1. Declare the cursor: You start by declaring a cursor and associating it with a SELECT statement that defines the result set you want to work with. You can specify any necessary filtering, sorting, or joining conditions in the SELECT statement.

2. Open the cursor: Once the cursor is declared, you need to open it to establish the result set. This step makes the result set available for retrieval.

3. Fetch the rows: After opening the cursor, you can fetch the rows one by one or in a batch. Fetching retrieves the current row from the result set and allows you to perform operations on it.

4. Process the rows: Once you have fetched a row, you can perform operations on it, such as reading the values, updating or deleting the row, or performing calculations based on the row's data.

5. Repeat fetch and process: You continue to fetch and process rows until you have processed all the rows in the result set.

6. Close the cursor: After you have finished working with the cursor, you need to close it to release the resources associated with it and free up memory.

Here's an example that demonstrates the basic usage of a cursor in SQL:

```
DECLARE @EmployeeID INT;
DECLARE @EmployeeName VARCHAR(50);

DECLARE MyCursor CURSOR FOR
SELECT EmployeeID, EmployeeName
FROM Employees;

OPEN MyCursor;

FETCH NEXT FROM MyCursor INTO @EmployeeID, @EmployeeName;

WHILE @@FETCH_STATUS = 0
BEGIN
    -- Perform operations on the current row
    -- Access @EmployeeID and @EmployeeName

    -- Fetch the next row
    FETCH NEXT FROM MyCursor INTO @EmployeeID, @EmployeeName;
END;

CLOSE MyCursor;
DEALLOCATE MyCursor;
```

In this example, a cursor named `MyCursor` is declared and associated with a SELECT statement that retrieves `EmployeeID` and `EmployeeName` from the `Employees` table. The cursor is then opened, and the rows are fetched one by one using a `WHILE` loop. Within the loop, you can access and process the values of each row.

It's important to note that cursors can have performance implications and should be used judiciously. In many cases, set-based operations using SQL's inherent capabilities can provide more efficient solutions. Cursors are typically used when row-by-row processing is necessary or when there are complex logic or business requirements that cannot be easily achieved with set-based operations.

Cursor Examples

1)

```
declare c1 cursor //declare cursor
for select distinct st_fname from student where st_fname is not null //specify which select statement
to loop for
for read only //this cursor is used only to display query result

declare @name varchar(20),@all_names varchar(300)='' //declare variables to hold the returned values
open c1
fetch c1 into @name     --counter=0
while @@FETCH_STATUS=0
       begin
              set @all_names=CONCAT(@all_names,',',@name)
              fetch c1 into @name   --counter++
       end
select @all_names //display all names concatenated
close c1  //notice: we can close the cursor during the loop and reopen it to start from where it stoped
deallocate c1  //finally we deallocate the cursor to free up memory space

//this cursor is used to select students first name from table student and concate these names in
one variable
```

2)

```
declare c1 cursor //declare cursor
for select salary from instructor
for update //this cursor is used to manipulate table instructor data (insert, update, delete)

declare @sal int //declare variable to hold  the salary of each iteration
open c1
fetch c1 into @sal
while @@FETCH_STATUS=0
       begin
              if @sal>=3000
                     update instructor    --1 row affected
                            set salary=@sal*1.20
                     where current of c1 //to update only the row at this iteration not the whole table
              else if @sal<3000
                     update instructor
                            set salary=@sal*1.10
                     where current of c1
              else
                     delete from instructor
                     where current of c1
              fetch c1 into @sal //counter++, if we don't write this line we go in infinte loop
       end
close c1
deallocate c1
//this cursor is used to loop over instructors table and update the salary for instructors row by row
under specific conditions
```

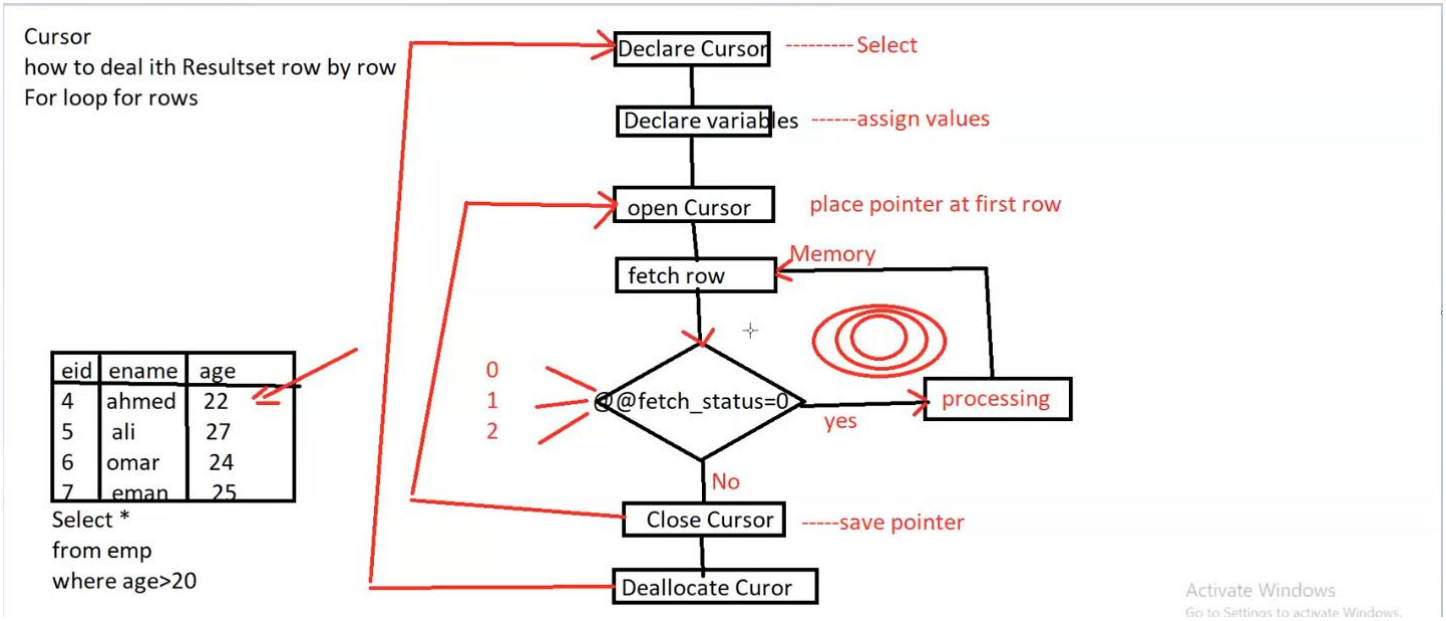
Fetch Status

In SQL, when using a cursor, the FETCH statement is used to retrieve rows from the result set of the cursor. After executing the FETCH statement, the cursor's status is updated to indicate the result of the fetch operation. The cursor status can be checked using the @@FETCH_STATUS system variable.

The @@FETCH_STATUS variable returns an integer value that represents the status of the most recent FETCH operation. Here are the commonly used values of @@FETCH_STATUS and their meanings:

- -1: The FETCH operation failed or the row was not found.
- 0: The FETCH operation was successful, and a row was fetched.
- 1: The FETCH operation was successful, and the row was not found (end of the cursor result set).
- 2: The FETCH operation was successful, but the row fetched was a duplicate of a previously fetched row when using a SCROLL cursor.

You can use the @@FETCH_STATUS variable in a loop to iterate through the result set of a cursor and perform certain actions based on the fetch status. Here's an example:



Backup types (chatgpt)

There are several types of backups commonly used in database management:

1. Full Backup: A full backup involves making a copy of the entire database, including all its data and objects. It provides a complete snapshot of the database at a specific point in time. Full backups are typically performed periodically to create a baseline for other types of backups.

2. Differential Backup: A differential backup captures only the changes made to the database since the last full backup. It includes all changes made after the full backup, making it faster and smaller in size than a full backup. Differential backups can be used to restore the database to a specific point in time by applying the last full backup and the most recent differential backup.

3. Incremental Backup: An incremental backup captures only the changes made to the database since the last backup, whether it is a full backup or an incremental backup. It stores the changes made since the last backup, making it smaller and faster than full or differential backups. Incremental backups require a series of backup files to restore the database to a specific point in time.

4. Transaction Log Backup: A transaction log backup captures the log records(from .ldf file) that record changes to the database since the last transaction log backup. It is used in conjunction with the full or differential backup to allow point-in-time recovery. Transaction log backups enable recovery of the database to a specific transaction or point in time.

5. Copy-Only Backup: A copy-only backup is an ad-hoc backup that doesn't affect the normal backup and restore sequence. It creates a copy of the database without affecting the backup chain or the subsequent backups. Copy-only backups are useful for creating backups for specific purposes, such as testing or development, without interrupting the regular backup strategy.

6. Partial Backup: A partial backup backs up only a specified subset of the database, such as specific filegroups or specific tables. It is useful when only a portion of the database requires backup and can help reduce backup time and storage requirements.

Each type of backup serves a different purpose and is suited for different scenarios. Organizations often use a combination of these backup types to ensure data protection, minimize downtime, and meet recovery objectives. The choice of backup strategy depends on factors such as the size of the database, recovery time objectives, available resources, and business requirements.

Full backup command

```
backup database SD
to disk='d:\SD_db.bak'
```

Job

In SQL, a job refers to a scheduled task or a series of tasks that are executed automatically at specified intervals or on specific events within a database management system. Jobs are used to automate routine database operations, data maintenance, report generation, and other administrative tasks.

Here are some key aspects of jobs in SQL:

1. Job Scheduler: SQL databases typically provide a job scheduling subsystem that allows you to define and manage jobs. This subsystem includes features for scheduling jobs based on time, event triggers, or dependencies.

2. Job Definition: A job is defined by specifying the tasks or commands to be executed, the schedule or event trigger for execution, and any additional parameters or options required for the job. Jobs can be written using SQL statements, stored procedures, or scripts in languages supported by the database system (such as T-SQL for SQL Server or PL/SQL for Oracle).

3. Job Execution: Once a job is scheduled, the job scheduler takes care of executing the job at the specified time or when the triggering event occurs. The job scheduler manages the job queue, prioritizes jobs, and ensures that jobs are executed according to their schedules and dependencies.

4. Job Monitoring and Logging: SQL databases typically provide tools and utilities to monitor and track the execution of jobs. Job logs or status reports can be generated to provide information about job success or failure, execution time, resource usage, and any error messages or notifications.

5. Job Management: Job management features allow you to view, modify, enable/disable, and delete jobs. You can also configure job dependencies, define job alerts or notifications, and set up job history retention policies.

6. Examples of Job Tasks: Jobs can be used for a variety of tasks, such as data backups, database maintenance tasks (index rebuilds, statistics updates), data imports/exports, report generation, data synchronization, and automated data processing.

Jobs provide a convenient way to automate repetitive tasks, ensure consistency in database operations, and improve overall system efficiency. By scheduling jobs, database administrators can offload routine tasks, reduce manual effort, and focus on more critical aspects of database management.

Notice: To use jobs you have to start **SQL Server Agent**

Go to job and create a new job specify its name, steps, schedule and other configuration

Identity Insertion

In SQL, the `IDENTITY_INSERT` option is used to control whether explicit values can be inserted into an identity column of a table. By default, the `IDENTITY_INSERT` option is set to `OFF`, which means that explicit values cannot be inserted into an identity column. However, you can use the `SET IDENTITY_INSERT` statement to turn this option `ON` or `OFF` for a specific table.

Here's the syntax to enable or disable the `IDENTITY_INSERT` option:

To enable `IDENTITY_INSERT`:

```sql
SET IDENTITY_INSERT table_name ON;
```

To disable `IDENTITY_INSERT`:

```sql
SET IDENTITY_INSERT table_name OFF;
```

In the above syntax, `table_name` should be replaced with the name of the table for which you want to enable or disable the `IDENTITY_INSERT` option.

When `IDENTITY_INSERT` is set to `ON`, you can explicitly insert values into the identity column of the specified table. This is useful in scenarios where you need to insert specific values into an identity column, such as during data migration or when synchronizing data between databases.

It's important to note that enabling `IDENTITY_INSERT` requires appropriate permissions. You must have the `ALTER` permission on the table, and you may need additional permissions depending on your database system and security settings.

Remember to turn `IDENTITY_INSERT` back `OFF` after you have completed the necessary inserts into the identity column to ensure that the automatic identity generation resumes for subsequent inserts.

Bulk insert

In SQL, the `BULK INSERT` statement is used to efficiently insert large amounts of data from an external data file into a table. It is commonly used when you have a file containing structured data that you want to import into a SQL table.

The basic syntax of the `BULK INSERT` statement is as follows:

```sql
BULK INSERT target_table
FROM 'data_file'
WITH (
    [OPTIONS]
);
```

Here's a breakdown of the main components of the `BULK INSERT` statement:

- **`target_table`:** This is the name of the table where you want to insert the data.
- **`data_file`:** This is the path and filename of the data file that contains the data you want to import.
- **`OPTIONS`:** This is an optional section where you can specify various options for the bulk insert operation, such as field and row terminators, data format specifications, error handling, and more.

Here's an example that demonstrates how to use the `BULK INSERT` statement:

```sql
BULK INSERT Employees
FROM 'C:\data\employees.csv'
WITH (
    FIELDTERMINATOR = ',',
    ROWTERMINATOR = '\n',
    FIRSTROW = 2,
    BATCHSIZE = 1000,
    ERRORFILE = 'C:\data\bulk_insert_errors.txt'
);
```

In the above example, the data from the 'employees.csv' file is bulk-inserted into the `Employees` table. The file is assumed to be comma-separated, with newline as the row terminator. The `FIRSTROW` option is used to skip the header row in the file. The `BATCHSIZE` option specifies the number of rows to be processed at a time. The `ERRORFILE` option specifies the path and filename where any error records during the bulk insert will be written.

It's important to ensure that the structure of the data file matches the structure of the target table, including the number and order of columns. Additionally, appropriate permissions and access rights are required to perform the bulk insert operation.

The `BULK INSERT` statement can be a powerful tool for efficiently importing large volumes of data into SQL tables. It is often used for data warehousing, data migration, or any scenario where high-performance data loading is required.

Snapshot (chatgpt)

In SQL, a snapshot refers to a read-only, static view of a database at a specific point in time. It provides a consistent and isolated copy of the data as it existed at the time the snapshot was created. Snapshots are primarily used for reporting, analysis, and data consistency purposes.

Here are some key points about snapshots in SQL:

1. Read-Only View: A snapshot is a read-only copy of a database or a specific subset of its data. It allows users to query the data without impacting the ongoing transactions or modifications in the original database.

2. Point-in-Time View: A snapshot captures the data at a specific point in time, reflecting the state of the database when the snapshot was created. This is useful for historical analysis or for viewing data as it existed at a particular moment, even if subsequent changes have been made to the original database.

3. Data Consistency: Snapshots provide data consistency because they represent a consistent view of the database at the time of creation. This ensures that queries against the snapshot produce accurate and coherent results.

4. Data Integrity: Snapshots maintain data integrity by using read-locking mechanisms to prevent concurrent modifications or updates to the data being viewed. This ensures that the snapshot remains stable and consistent throughout the query process.

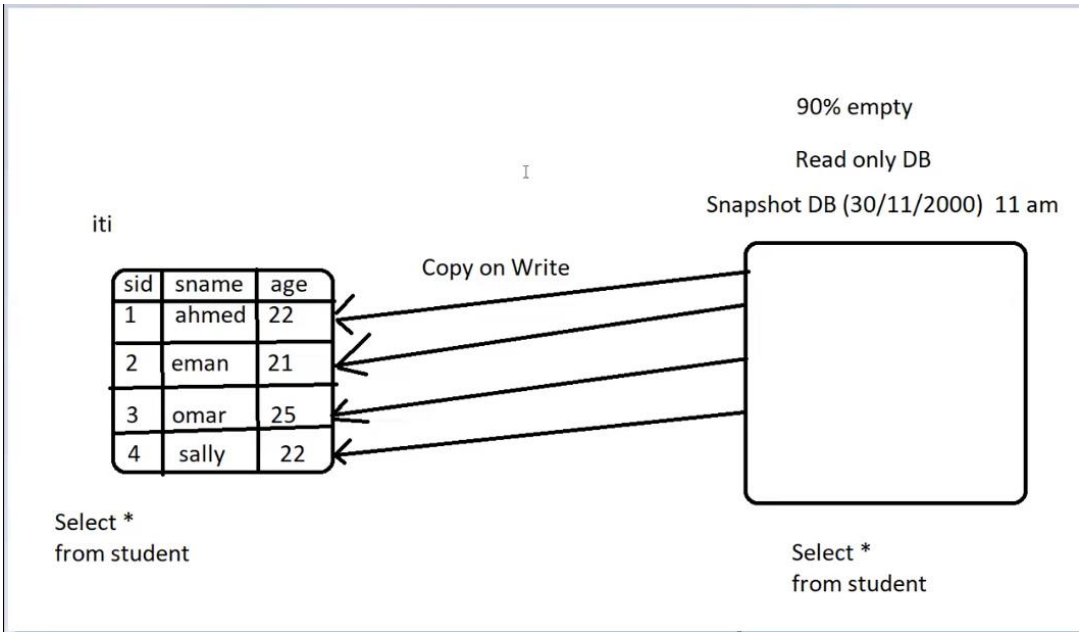
5. Optimized Performance: Since snapshots are read-only, they can be optimized for read operations, allowing for improved performance when querying large volumes of data or complex queries. Snapshots can be indexed or partitioned to further enhance query performance.

6. Types of Snapshots: SQL databases offer different types of snapshots, such as database snapshots (which capture the entire database) or schema-level snapshots (which capture a specific schema or subset of tables). Some databases also support snapshot isolation levels, which provide transaction-level snapshots for concurrent access.

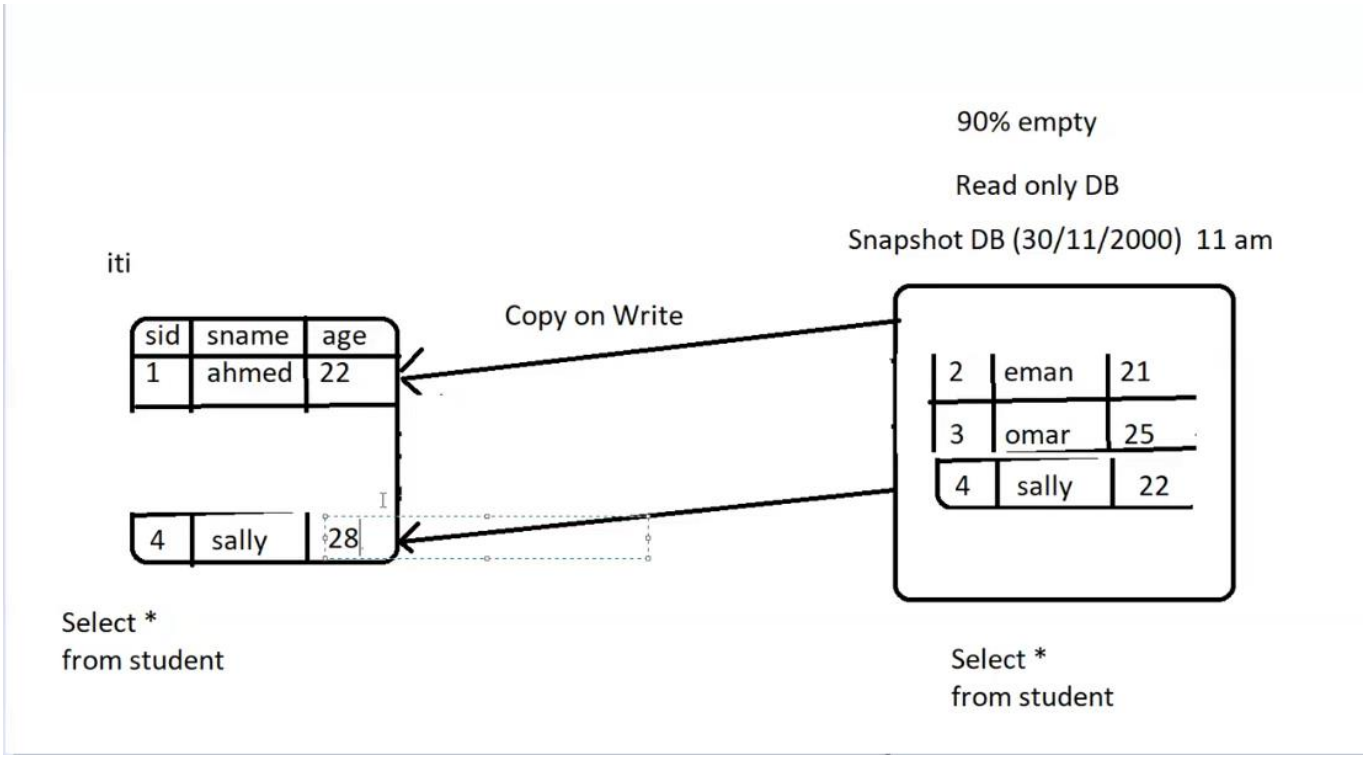
Snapshots provide a valuable mechanism for analyzing historical data, generating reports, and maintaining data integrity in SQL databases. They offer a consistent and isolated view of the database, ensuring that queries against the snapshot yield accurate results without interfering with ongoing database operations.

```
Create database Snap1
On
(
 name='iti',  --mdf
 filename='d:\s1.ss'
)
as snapshot of ITI
```

Notes



//When we create a snapshot, it basically has pointers to data location on the harddisk



//but if the table data is changed, then a copy of the changed rows will be saved in the snapshot so that we can get the table state when the snapshot was taken