<>-not equal sign

**Types of Sql Statements**

SQL commands are mainly categorized into four categories as:

1. DDL–Data Definition Language(create,drop,alter,truncate,rename,comment)
2. DQl – Data Query Language
3. DML – Data Manipulation Language
4. DCL – Data Control Language



**TRUNCATE**

TRUNCATE: This is used to remove all records from a table, including all spaces allocated for the records are removed.

truncate table GraduatedStudents

Now all the records are removed. Only the structure of the tables is preserved.

**Creating a table**

Create database databasename;

create table students(

student\_id int primary key,

name varchar(20),

major varchar(20)

)

Here we gotta set primary keys manually which is not right:

insert into students values(1,'Vako',20);

insert into students values(2,'Vako',20);

But with identity keyword it sets that automatically and starts from zero.

create table people(

person\_id int primary key identity,

name varchar(20),

age int

)

So we would just do:

insert into people values('Vako',20);

insert into people values('Lala',25);

That’s it!

**Comments**

--our comment or for multiline comments we use /\* \*/

**Alter a table(adding a new column)**

if we want to change a table we use alter keyword!

If we forgot to add a column then we can use ‘alter’ keyword:

Alter table tableName add columnName varchar(16)

alter table students add gpa int

then we use ‘update’ keyword to set the values because they are null.

Update tableName set column1=value1, column2=value2

where condition

update students set gpa=7 where student\_id=3

**SQL constraints (NOT NULL, UNIQUE)**

If we wanna add a required column meaning that it cannot be null then we use **not null** attribute:

create table people(

id int primary key identity,

[name] varchar(16) not null,

age int unique

)

So now there can’t be equal ages, and name cannot be omitted however it can be an empty string.

insert into people values('Vahid',20)

insert into people values('Hey',20)

Violation of UNIQUE KEY constraint

We can also add contsraints as shown below

CREATE TABLE Persons (  
    ID int NOT NULL,  
    LastName varchar(255) NOT NULL,  
    FirstName varchar(255),  
    Age int,  
    City varchar(255),  
    CONSTRAINT CHK\_Person CHECK (Age>=18 AND City='Sandnes')  
);

CREATE TABLE dbo.Bulk\_PortIn(

ID int PRIMARY KEY,

BatchID int,

CONSTRAINT FK\_\_Bulk\_Port\_Id\_\_Batch\_Id FOREIGN KEY (BatchID) REFERENCES dbo.Bulk\_Statistics(ID)

);

**Check constraint**

The CHECK constraint is used to limit the value range that can be placed in a column.

create table Students(

Id int primary key identity,

Name varchar(16) check(name like '%a%')

)

insert into Students values('Hey')

The INSERT statement conflicted with the CHECK constraint "CK\_\_Students\_\_Name\_\_3A81B327".

It has to contain an a in it!

create table Students(

Id int primary key identity,

Name varchar(16) check(name like '%a%'),

Age int check(Age>1 and Age<10)

)

insert into Students values('Hay',9)

**Naming Convetion for Constraints**

PK\_TableName\_ColumnName -- primary key constraint FK\_TableName\_ColumnName - foreign key constraint CK\_TableName\_ColumnName - check constraint

UQ - for unique constraint

DF - for default constraint

**Deleting a record(row) from a table**

We can use just **delete** keyword here:

delete from people where [name]=''

**Foreign key**

A FOREIGN KEY is a field (or collection of fields) in one table, that refers to the PRIMARY KEY in another table.

create table pets(

pet\_id int primary key identity,

name varchar(16) not null

)

alter table people add pet\_id int foreign key references pets(pet\_id)

**Horizontal Joins**

A JOIN clause is used to combine rows from two or more tables, based on a related column between them.

Inner join -  selects records that have matching values in both tables

LEFT (OUTER) JOIN: Returns all records from the left table, and the matched records from the right table (if left table has more fields then right table’s fields will be set to null)

RIGHT (OUTER) JOIN: Returns all records from the right table, and the matched records from the left table

FULL (OUTER) JOIN: Returns all records when there is a match in either left or right table



select \*

from people

inner join pets

on people.pet\_id=pets.pet\_id

Self Join-A self join is a regular join, but the table is joined with itself. it is only used in self-referencing relationships. We specify it by using simple inner, right or left joins.

create table Students(

Id int primary key identity,

Name varchar(16)

)

alter table Students add Dependency\_id int references Students(id)

select d.Name,d.Dependency\_id,s.Name as 'Depends on' from Students as s inner join Students d on s.Id=d.Dependency\_id



Non-equal join-When you join two tables using other conditional operators, beyond the equal sign, non-equil JOINs come into play. Comparison operators, like <, >, <=, >=, !=, and <> and the BETWEEN operator work perfectly for joining tables in SQL.

create table Students(

Id int primary key identity,

Name varchar(16),

Age int

)

create table GraduatedStudents(

Id int primary key identity,

Name varchar(16),

Age int

)

insert into Students values('Vahid',20),('Nihad',25),('Azer',18),('Lala',35)

insert into GraduatedStudents values('Zakir',50),('Ferid',15),('Leyla',28),('Vaqif',45)

select \* from Students inner join GraduatedStudents on Students.Id=GraduatedStudents.Id and GraduatedStudents.Age>30



Sql starts checking like this-> first it starts checkhing the first record of GraduatedStudents id if it is equal to the first Students id and GraduatedStudents Age is greater than 30 then it goes to the second record and checks the same GraduatedStudents id with the Students id and the age if it returns true for all then it selects the same GraduatedStudents again

create table Students(

Id int primary key identity,

Name varchar(16),

Grade int

)

create table Marks(

Id int primary key identity,

Letter varchar(1),

MinGrade int,

MaxGrade int,

)

insert into Marks values('A',91,100),('B',81,90),('C',71,80),('D',61,70)

insert into Students values('Zakir',70),('Ferid',95),('Leyla',78),('Vaqif',85),('Vahid',100)

select \* from Students join Marks on Students.Grade between Marks.MinGrade and Marks.MaxGrade



Cross join-It takes all the records from the table on the left and pairs every single record of it with the table on the right. It is useful when we want to compare all the records on the left table with every single row on the right table.

create table A(

Id int primary key identity,

Name varchar(16)

)

create table B(

Id int primary key identity,

Name varchar(16)

)

insert into A values('A1')

insert into B values('B1'),('B2'),('B3')

select A.Name A ,B.Name B from A cross join B



**Multi Joins**

create table Students(

Id int primary key identity,

Name varchar(16),

Grade int

)

create table Groups(

Id int primary key identity,

Name varchar(16),

)

create table GroupStudents(

Id int primary key identity,

Group\_id int foreign key references [Groups](Id),

Student\_id int foreign key references Students(Id)

)

insert into Students values('Zakir',70),('Ferid',95),('Leyla',78),('Vaqif',85),('Vahid',100)

insert into Groups values('P319'),('P320')

insert into GroupStudents values (1,1), (1,2), (1,3), (2,4), (2,5)

select s.name,g.Name as Groups from Students s join GroupStudents on s.Id=GroupStudents.Student\_id join Groups g on g.Id=GroupStudents.Group\_id



**Deleting a field (column)**

In order to delete an entire column, we use “drop” keyword.

alter table people add [name] nvarchar(20)

alter table people drop column [name]

**Group by (clause)**

For instance, we have got people from the same country and now we can use group by clause to select all the people from the same country.

If there is a cloumn and aggregate function up in our select statement, then we can’t select like this->

select name, count(\*) from Students

Column 'Students.name' is invalid in the select list because it is not contained in either an aggregate function or the GROUP BY clause.

So here we gotta use group by clause.

select country, count(\*) from Students group by country

select country, name,count(\*) from Students group by country,name

Here it first groups them by country and when there are two identical country names it groups them by their names

America Jeck 1

Cacusus Lala 1

Azerbaijan Nihad 1

Azerbaijan Vahid 1

After group by clause, if we wanna filter it then we have to use having clause.

select country,name,count(\*) from Students group by country,name having country='Azerbaijan'

We can use where clause only before group by clause!

select country,name,count(\*) from Students where country='Azerbaijan' group by country,name

select CustomerName, SUM(Amount) as Expenditure from Orders group by CustomerName order by CustomerName

**Copying values from another table**

insert into GraduatedStudents select name,surname,phoneNumber,score,comment from Students where isGraduated=1

**Vertical Joins:** **Union, Union All, Intersect, Except**

1. **UNION**: Combine two or more result sets into a single set, without duplicates.
2. **UNION ALL**: Combine two or more result sets into a single set, including all duplicates.
3. **INTERSECT**: Takes the data that is in both tables
4. **EXCEPT**: Takes the data that is in the first table but not in the second

select \* from UnGraduatedStudents

select \* from GraduatedStudents



select \* from UnGraduatedStudents

union

select \* from GraduatedStudents



select \* from UnGraduatedStudents

union all

select \* from GraduatedStudents



select \* from UnGraduatedStudents

intersect

select \* from GraduatedStudents



select \* from UnGraduatedStudents

Except

select \* from GraduatedStudents



**Create Index**

The CREATE INDEX statement is used to create indexes in tables for columns. Indexes are used to retrieve data from the database more quickly than otherwise. The users cannot see the indexes, they are just used to speed up searches/queries.

The search is implemented by binary search method.

Updating a table with indexes takes more time than updating a table without (because the indexes also need an update). So, only create indexes on columns that will be frequently searched against.

CREATE INDEX index\_name  
ON table\_name (column1, column2, ...);

DROP INDEX index\_name ON table\_name;

create table Students(

Id int primary key identity,

Name varchar(16)

)

create index idx\_Student\_Names on Students(Name)

select Name from Students

**Clustered vs Non-clustered indexes**

You would create indexes on columns that appear in where or join clause (Emails, Ids, etc.).

A table can have only one clusered index. A clustered index determines the physical order of data in a table. For this reason, a table can have only one clustered index. Note that Id column is marked as primary key. Primary key, constraint create clustered indexes automatically if no clustered index already exists on the table.

Clustered index can contain multiple columns (a composite index), like the way a telephone directory is organized by last name and first name.

Create a composite clustered Index on the Gender and Salary columns

Create Clustered Index IX tblEmployee\_Gender\_Salary

ON Employees (Gender DESC

Salary ASC)

The non-clustered index does not sort the data rows physically. It creates a separate key-value structure from the table data where the key contains the column values (on which a non-clustered index is declared) and each value contains a pointer to the data row that contains the actual value. It is similar to a textbook having an index at the back of the book with page numbers pointing to the actual information.

CREATE NONCLUSTERED INDEX NCI\_Employee\_Email

ON dbo.Employee(Email);

Example of indexes with good selectivity are:

* Unique indexes on NOT NULL columns have the highest selectivity, as there is only one row for each index entry.

Examples of indexes with poor selectivity are:

* Index on the column Gender, which stores just two unique values (M/F)
* Indexes on columns storing Boolean data; e.g., Y/N, 1/0, T/F

Another way of understanding selectivity is as a measure of the uniqueness of the data in the indexed column(s).

* Higher selectivity means
  + More unique data
  + Fewer duplicates
  + Fewer number of rows for each key value (result after search with specific criteria)
* Lower selectivity means
  + Less unique data
  + More duplicates
  + Larger number of rows for each key value

In short, **a highly selective index has few rows for each index entry and an unselective index has many rows for each index entry.**

If the index being used is unselective, it will return a large number of rows for each Key value. This could cause:

* A higher number of index blocks to be visited
* A higher number of table block visits, which could further be escalated by a high clustering factor of the index

Both the above factors could even lead the optimizer to choose Full table scan over index access because of lower cost of FTS. This means that we will have to bear the overhead of slower DML operations and maintenance of an unselective index even though it may not be utilized at all. Consider for example, an index having two distinct values in the key column(s). Assuming uniform data distribution, 50% ( = (1/2)\*100) of the rows in a table have the same value for the indexed key. In such a case, it is more efficient to perform a full table scan than accessing the table via index.

If the index is **not highly unselective**, the optimizer may choose to use the index but in that case,

* A large number of index and data blocks would initially be read into the buffer cache, resulting in waits for db file sequential read, Free buffers and Latch: Cache Buffer LRU Chain.
* The buffer cache might fall short of space due to the application’s unnecessarily requiring many more blocks, resulting in excessive linking / unlinking of buffers in the Cache Buffer chains, causing waits for Latch: Cache Buffer Chain wait event.
* The query would read too many data blocks into the buffer cache, keeping in wait other sessions that want to access one or more of those same blocks, thereby resulting in Buffer Busy waits.

**Selectivity of an index determines its effectiveness in optimizing performance.**The higher the selectivity, the fewer rows are returned by the index scan and the faster the query engine can reduce the size of the result set. It is desirable to have indexes with fairly high selectivity to avoid performance issues.

It is worth mentioning here that

* Oracle implicitly creates B-tree indexes on the columns having unique and primary key integrity constraints.
* A B-tree index does not store NULL entries.
* As a general guideline, B-tree indexes should be created on the columns that are often queried for less than 15% of the table's rows.

The optimizer prefers to

* Perform a Full Table Scan rather than use a highly unselective index.
* Access a table via index if the index is selective.

**Procedures**

There are 2 types of stored procedures: system procedures and user defined procedures.

create procedure GetStudentsInfo

As

select s.name,g.Name as Groups from Students s join GroupStudents gs on s.Id=gs.Student\_id join Groups g on g.Id=gs.Group\_id

create procedure GetStudentsById @Id int

As

begin

select \* from Students where Id=@Id

end

exec GetStudentsById 2

or we can execute it by not typing the exec keyword GetStudentsById 2

drop procedure GetStudentsById

If we want to pass a table name then it is going to be a string and note that we can’t use strings in select statements so we gotta use the entire select statement as a string then we need to parse it by executing sp\_executesql procedure.

Remember that sys.sp\_executesql only takes nvarchar!

create Procedure GetStudents @Table varchar(100),@GivenAge int

As

begin

declare @sqlquery nvarchar(200)

set @sqlquery='select \* from '+@Table+' where Age>'+str(@GivenAge)+' or Age<'+str(@GivenAge)

exec sys.sp\_executesql @sqlquery

end

exec GetStudents 'People',10

**Views**

In SQL, a view is a virtual table. If we have got an enormous select statement, then we use vews.

create View v\_GetStudentsInfo

As

select s.name,g.Name as Groups from Students s join GroupStudents gs on s.Id=gs.Student\_id join Groups g on g.Id=gs.Group\_id

select \* from v\_GetStudentsInfo

If we don’t need to pass an argument or anything like this, then we can just use views instead of procedures.

You can add SQL statements and functions to a view and present the data as if the data were coming from one single table.

drop view v\_GetStudentsInfo

The naming convention for view is as follows 🡪 v\_viewName

As keyword is used in sql to show the body of the view.

When using views, the same query is executed and the virtual table is created for us then we prefix it with select statement as above.

One of the advantages is the security as we can just share the virtual table with the client as opposed to the entire structure of the database as well as the query. The difference between a normal view and materialized view is that materialized view stores the result as well as the query however it doesn’t execute the query when we call it. It actually uses the result. One of the disadvantages of using materialized view is that it doesn’t get updated automatically so we need to manually refresh it.

**Functions**

There are 2 types of user defined functions: Table valued functions and scalar valued functions. Scalar valued functions return either int, varchar(n), bit. Table valued functions return a table.

The main differences between a function and procedure are that in functions you always have to return a value however in procedures you can both return or not return a value. Also you can’t execute a procedure in a function however you can call a function in a procedure. In functions, you can only use select statement but in procedures you can use insert, update and etc. One advantage of using functions is that you can call a function in a select statement but you cannot use procedures in a select statement.

create Function GetPerson(@Name varchar(10))

returns int

as

begin

declare @Count int

select @count=Sum(Age) from People where Name=@Name

return @count

end

select dbo.GetPerson('Vahid')

drop function GetPerson

**Triggers**

A SQL trigger is a database object which fires when an event occurs in a database.

create trigger ShowAfterInsert

on Products

after insert

as

begin

select \* from Products

end

insert into Products values('Solfetka'), ('Rucka')

drop trigger ShowAfterInsert

DML trigger statements use two special tables: the deleted table and the inserted tables. SQL Server automatically creates and manages these tables. You can use these temporary, memory-resident tables to test the effects of certain data modifications and to set conditions for DML trigger actions.

create trigger AfterDelete on Products

after delete

as

begin

insert into DelProducts(Name) select Name from deleted where Id in (select Id from deleted)

end

Now when we delete something it is first gonna insert the name of the deleted item in the DelProducts table and then it will delete it.

We can also use “instead of delete”

create trigger InsertInsteadOfDelete on Products

instead of delete

as

begin

insert into DelProducts(Name) select Name from deleted where Id in (select Id from deleted)

end

**Postgres**

Some differences between sql and posteSQL:

Sql is a relational database management system where postgres is an object realational database management system meaning that it has features such as table inhreritance and function overloading.

SqL is licences by microsoft and is not open source however postgres is open source.

In Postgress shell, some common commands:

\l to list all the databases

\c or \connect to connect to a database

\dt to list all the tables

\d “Stories” to list all the info (column and etc) of a table

Select \* from “Stories”;

When we are in the postgres container we need to get into the postgres database🡺 psql –U user-name

We must use semicolon at the end of our query to tell the shell that it is the end of our query and also double quotes are required if we include capital letters in our table name in postgres.

**Run psql**

When we run postgres we need to specify the password🡺

docker run –name mypsql –e POSTGRES\_PASSWORD=password postgres

**Restore psql database**

First we need to be in D:\PostgreSQL\14\bin folder to use this command 🡺 pg\_dump -U user -d db\_name -h 127.0.0.1 > dump.sql

dump.sql is created by the psql shell. We can also exclude host part.

Then we can seed our new database from this dump.sql file 🡺

cat dump.sql | docker exec -i <container-name> psql -U <user> -d <database> Here everything that was in the dump.sql file was copied to the database.

**Normalization**

Normalization is the process of organizing the data in the database. Normalization is used to minimize the redundancy from a relation or set of relations. It is also used to eliminate undesirable characteristics like Insertion, Update, and Deletion Anomalies. Normalization divides the larger table into smaller and links them using relationships.

The **delete anomaly** refers to the situation where the deletion of data results in the unintended loss of some other important data.

The **update anomaly** is when an update of a single data value requires multiple rows of data to be updated.

**Insertion Anomaly** refers to when one cannot insert a new tuple into a relationship due to lack of data.



**1NF**: A database is in first normal form if it satisfies the following conditions:

* Contains only atomic values
* There are no repeating groups (column names are unique)
* No mixing data types within one column

An atomic value is a value that cannot be divided. For example, in the table shown below, the values in the [Color] column in the first row can be divided into "red" and "green", hence [TABLE\_PRODUCT] is not in 1NF.



It states that an attribute of a table cannot hold multiple values. It must hold only single-valued attribute. First normal form disallows the multi-valued attribute, composite attribute, and their combinations.

**2NF**: a table shouldn’t have a partial dependency and all attributes must depend on a single primary key (or on an entire primary key if it PK consists of a set of columns). A table is in 2NF, only if a relation is in 1NF and meets all the rules, and every non-key attribute is fully dependent on an entire primary key. For example, in the below table teacher is dependant only on the subject\_id not on the entire primary key (which is a set of columns: score\_id, student\_id, subject\_id) of the table which is partial dependency as opposed to student\_id and subject\_id, making a primary key altogether. Also subject\_id makes up one part of the primary key and only in this case it is partial dependency. If it didn’t make up any part of the primary key, then it would be transitive dependency.



We need to remove teacher name from this table and move it to subject table where subject defines teacher.

**3NF**: A table shouldn’t have a transitive dependency meaning that all fields (columns) must be determined by only the primary key and no other column. A table is in 3NF, only if a relation is in 2NF and meets all the rules, and it has no Transitive Functional Dependency.



So, always everything must depend on only the candidate key.

**4NF**: no multi-valued dependencies.



Here every customer can have many newsletters, so here we have 2 records for each product, however we should have a separate table for this to reduce redundancy, thus complying with the forth form of normalization.



**Partial vs Transitive dependency**

If a non-prime attribute depends on the prime attribute this is functional dependency.

If a non-key attribute depends on an attribute that is a part of the primary key (aka candidate key) then it is a partial dependency as in the above example with subject🡪teacher relation.

But if a non-key attribute (aka non-prime attribute) depends on another non-key attribute (non-prime) that doesn’t make any part of the primary key then it is a transitive dependency as in the above example with exam\_name🡪total\_marks relation.

**Candidate key** is a single key or a group of multiple keys that uniquely identify rows in a table.

**ACID**

ACID (atomicity, consistency, isolation, durability) is a set of properties of database transactions intended to guarantee data validity despite errors, power failures, and other mishaps. In the context of databases, a sequence of database operations that satisfy the ACID properties (which can be perceived as a single logical operation on the data) is called a **transaction**. For example, a transfer of funds from one bank account to another, even involving multiple changes such as debiting one account and crediting another, is a single transaction.

Transactions are often composed of multiple statements. **Atomicity** guarantees that each transaction is treated as a single "unit", which either succeeds completely or fails completely: if any of the statements constituting a transaction fails to complete, the entire transaction fails and the database is left unchanged. A guarantee of atomicity prevents updates to the database occurring only partially, which can cause greater problems than rejecting the whole series outright. As a consequence, the transaction cannot be observed to be in progress by another database client.

**Consistency**: Any data written to the database must be valid according to all defined rules, including constraints, cascades, triggers, and any combination thereof.

**Isolation**: Transactions are often executed concurrently (e.g., multiple transactions reading and writing to a table at the same time). Isolation ensures that concurrent execution of transactions leaves the database in the same state that would have been obtained if the transactions were executed sequentially.

**Durability**: Durability guarantees that once a transaction has been committed, it will remain committed even in the case of a system failure (e.g., power outage or crash). This usually means that completed transactions (or their effects) are recorded in non-volatile memory.

**Char, nchar, varchar, nvarchar**

* **nchar** and **nvarchar** can store **Unicode** characters.
* **char** and **varchar** **cannot store Unicode** characters.
* **char** and **nchar** are **fixed-length** which will **reserve storage space** for number of characters you specify even if you don't use up all that space.
* **varchar** and **nvarchar** are **variable-length** which will only use up spaces for the characters you store. It **will not reserve storage like char or nchar**.

nchar and nvarchar will take up twice as much storage space, so it may be wise to use them only if you need *Unicode* support.

**Transaction Types**

The SQL Server Transactions are classified into three types, they are as follows

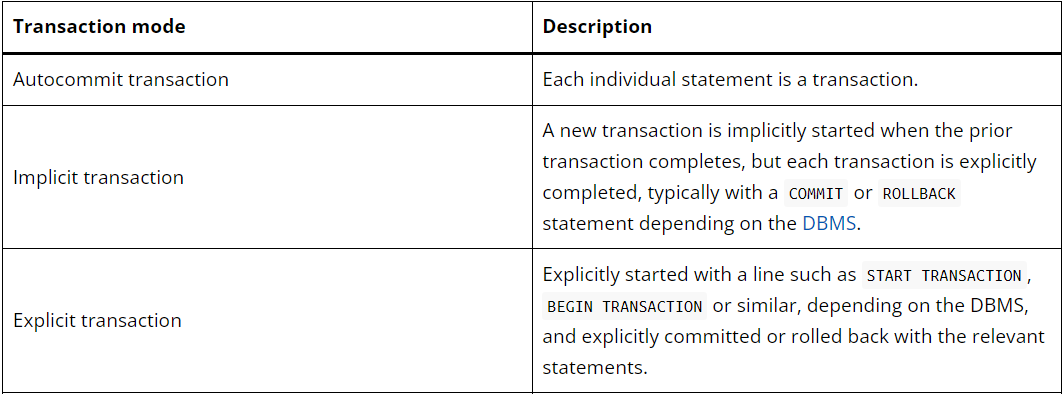
1. **Auto Commit Transaction Mode (default)**
2. **Implicit Transaction Mode**
3. **Explicit Transaction Mode**

Auto Commit Transaction Mode in SQL Server: This is the default transaction mode in SQL Server. In this transaction mode, each SQL statement is treated as a separate transaction. In this Transaction Mode, as a developer, we are not responsible for either beginning the transaction (i.e. Begin Transaction) or ending a transaction (i.e. either Commit or Roll Back). Whenever we execute any DML statement, the SQL Server will automatically begin the transaction as well as end the transaction with a Commit or Rollback i.e. if the transaction is completed successfully, it is committed. If the transaction faces any error, it is rolled back. So the programmer does not have any control over them.

Implicit Transaction Mode in SQL Server: In the Implicit mode of transaction, the SQL Server is responsible for beginning the transaction implicitly before the execution of any DML statement and the developers are responsible to end the transaction with a commit or rollback. Once the transaction is ended ie. once the developer executes either the commit or rollback command, then automatically a new transaction will start by SQL Server. That means, in the case of implicit mode, a new transaction will start automatically by SQL Server after the current transaction is committed or rolled back by the programmer. In order to use implicit transaction mode in SQL Server, first, we need to set the implicit transaction mode to ON using the SET IMPLICIT\_TRANSACTIONS statement.

In SQL Server, an implicit transaction is when a new transaction is implicitly started when the prior transaction completes, but each transaction is explicitly completed with a COMMIT or ROLLBACK statement. This is not to be confused with autocommit mode, where the transaction is started and ended implicitly.

Explicit Transaction Mode in SQL Server: In the Explicit mode of transaction, the developer is only responsible for beginning the transaction as well as ending the transaction. In other words, we can say that the transactions that have a START and END explicitly written by the programmer are called explicit transactions.



**Transaction Isolation levels**

Isolation property ensures that multiple transactions can occur concurrently without leading to the inconsistency of the database state. Transactions occur independently without interference. Changes occurring in a particular transaction will not be visible to any other transaction until that particular change in that transaction is written to memory or has been committed.

Transaction isolation levels are defined to address the following phenomena:

* **Dirty Read –**A Dirty read is a situation when a transaction reads data that has not yet been committed. For example, Let’s say transaction 1 updates a row and leaves it uncommitted, meanwhile, Transaction 2 reads the updated row. If transaction 1 rolls back the change, transaction 2 will have read data that is considered never to have existed.
* **Non-Repeatable read –**Non-Repeatable read occurs when a transaction reads the same row twice and gets a different value each time. For example, suppose transaction T1 reads data. Due to concurrency, another transaction T2 updates the same data and commit, now if transaction T1 rereads the same data, it will retrieve a different value.
* **Phantom Read –**Phantom Read occurs when two same queries are executed, but the rows retrieved by the two, are different. For example, suppose transaction T1 retrieves a set of rows that satisfy some search criteria. Now, Transaction T2 generates some new rows that match the search criteria for transaction T1. If transaction T1 re-executes the statement that reads the rows, it gets a different set of rows this time.

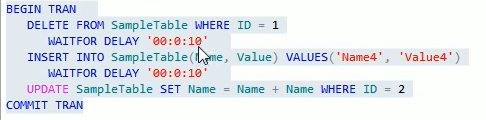
Based on these phenomena, The SQL standard defines four isolation levels : (these are aka pessimistic locking)

1. **Read Uncommitted –**Read Uncommitted is the lowest isolation level. In this level, one transaction may read not yet committed changes made by other transactions, thereby allowing dirty reads. At this level, transactions are not isolated from each other.
2. **Read Committed –**This isolation level guarantees that any data read is committed at the moment it is read. Thus it does not allow dirty read. The transaction holds a read or write lock on the current row, and thus prevents other transactions from reading, updating, or deleting it.
3. **Repeatable Read –**This is the most restrictive isolation level. The transaction holds read locks on all rows it references and writes locks on referenced rows for update and delete actions. Since other transactions cannot read, update or delete these rows, consequently it avoids non-repeatable read.

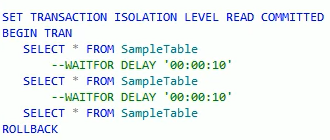
Existing rows are locked from being changed. New rows can be inserted later (phantom row). But, once a new row has been inserted then it also will be protected. So, it only protects existing rows from being updated.

1. **Serializable –**This is the highest isolation level. A *serializable* execution is guaranteed to be serializable. Serializable execution is defined to be an execution of operations in which concurrently executing transactions appears to be serially executing.

This is even more restrict and doesn’t even allow phantom rows. So, it guarantees repeated reads and disallows phantom rows



We are beginning a transaction and waiting for some time to emulate an operation.



So the latter transaction will need to wait for first transaction to be committed so that it can read because it only reads committed data.

The default transaction isolation level is READ COMMITTED.

We can use the (NOLOCK) table hint to read uncommitted transaction from read committed transaction level It is the same as Read uncommitted🡺



When we use read committed transaction isolation level, no operation will be done until the other transaction ends.

**Note that** transactions will only lock the rows that they are acquiring so if it is select \* then the transaction will lock everything. If it is where id=1 then only the first row will be locked.

An UPDATE without a WHERE clause will lock all rows in the table. The rows can not be deleted or updated from a different transaction because they are locked. But you can insert new rows without problems. So transactions lock only rows that they need in the transaction.

**Optimistic vs Pessimistic locking**

**Optimistic** Locking is a strategy where you read a record, take note of a version number (other methods to do this involve dates, timestamps or checksums/hashes) and check that the version hasn't changed before you write the record back. When you write the record back you filter the update on the version to make sure it's atomic. (i.e. hasn't been updated between when you check the version and write the record to the disk) and update the version in one hit. If the record is dirty (i.e. different version to yours) you abort the transaction and the user can re-start it. Optimistic locking is used when you don't expect many collisions. It costs less to do a normal operation but if the collision DOES occur you would pay a higher price to resolve it as the transaction is aborted.

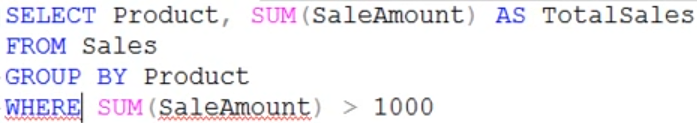
**Pessimistic** Locking is when you lock the record for your exclusive use until you have finished with it. It has much better integrity than optimistic locking but requires you to be careful with your application design to avoid Deadlocks.

**Where vs Having**

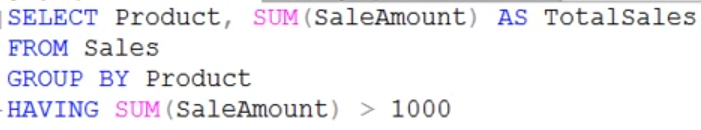
Where clause is used to filter individual rows whereas having clause is used to filter groups. Also, where comes before group by and having comes after group by. This means the following 🡺

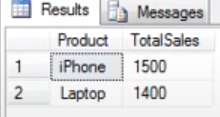


In where clause, we cannot use aggregate functions however in having clause we can use them.

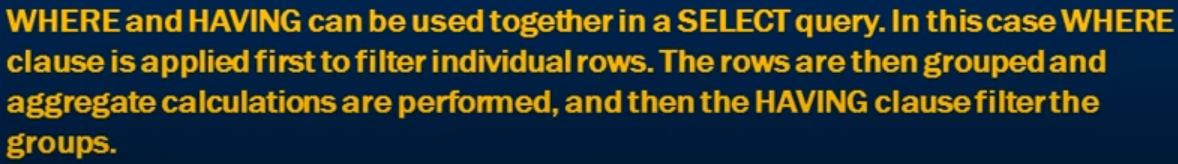


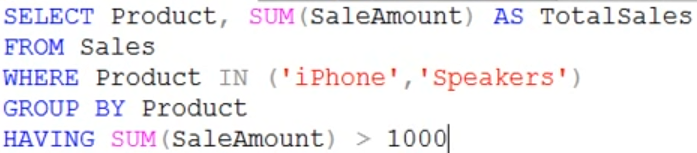
-----------------------------------------------------------------------------------------------





So here there is a perfomance issue with having.





**CAP Theorem**

Consistency, availability and partition tolerance in distributed system. You can never have all three these you can only have 2 of them.

Consistency means that you are not reading dirty data (likely to be rollbacked or etc.). So if you want consistency then you need to sacrifice availability. We block all the tables till all tables have the same data (committed).

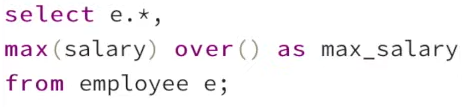
If we choose availability then we show the data despite the fact the it can be a dirty data.

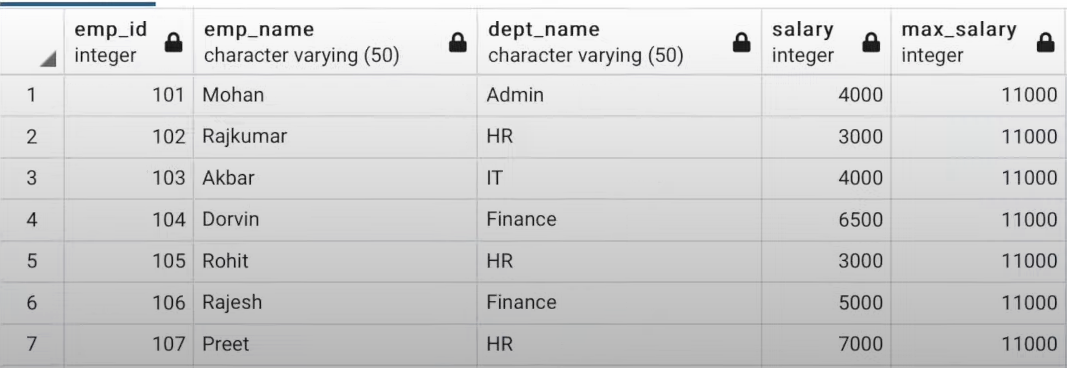
High consistency reduces availability and vice versa.

Partition tolerance as the name implies states that even if one of your systems get partitioned from the system (breaks) then it can continue without single point of failure.

**Analytic functions (aka Window function)**

Analytic functions calculate an aggregate value based on a group of rows. Unlike aggregate functions, however, analytic functions can return multiple rows for each group. Window functions compute values over a group of rows and return a single result for each row. This is different from an aggregate function, which returns a single result for a group of rows. For example, 🡺

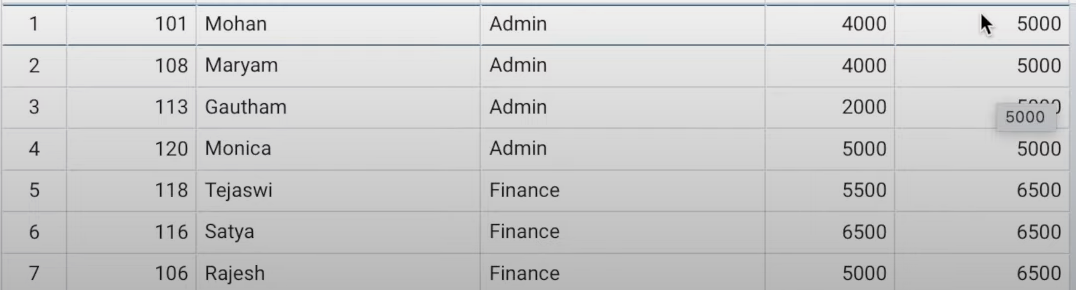




We use “over” clause to tell sequel to treat the max function in an analytical way.

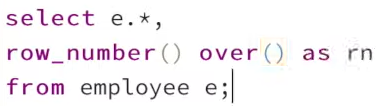
In the above example we don’t partition it by anything so there isn’t any boundary here thus there will only be one window. If there were a boundary then the aggregate calculation would be performed for each boundary and It would always reset itself when crossing one boundary. As a result, we would get multiple windows (sets).





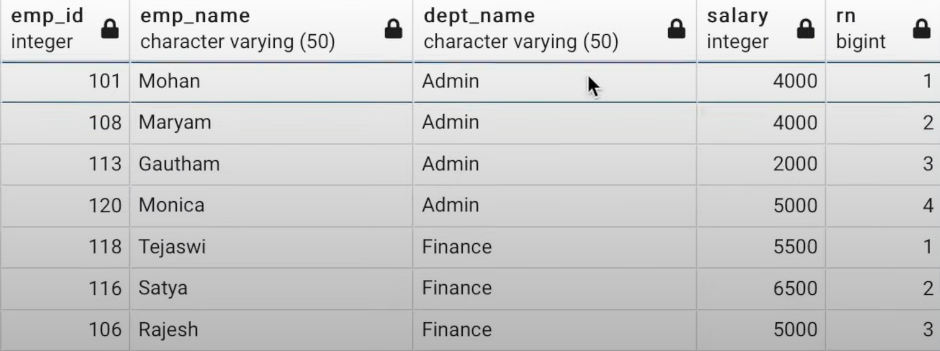
As you can see it returns all the rows for each department with max value in the department. It recalculates the max value when it reaches its boundary.

Row\_Number()

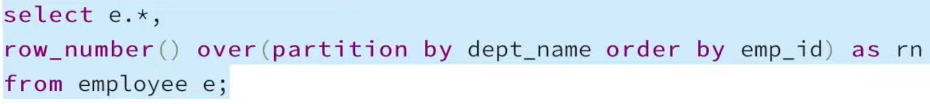


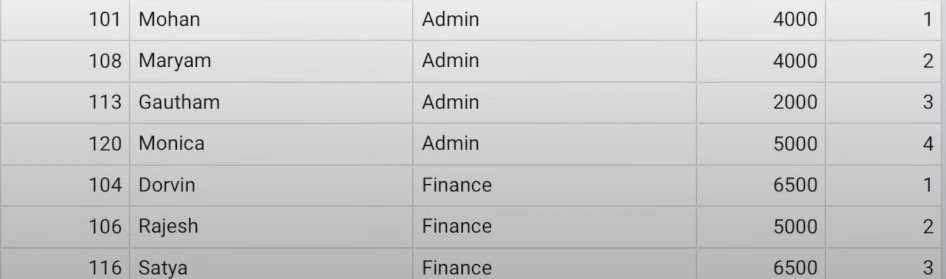
This will result in rows with their corresponding row number. And If we have some kind of groups (partitions by column) then row number will get reset once it reaches the end of the boundary of the group.





It recalculates the row number after the boundary of the group is reached. We can also order by some column 🡺





As you can see Row\_number also works in accordance with the order.