Oracle-- <https://apex.oracle.com/en/>

Khalid0142759 Shahela@24oct

# RDBMS Cache

An RDBMS cache refers to a temporary storage area within a relational database management system (RDBMS) where frequently accessed data is stored in memory **(RAM**)  instead of slower disk storage, allowing for faster retrieval compared to accessing the primary database directly, significantly improving application performance by reducing the load on the database server; essentially, it's a "buffer" for frequently queried data that can be accessed much quicker than reading directly from the database storage.

This caching technique  can be applied to any type of database. However, this paper focuses on relational databases because they are the most common database caching use case.

## ****Benefits****

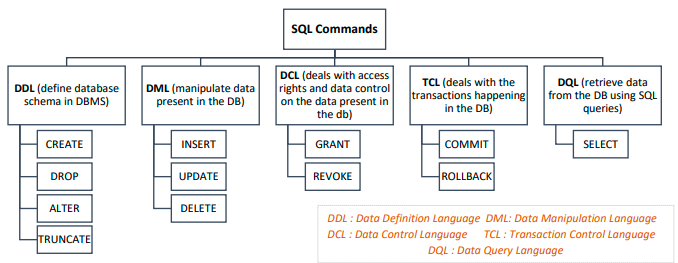
1. Reduced database load
2. Faster response times for common queries
3. Enhanced application scalability

## ****Data retrieval process****

When the same query is executed again, the database checks the cache first and returns the data directly from there if found, bypassing the need to access the primary database.

SELECT @@SERVERNAME,@@VERSION,@@SERVICENAME

# DDL,DML



# Data type

|  |  |  |
| --- | --- | --- |
| **Data type** | **Format** | **Range** |
| Time(SQL 2008) | hh:mm:ss[.nnnnnnn] | 00:00:00.0000000 through 23:59:59.9999999 |
| date(SQL 2008) | YYYY-MM-DD | 0001-01-01 through 9999-12-31 |
| smalldatetime | YYYY-MM-DD hh:mm:ss | 1900-01-01 through 2079-06-06 |
| datetime | YYYY-MM-DD hh:mm:ss[.nnn] | 1753-01-01 through 9999-12-31 |
| datetime2 | YYYY-MM-DD hh:mm:ss[.nnnnnnn] | 0001-01-01 00:00:00.0000000 through 9999-12-31 23:59:59.9999999 |
| tinyint | **1 byte**--2 power(8) | 0-255 |
| smallint | **2 byte**--2 power(16) | (2^16)/2 ( -32,768 to +32,767) |
| int/integer | **4 byte**--2 power(32) | (2^32)/2 (-2,147,483,648 to +2,147,483,647) |
| bigint | **8 byte**--2 power(64) | (2^64)/2 (-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807) |
| decimal |  |  |

# Difference between nvarchar and varchar

* **nchar** and **nvarchar** can store **Unicode** characters.
* **Char and varchar can store ASCII character**
* Regarding memory usage, nvarchar uses 2 bytes per character, whereas varchar uses 1.

# INFORMATION\_SCHEMA

## ****Table List****

SELECT \* FROM INFORMATION\_SCHEMA.TABLES

## ****View list****

**SELECT** table\_name **FROM** information\_schema.views **WHERE** table\_schema = 'database\_name'

## Constraints List

SELECT \* FROM INFORMATION\_SCHEMA.TABLE\_CONSTRAINTS WHERE CONSTRAINT\_CATALOG='DbName' and TABLE\_NAME='TABLE\_NAME';

## Column list

select COLUMN\_NAME, DATA\_TYPE, CHARACTER\_MAXIMUM\_LENGTH,NUMERIC\_PRECISION, DATETIME\_PRECISION, IS\_NULLABLE from INFORMATION\_SCHEMA.COLUMNS where TABLE\_NAME='Test'

## Master Query to list—View,PROC,Function,Trigger

SELECT O.name, M.definition, O.type\_desc, O.type FROM **sys.sql\_modules** M

INNER JOIN **sys.objects** O ON M.object\_id=O.object\_id

WHERE O.type IN ('IF','TF','FN')

# Temp Table

These tables can be created at **runtime** and can do the all kinds of operations that one normal table can do. But, based on the table types, the scope is limited. These tables are created inside tempdb database.

### Local Temp Table

Local temp tables are only available to the current connection for the user; and they are automatically deleted when the user disconnects from instances. Local temporary table name is stared with hash ("#") sign.

### Global Temp Table

Global Temporary tables name starts with a double hash ("##"). Once this table has been created by a connection, like a permanent table it is then available to any user by any connection. It can only be deleted once all connections have been closed.

### Table variable

The table variable is a special type of the local variable that helps to store data temporarily, similar to the temp table in SQL Server. In fact, the table variable provides all the properties of the local variable, but the local variables have some limitations, unlike temp or regular tables.

DECLARE @tbl TABLE(id int,name varchar(20))

SELECT \* FROM @tbl

* At the same time, we can insert,update and delete the data
* table variables are stored in the tempdb database.
* The lifecycle of the table variables starts in the declaration point and ends at the end of the batch. As a result, the table variable in SQL Server is automatically dropped at the end of the batch:

DECLARE @tbl TABLE(id int,name varchar(20))

SELECT \* FROM @tbl

insert into @tbl values(1,'khalid1');

insert into @tbl values(2,'khalid2');

update @tbl set name='khalid11' where id=1

delete from @tbl where id=2

select \* FROM @tbl.

#### Table variable constraint

Table variables allow us to create the following constraints:

* Primary Key
* Unique
* Null
* Check

DECLARE @TestTable TABLE

(ID INT PRIMARY KEY,

Col1 VARCHAR(40) UNIQUE,

Col2 VARCHAR(40) NOT NULL,

Col3 int CHECK (Col3>=18))

INSERT INTO @TestTable

VALUES(1,'Value1',12 , 20)

SELECT \* FROM @TestTable

Limitation

* Foreign Key constraints cannot use for the table variables.
* we have to define the constraints when we are declaring the table variable otherwise, we experience an error.

# Declare variable

DECLARE

@variable\_name datatype [ = initial\_value ],

@variable\_name datatype [ = initial\_value ],

...;

SET @techonthenet = 'Example showing how to declare variable';

## Example

DECLARE @t1 datetime, @n int;

SET @t1=getdate();

SET @n=10;

WHILE @n>0 begin

SELECT @n As n,@t1 as time;

SET @n=@n-1;

SET @t1=@t1-1;

end

## Declare variable with table type

DECLARE @Pet Table(PetID INT, OwnerID INT , PetName NVARCHAR(50), AnimalType NVARCHAR(10))

INSERT INTO @Pet(PetID, OwnerID, PetName, AnimalType) VALUES(1, 1, 'Sonya', 'Dog')

INSERT INTO @Pet(PetID, OwnerID, PetName, AnimalType) VALUES(2, 2, 'Bessy', 'Cat')

## Difference between table-variable and Temporary Table

* The first one is that transaction logs are not recorded for the table-variables. Hence, they are out of scope of the transaction mechanism

CREATE table #T (s varchar(128))

DECLARE @T table (s varchar(128))

INSERT into #T select 'old value #'

INSERT into @T select 'old value @'

BEGIN transaction

UPDATE #T set s='new value #'

UPDATE @T set s='new value @'

ROLLBACK transaction

SELECT \* from #T

SELECT \* from @T

s

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

old value #

s

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

new value @

* table-variables exist only in the same scope as variables. Contrary to the temporary tables, they are visible for entire session

# Function

Functions in SQL Server are the database objects that contains a **set of SQL statements to perform a specific task**. A function accepts input parameters, perform actions, and then return the result. We should note that functions always return either a single value or a table.

* A function must have a name, and the name cannot begin with a special character such as @, $, #, or other similar characters
* Whenever a function is called, it compiles.
* Functions must return a value or result.
* Functions use only input parameters.
* We cannot use TRY and CATCH statements in functions.

## Type of function

### System Functions--- ****built-in functions****

* Aggregate Function—Count,Sum,AVG,MAX,MIN
* Numeric Function
* Conversion Function
* String Function
* Date and Time Function

### User-Defined Functions

* Scalar Functions-- accepts parameters, either single or multiple and returns a single value.
* Table-Valued Functions

**Function Syntax**

CREATE **FUNCTION** schema\_name.function\_name (parameter\_list)

**RETURNS** data\_type **AS**

**BEGIN**

    statements

**RETURN** value

**END**

## Scalar Function

### without parameter

CREATE FUNCTION fn1()

returns varchar(20) AS

BEGIN

RETURN 'This is Khalid Anwar'

END

We can call function using **SELECT** **schemaName**.**fucntionName**

e.g – SELECT dbo.fn1()

### with parameter

CREATE FUNCTION fn2(@name varchar(20))

RETURNS varchar(20) AS

BEGIN

RETURN 'Engineer--'+@name

END

We can call function using SELECT dbo.fn2('MD KHALID ANWAR')

## Inline ****Table-Values Functions****

returns data of a **table** type.

A single SELECT statement should be used to determine the value of the table variable.

This function does not allow declaration(inside body) and **begin**-**end** keyword

ALTER FUNCTION fn\_EMP(@id int)

RETURNS **TABLE**

AS

RETURN

SELECT \* FROM EMP where id=@id

### Calling Function

SELECT \* FROM fn\_EMP(1)

## Multi statement Table Valued Function

ALTER FUNCTION table\_valuedfn\_EMP()

RETURNS @table\_valued TABLE (ID INT,ename VARCHAR(20))

AS

BEGIN

INSERT INTO @table\_valued VALUES(1,'kk1')

INSERT INTO @table\_valued VALUES(2,'kk2')

RETURN;

END;

### Calling Function

SELECT \* FROM table\_valuedfn\_EMP()

# PROCEDURE

Stored procedures are precompiled SQL statements that are stored in the **database**and can be executed as a **single unit**.

A SQL Stored Procedure is a collection of SQL statements bundled together to perform a specific task.

**CREATE** **PROCEDURE** procedureName @parameter datatype [OUT |IN], @parameter datatype [OUT |IN]

AS

BEGIN

[declaration\_section]

executable\_section

END;

--Oracle has () which is different

## PROCDEURE with OUT parameter

### Example 1

CREATE PROCEDURE proc1 @Name VARCHAR(50) OUT

AS

BEGIN

DECLARE @age int;

SET @age = 8;

IF @age < 10

SET @Name = 'MD KHALID ANWAR';

ELSE

SET @Name = 'SHAHELA PARVEEN';

END

### Calling Procedure

DECLARE @name1 varchar(100);

EXEC proc1 @name1 OUT;

PRINT @name1;

## PROCDEURE without parameter

### Example 2

CREATE PROCEDURE MyProc

AS

SELECT \* FROM EMP

---EXEC MyProc

## PROCDEURE with IN parameter

Default is IN parameter

CREATE PROCEDURE proc3 @a INT,@b INT,@c INT OUT

AS

BEGIN

SET @c=@a + @b

END

### Calling Procedure

It is different from oracle.oracle uses () while sending parameter

DECLARE @c INT

EXEC proc3 11,12,@c OUT

PRINT @C

## Difference between Function and procedure

|  |  |  |
| --- | --- | --- |
| SN | **Function** | **Procedure** |
| 1 | Functions always return a value after the execution of queries. | The procedure can return a value using “IN OUT” and “OUT” arguments. |
| 2 | A Function can not return multiple result sets. | A procedure is able to return multiple result sets. Multiple select query |
| 3 | Function can be called inside proc | PROC can be called from function |
| 4 | A function used only to read data. | A procedure can be used to read and modify data. |
| 5 | The function does not support try-catch blocks. | Procedure supports try-catch blocks for error handling. |

## View Stored procedure using query

SELECT name, create\_date, modify\_date,\* FROM sys.procedures

SELECT name, create\_date,\* FROM sys.objects WHERE type = 'P'

# SQL VIEW

*Views* are virtual tables that allow you to represent data in an alternate way. You can create a view only in the current database. It is a type of virtual table that simplifies how users interact with data across one or more tables. Unlike traditional tables, a view in SQL does not store data on disk; instead, it dynamically retrieves data based on a pre-defined query each time it’s accessed.

* **Simplify Complex Queries:** Encapsulate complex joins and conditions into a single object.
* **Enhance Security:**Restrict access to specific columns or rows.
* **Present Data Flexibly:** Provide tailored data views for different users.

CREATE VIEW StorePersonnel

AS SELECT SalesPersonID, Name FROM AdventureWorks.Sales.Store WHERE SalesPersonID >250

List All View

**SELECT** table\_name **FROM** information\_schema.views **WHERE** table\_schema = 'database\_name'

## Drop View

**DROP VIEW** view\_name;

## ****Explain the concept of materialized views and their benefits.****

Materialized views are pre-computed copies of database queries stored as tables. They improve query performance by providing pre-aggregated or filtered data, reducing processing needs for repeated queries.

A **materialized view** stores the **result of a query** physically in the database. It can be refreshed manually or automatically to **reflect updates** in the underlying tables.

This is applicable in Azure datawarehouse and not applicable to SQL Server

**Syntax**

**CREATE MATERIALIZED VIEW** materialized\_view\_name

BUILD [IMMEDIATE | DEFERRED]

REFRESH [FAST | COMPLETE | FORCE]

ON [COMMIT | DEMAND]

AS

SELECT column1, column2, ...

FROM table\_name

WHERE condition;

# Pivot

SELECT first\_column AS <first\_column\_alias>,

[pivot\_value1], [pivot\_value2], ... [pivot\_value\_n]

FROM

(<source\_table>) AS <source\_table\_alias>

PIVOT

(

aggregate\_function(<aggregate\_column>)

FOR <pivot\_column>

IN ([pivot\_value1], [pivot\_value2], ... [pivot\_value\_n])

) AS <pivot\_table\_alias>;

### Different from oracle:

--IN parameter where [] is mandatory

-- aliase table and pivot name is mandatory

### Example

select \* from

(select \* from employees)a

PIVOT(sum(salary)

FOR dept\_id IN ([30],[45]))pv

***Example-2*** *pivot on multiple columns*

SELECT \* FROM (

SELECT Seq, ItemCode, ItemModel, ItemName, ItemColour FROM TblPivot ) AS P

-- For ItemColour

PIVOT ( Count(ItemCode) FOR ItemColour IN ([Red], [Blue], [Green]) ) AS pv1

-- For ItemName

PIVOT ( Count(ItemModel) FOR ItemName IN ([Samsung Mobile], [Nokia Mobile], [Motorola Mobile]) ) AS pv2

## MySql syntax

<https://www.geeksforgeeks.org/how-to-return-pivot-table-output-in-mysql/>

SELECT

non\_pivoted\_column,

**SUM**(

CASE WHEN pivoted\_column = 'value1' THEN aggregate\_column END

) AS value1,

SUM(

CASE WHEN pivoted\_column = 'value2' THEN aggregate\_column END

) AS value2,

-- Additional pivoted columns as needed

FROM your\_table

GROUP BYnon\_pivoted\_column;

**Example**: Below are table details

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **NAME** | **COLLEGE** | **ROLL NUMBER** | **SUBJECT** | **MARKS** |
| ROMY | BVP | 261150 | DBMS | 90 |
| ROMY | BVP | 261150 | NETWORKING | 87 |
| ROMY | BVP | 261150 | GRAPHICS | 95 |
| PUSHKAR | MSIT | 898888 | DBMS | 91 |
| PUSHKAR | MSIT | 898888 | NETWORKING | 90 |
| PUSHKAR | MSIT | 898888 | GRAPHICS | 78 |

Output should be as below.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **NAME** | **COLLEGE** | **ROLL NUMBER** | **DBMS** | **NETWORKING** | **GRAPHICS** |
| ROMY | BVP | 261150 | 90 | 87 | 95 |
| PUSHKAR | MSIT | 898888 | 91 | 90 | 78 |

SELECT NAME,COLLEGE,ROLL\_NO,

SUM(CASE WHEN SUBJECT='DBMS' THEN MARKS END) AS 'DBMS',

SUM(CASE WHEN SUBJECT='NETWORKING' THEN MARKS END) AS 'NETWORKING',

SUM(CASE WHEN SUBJECT='GRAPHICS' THEN MARKS END) AS 'GRAPHICS'

FROM pivot1

GROUP BY NAME,COLLEGE,ROLL\_NO

# Constraint

* NOT NULL
* CHECK
* UNIQUE
* PRIMARY KEY
* FOREIGN KEY
* DEFAULT

# CREATE Table

CREATE TABLE customers( customer\_id DECIMAL(10,2) NOT NULL,

customer\_name varchar(50) NOT NULL,city varchar(50));

## Create table with primary key

## create table primaryTbl(id int,name varchar(20), primary key (id)) --single key

create table primaryTbl1(id1 int,id2 int,name varchar(20), primary key (id1,id2)) ---composite key

## Create table with Foreign key

## create table foreignTable(id int,fid int,name varchar(20),

## primary key (id),foreign key (fid) REFERENCES primaryTbl(id))

## Create table with Unique key

**CREATE TABLE UniqueTable (ID int NOT NULL UNIQUE,Name varchar(255) NOT NULL)**

## Rename table

In oracle Alter table table\_name RENAME TO new\_table\_name

In SQL Serve**r SP\_RENAME ‘old\_table\_name’,’new\_table\_name’;**

## Rename Column

sp\_rename 'tableName.oldColumnName', 'NewColumnName',’**COLUMN**’

# DROP table

**DROP TABLE IF EXISTS** tableName

**DROP TABLE** tableName

# Alter Table

## Column

### ADD Column

## Alter Table tblName ADD col1 varchar(10),col2 varchar(20) --oracle has () for list of columns

### Modify dataType

Alter Table customers **ALTER** **COLUMN** customerName varchar2(100) --only single column

Alter Table customers **ALTER** **COLUMN** id int NOT NULL --changed not nullable column

### DROP COLUMN

Alter Table tableName **DROP COLUMN** column\_name;

## PRIMARY KEY

By default , adding primary key created unique clustered index

**ALTER TABLE** customers **ADD CONSTRAINT** pk1 **PRIMARY KEY** (idnt)

Not: Cannot define PRIMARY KEY constraint on nullable column in table

If you want to create nonclustered unique index on primary key then as follow

ALTER TABLE customers ADD CONSTRAINT pk1 PRIMARY KEY NONCLUSTERED (idnt)

### View constraint

**Sp\_help** tableName --One way to use stored proc

SELECT \* FROM INFORMATION\_SCHEMA.TABLE\_CONSTRAINTS WHERE CONSTRAINT\_CATALOG='DbName' and TABLE\_NAME='TABLE\_NAME'; --system view

### DROP Constraint

ALTER TABLE customers DROP CONSTRAINT pk1

### Disable Constraint

ALTER TABLE customers NOCHECK CONSTRAINT pk1

### Enable Constraint

ALTER TABLE customers CHECK CONSTRAINT pk1

# FOREIGN KEY CONSTRAINT

ALTER TABLE customers\_fk ADD CONSTRAINT fk1 FOREIGN KEY (FK\_idnt) REFERENCES customers(idnt)

### DROP Constraint

ALTER TABLE customers\_fk DROP CONSTRAINT fk1

### Disable Constraint

ALTER TABLE table\_name NOCHECK CONSTRAINT constraint\_name;

ALTER TABLE customers\_fk NOCHECK CONSTRAINT fk1

### Enable Constraint

ALTER TABLE table\_name CHECK CONSTRAINT Constraint\_name;

ALTER TABLE customers\_fk CHECK CONSTRAINT fk1

# UNIQUE CONSTRAINT

alter table customers

ADD CONSTRAINT uq UNIQUE (cid)

## Create copy from old table

### SELECT INTO

SELECT expressions INTO new\_table FROM tables [WHERE conditions];

expressions

The columns or calculations that you wish to retrieve.

SELECT employee\_id, last\_name **INTO contacts** FROM employees WHERE employee\_id < 1000;

SELECT \* INTO contacts FROM employees WHERE employee\_id < 1000;

**Example**

**Select** customerfirstname+' '+customerlastname as [Customer Name] , customerphonenumber, inventoryname,saledate,salequantity,saleunitprice,salequantity\*saleunitprice as [Total Amount]

into customerRec

**from** customer

**inner** **join** sale **on** customer.customerid=sale.customerid

**inner** **join** inventory on sale.inventoryid=inventory.inventoryid

**order** **by** customerfirstname +' '+ customerlastname,inventoryname

**In MySQL**

CREATE TABLE newTable AS SELECT \* FROM OldTable

# INSERT INTO SELECT

INSERT INTO table\_name1 SELECT \* from table\_name2 where col1=1;

INSERT INTO Customers (CustomerName, City, Country) SELECT SupplierName, City, Country FROM Suppliers;

### foreign key with Cascade DELETE

A foreign key with cascade delete means that if a record in the parent table is deleted, then the corresponding records in the child table will automatically be deleted. This is called a cascade delete in SQL Server. A foreign key with cascade delete can be created using either a CREATE TABLE statement or an ALTER TABLE statement.

ALTER TABLE customers\_fk ADD CONSTRAINT fk1 FOREIGN KEY (FK\_idnt) REFERENCES customers(idnt) ON DELETE CASCADE

# CHECK Constraint

CREATE TABLE table\_name

(

column1datatype null/not null,

column2datatype null/not null,

...

CONSTRAINT constraint\_name CHECK (column\_name condition) [DISABLE]

);

CREATE TABLE frgn(id1 int,name1 varchar(10),CONSTRAINT ckCHECK(id in (1,2,3))

CREATE TABLE frgn(id1 int,name1 varchar(10),CONSTRAINT ckCHECK(id >=10)

CREATE TABLE frgn(id1 int,name1 varchar(10),CONSTRAINT ckCHECK(id between 10 and 100)

Alter table table\_nameADD CONSTRAINTconstraint\_name CHECK (col\_name>=10)

ALTER TABLE [TableName] DROP CONSTRAINT [CONSTRAINT\_NAME]

# DEFAULT CONSTRAINT

A default is bound to a column or alias data type and specifies a default value for the column or columns, when no value is supplied.

CREATE TABLE table\_Name (

ID int NOT NULL,

LastName varchar(255) NOT NULL,

FirstName varchar(255),

Age int,

City varchar(255) DEFAULT 'Sandnes',

OrderDate datetime DEFAULT GETDATE());

ALTER TABLE table\_name ADD CONSTRAINT constraint\_name DEFAULT GETDATE() for column\_name

ALTER TABLE table\_name DROP CONSTRAINT Contraint\_name

# IDENTITY

* The identity must only contain integer values but the column can be of any numeric datatype (bigint, int, tinyint, numeric, decimal).
* An identity column is not guaranteed to be unique nor consecutive
* Index can be created on identity column
* Constraint can be added on identity column like primary key

## CREATE TABLE with IDENTITY

CREATE TABLE tbl\_Indentity(ID INT IDENTITY(1,1) NOT NULL,NAME1 VARCHAR(20))

### In Mysql

We should use AUTO\_INCREMENT keyword and column must be part of key

CREATE Table IdentityEx(id int auto\_increment, name varchar(20),PRIMARY KEY(id));

## ALTER TABLE with IDENTITY

ALTER TABLE table\_name ADD column\_name data\_type IDENTITY(starting value,Increment value)

ALTER TABLE customers ADD idnt INT IDENTITY(1,2)

## Find current identity value

IDENT\_CURRENT(‘table\_name’)—must include quotation

SELECT IDENT\_CURRENT('tbl\_Indentity')

## Changing Current Seed

DBCC checkident(table\_name,reseed,1)

--Table can have only one identity column

--We can also reset the current seed to it’s original value via a truncate table. Delete does not do.

# SEQUENCES

This feature is available from SQL server 2012 onwards.

A sequence is a user defined schema bound object that **generates a sequence of numeric values**.

Sequences are frequently used in many databases because many applications require each row in a table to contain a unique value and sequences provides an easy way to generate them.

The sequence of numeric values is generated in an ascending or descending order at defined intervals and can be configured to restart when max\_value exceeds.

**Syntax:**

CREATE SEQUENCE sequence\_name

START WITH initial\_value

INCREMENT BY increment\_value

MINVALUE minimum value

MAXVALUE maximum value

CYCLE|NOCYCLE ;

**sequence\_name:** Name of the sequence.

**initial\_value:** starting value from where the sequence starts.

Initial\_value should be greater than or equal

to minimum value and less than equal to maximum value.

**increment\_value:** Value by which sequence will increment itself.

Increment\_value can be positive or negative.

**minimum\_value:** Minimum value of the sequence.

**maximum\_value:** Maximum value of the sequence.

**cycle:** When sequence reaches its set\_limit

it starts from beginning.

## Difference between identity and Sequence

An **identity** is a sequence that's automatically created and associated with a table column. Identity properties are tied to a specific table and can't be shared with other tables.

**Sequences** are user-defined database objects that can generate a series of numeric values. Sequences are independent objects that can be used across multiple tables.

# TRIGGER

It is a specialized category of stored procedure that is called automatically when a database server event occurs. Each trigger is always associated with a **table**. A trigger is called a special procedure because it cannot be called directly like a stored procedure.

* We cannot manually execute/invoked triggers.
* Triggers have no chance of receiving parameters.
* A transaction cannot be committed or rolled back inside a trigger.

**Syntax**

CREATE [ OR ALTER ] TRIGGER [ schema\_name . ]trigger\_name

ON { table | view }

{ FOR | AFTER | INSTEAD OF }

{ [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }

AS { sql\_statement}

## DML Trigger

Data Manipulation Language (DML) triggers execute when a user tries to modify data. DML triggers are carried out on DML events such as **INSERT**,**UPDATE**, or **DELETE** statements

There are two types of DML trigger

1. After Triggers-- SQL Server cannot fire the AFTER trigger when the data insertion failed.

-- this trigger is executed **after** completes an insert, update or delete operations.

-- AFTER triggers cannot be defined on **views**. It is applicable to table

* AFTER INSERT
* AFTER DELETE
* AFTER UPDATE

1. Instead Of Triggers-- SQL Server also fires the Instead of Trigger if the data insertion fails.

-- this trigger is only executed **instead of** insert, update or delete operations

-- it is opposite of After trigger.

-- This is applicable to View

You can't specify INSTEAD OF for DDL or logon triggers.

* INSTEAD OF INSERT
* INSTEAD OF DELETE
* INSTEAD OF UPDATE

**CREATE** **TRIGGER** triggerName on tableName AFTER INSERT|UPDATE|DELETE

AS

DROP TRIGGER Trigger\_name

## Inserted

It is special table .The inserted table keeps the copy of the row when you insert a new row into the actual table.

It is used UPDATE/INSERT trigger

CREATE TRIGGER trg1 ON myTable FOR INSERT

AS

BEGIN

Declare @Id int,@name varchar(20)

SELECT @Id = id,@name=name from inserted

INSERT INTO trgTable VALUES (@id,@name)

END

## Deleted

It is special table.deleted table keeps the copy of the row you have just deleted from the actual table.

It is used for DELETE trigger

CREATE TRIGGER trg2 ON myTable FOR DELETE

AS

BEGIN

Declare @Id int,@name varchar(20)

SELECT @Id = id,@name=name from deleted

INSERT INTO trgTable VALUES (@id,@name)

END

There are two types of triggers

## DDL Trigger(ON Database)

It includes creating and dropping database objects such as tables, views, procedures, and logins. DDL triggers can be associated with CREATE, ALTER, and DROP statements.

It is viewed by sys.triggers.

CREATE TRIGGER NoTableUpdate

ON DATABASE FOR DROP\_TABLE, ALTER\_TABLE

AS

PRINT 'DROP TABLE and ALTER TABLE statements are not allowed'

ROLLBACK

#### Example-2

CREATE TRIGGER DDLTrigger\_Sample

ON DATABASE

FOR CREATE\_PROCEDURE, ALTER\_PROCEDURE, DROP\_PROCEDURE

AS

BEGIN

END

## Logon Trigger

### ALTER Trigger

ALTER TRIGGER NoTableUpdate

ON DATABASE FOR DROP\_TABLE, ALTER\_TABLE

AS

PRINT 'DROP TABLE and ALTER TABLE statements are not allowed'

ROLLBACK

### DROP Trigger

DROP TRIGGER NoTableUpdate on DATABASE

### DISABLE Trigger

DISABLE TRIGGER NoTableUpdate on DATABASE

### ENABLE Trigger

ENABLE TRIGGER NoTableUpdate on DATABASE

## DDL Trigger (ON ALL Server)

CREATE TRIGGER ddl\_trig\_database

ON ALL SERVER

FOR CREATE\_DATABASE

AS

PRINT 'Database Created.'

SELECT EVENTDATA().value('(/EVENT\_INSTANCE/TSQLCommand/CommandText)[1]','nvarchar(max)')

### Drop DDL Trigger

DROP TRIGGER ddl\_trig\_database ON ALL SERVER

--

FOR | AFTER  
AFTER specifies that the DML trigger is fired only when all operations specified in the triggering SQL statement have executed successfully. All referential cascade actions and constraint checks also must succeed before this trigger fires.

* AFTER is the default when FOR is the only keyword specified.
* AFTER triggers cannot be defined on views.
* There is no difference, they do the same thing For or After

## AFTER INSERT

Statement in trigger executed first then insert or update statement is executed

CREATE TRIGGER Tr\_AFTER ON t\_FOR\_TRIGGER

FOR INSERT

AS

INSERT INTO t\_FOR\_TRIGGER\_Audit SELECT EID,[name],salary,'After Insert',getdate() from inserted

PRINT 'AFTER INSERT trigger fired.'

## AFTER UPDATE

CREATE TRIGGER Tr\_AFTER\_UPDATE ON t\_FOR\_TRIGGER

AFTER UPDATE

AS

INSERT INTO t\_FOR\_TRIGGER\_Audit SELECT EID,[name],salary,'AFter UPDATE',getdate() from inserted

PRINT 'AFTER update trigger fired.'

## AFTER DELETE

CREATE TRIGGER Tr\_AFTER\_DELETE ON t\_FOR\_TRIGGER

AFTER DELETE

AS

INSERT INTO t\_FOR\_TRIGGER\_Audit SELECT EID,[name],salary,'AFter DELETE',getdate() from deleted

PRINT 'AFTER DELETE trigger fired.'

INSTEAD OF  
Specifies that the DML trigger is executed instead of the triggering SQL statement, therefore, overriding the actions of the triggering statements. INSTEAD OF cannot be specified for DDL or logon triggers.

## View Trigger

SELECT \* FROM sys.triggers WHERE type = 'TR';

# INDEX

An index is a performance-tuning method of allowing faster retrieval of records. An index creates an entry for each value that appears in the indexed columns. Indexes can be created on **tables**, **views**, and **temporary tables**. An index can even be created before there is data in the table. Table can have only one **clustered index** and 249 **non-clustered index**.

## View Index

SELECT \* FROM sys.indexes WHERE object\_id = OBJECT\_ID('TableName');

CREATE [ **UNIQUE** ] [ **CLUSTERED** | **NONCLUSTERED** ] INDEX index\_name

**ON** table\_name ( column1 [ASC | DESC ], ... column\_n [ ASC | DESC ] )

You can have all 4 permutations:

* "unique non-clustered"
* "unique clustered"
* "non-unique non-clustered"
* "non-unique clustered"

## Clustered Index

A clustered index is actually a table where the data for the rows are **stored**. It defines the order of the table data based on the key values that can be sorted in only one way. In the database, each table can have only one clustered index. In a relational database, if the table column contains a **primary key** or **unique key**, [MySQL](https://www.javatpoint.com/mysql-tutorial) allows you to create a clustered index named **PRIMARY** based on that specific column.

* By default unique clustered index is created while creating primary key
* By default unique nonclustered index is created while creating Unique key
* It always stores the index value in a B-tree structure where the actual data is stored in the leaf node.
* Since the data rows are stored in one direction, each table can only have a single clustered index.

**Syntax—**

CREATE CLUSTERED INDEX indexName ON TableName(ColName)

## Disadvantages

The main disadvantages of the clustered index are as follows:

* It contains many insert records in a non-sequential order.
* It always takes a long time to update the records.
* It needs extra work for SQL queries, such as insert, updates, and deletes.

## Non-Clustered Index

The structure of non-clustered indexes is similar to the clustered index except that the **actual data is not contained in the leaf nodes**. A non-clustered index has the non-clustered index key values, and each key-value entry contains a reference to the actual data.

CREATE CLUSTERED INDEX indexName ON TableName(ColName)

Or

CREATE INDEX indexName ON TableName(ColName)

Since we have not specified ASC | DESC to each of the columns, the index is created with each of the fields in **ascending** order. We could modify our example and change the sort orders to descending as follows:

CREATE INDEX indexName ON TableName(ColName DESC);

## UNIQUE Index (UNIQUE NONCLUSTERED Index)

It is also called **nonclustered unique** index.

Indicates that the combination of values in the indexed columns must be unique.

CREATE UNIQUE INDEX indexName ON TableName(ColName)

This example would create an index called contacts\_uidx on that contacts table that consists of the last\_name and first\_namefields, but also ensures that the there are only unique combinations of the two fields.

## UNIQUE CLUSTERED Index

You could modify this example further to make the unique index also clustered so that the physical order of the rows in the table is determined by the logical order of the index.

CREATE UNIQUE CLUSTERED INDEX indexName ON TableName(ColName)

**Composite Index**

If you are using more than one column in creating index then this is called composite index

**RENAME INDEX**

sp\_rename 'tableName.indexName', 'NewIndexName', **'INDEX'**;

## Drop Index

**DROP INDEX** tableName.IndexName;

## Index Rebuild/Reorganize

The Database Engine automatically modifies indexes whenever insert, update, or delete operations are made to the underlying data. For example, the addition of rows in a table can cause existing pages in rowstore indexes to split, making room for the insertion of new rows.Over time these modifications can cause the data in the index to become scattered in the database (fragmented).

# Table Partitioning

Imagine trying to find a specific book in a library with millions of books and no organization system. It would be a nightmare! Partitioning is like organizing the library into different sections based on genre, author, or publication date. This makes it much easier to find what you’re looking for.

* **Faster Queries**: By targeting specific partitions, the database can quickly locate and access relevant data
* **Parallel Processing**: Queries can be executed in parallel across multiple partitions, boosting performance for large datasets. Imagine multiple librarians helping you find your book simultaneously!
* **Efficient Data Management**: Operations like data loading, deletion, and archiving can be performed on individual partitions, minimizing disruption to the entire table. You can easily remove outdated books from a specific section without affecting the rest of the library.

**Step1: Create Partition function**

* Partition function can not creted on data type , Text,image,xml
* It does not talk about table
* We can reuse partition function for many table

Step2: Create Partition scheme

* One scheme can use only one partition function
* one partition function can be used in many scheme

Step3: Create Table partition

CREATE PARTITION FUNCTION myDateRangePF (int)

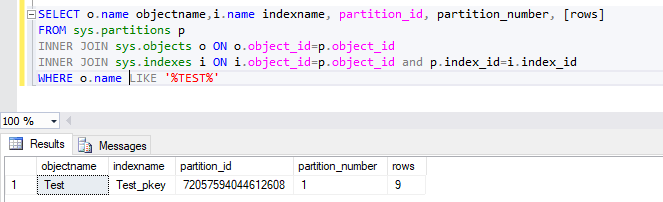
AS RANGE LEFT FOR VALUES (100, 2000,300)

Range LEFT

|  |  |  |  |
| --- | --- | --- | --- |
| Partition-1 | Partition-2 | Partition-3 | Partition-4 |
| <=100 | >100 and <=200 | >200 and <=300 | >300 |

Range Right

|  |  |  |  |
| --- | --- | --- | --- |
| Partition-1 | Partition-2 | Partition-3 | Partition-4 |
| <100 | >=100 and <200 | >=200 and <300 | >=300 |

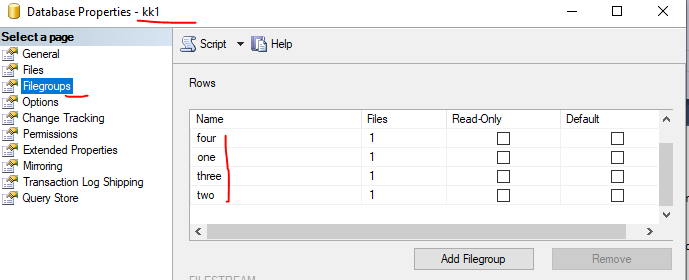


CREATE PARTITION SCHEME PartScheme

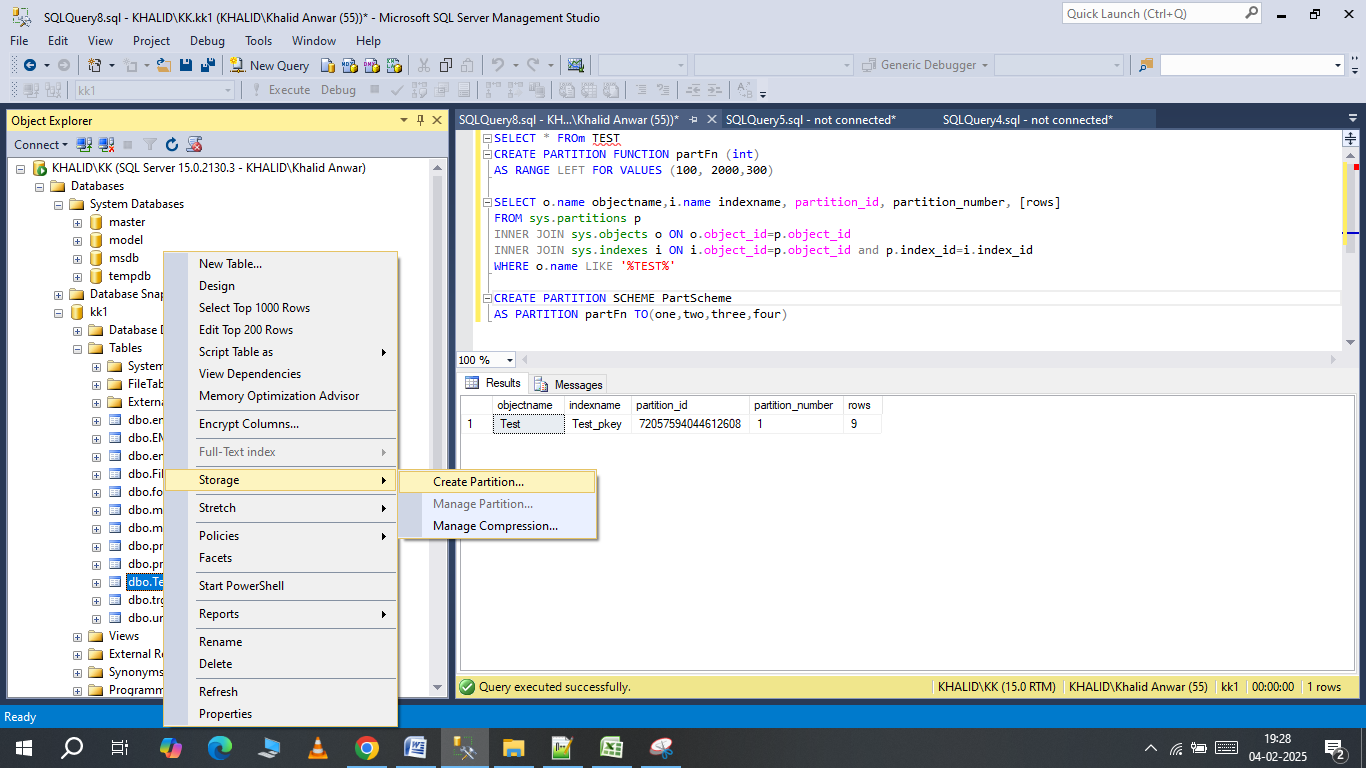
# AS PARTITION partFn TO(one,two,three,four)

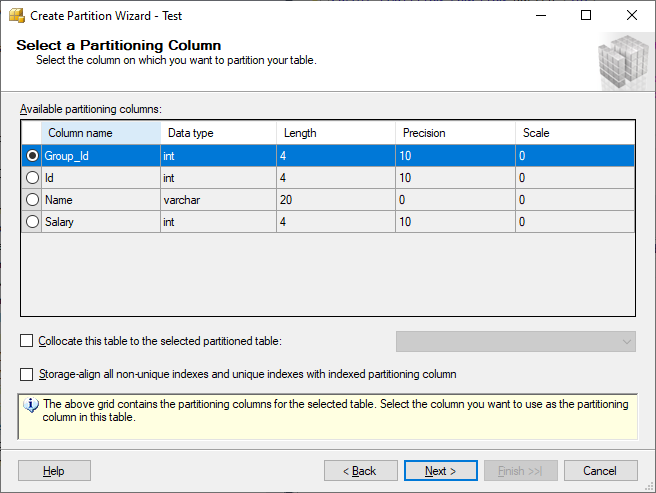
-- PartScheme scheme name

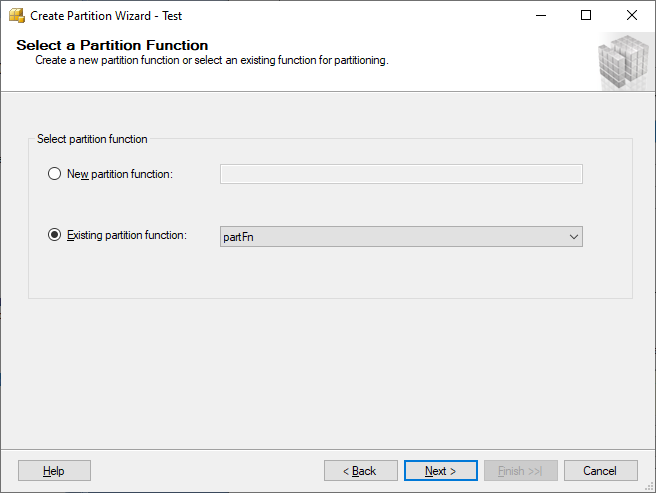
-- one,two,three,four --Filegroup name

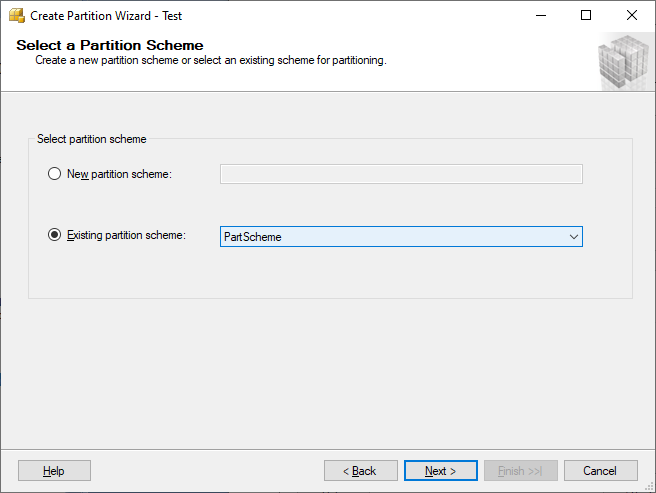


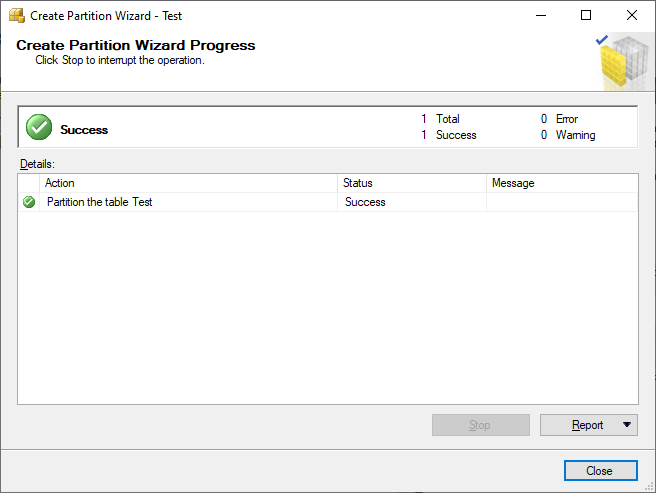
Create table partition



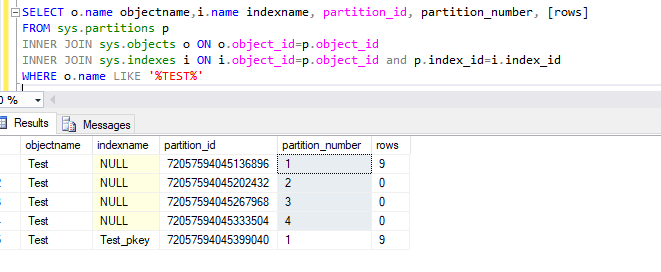








After partition creation



## View partitioon table

select schema\_name(schema\_id), name, COUNT(name) from sys.tables t

inner join sys.partitions p on t.object\_id = p.object\_id

where name='TEST'

group by schema\_name(schema\_id), name

having COUNT(name) > 1

### system table for partition table

**sys.partitions --partionid and objectid**

**sys.partition\_schemes --scheme name and partion functionid**

**sys.partition\_functions -- partion functioned and functionname**

**sys.partition\_range\_values --range value and functionid**

**sys.filegroups**

## Partition function and partion scheme and table mapping

select ps.Name AS PartitionScheme, pf.name AS PartitionFunction,fg.name AS FileGroupName

from sys.indexes i

JOIN sys.partitions p ON i.object\_id=p.object\_id AND i.index\_id=p.index\_id

join sys.partition\_schemes ps on ps.data\_space\_id = i.data\_space\_id

join sys.partition\_functions pf on pf.function\_id = ps.function\_id

join sys.allocation\_units au ON au.container\_id = p.hobt\_id

join sys.filegroups fg ON fg.data\_space\_id = au.data\_space\_id

where i.object\_id = object\_id('TEST')

## below query will give range details

SELECT

ISNULL(quotename(ix.name),'Heap') as IndexName

,ix.type\_desc as type

,prt.partition\_number

,prt.data\_compression\_desc

,ps.name as PartitionScheme

,pf.name as PartitionFunction

,case when ix.index\_id < 2 then prt.rows else 0 END as Rows

,au.TotalMB

,au.UsedMB

,s.[name]+'.'+t.[name] [TableName]

,c.name [PartByColName]

,case when (pf.boundary\_value\_on\_right = 1 and rv.value is not null) then '<' when (pf.boundary\_value\_on\_right = 0 and rv.value is not null) then '<=' else '' End as term

,rv.value

,fg.name as FilegroupName

FROM sys.partitions prt

inner join sys.indexes ix

on ix.object\_id = prt.object\_id and ix.index\_id = prt.index\_id

inner join sys.tables t

on t.object\_id = ix.object\_id AND ix.type IN (0,1)

JOIN sys.index\_columns AS ic

ON ic.[object\_id] = ix.[object\_id] AND ic.index\_id = ix.index\_id AND ic.partition\_ordinal >= 1

JOIN sys.columns AS c

ON t.[object\_id] = c.[object\_id] AND ic.column\_id = c.column\_id

inner join sys.schemas s

ON s.schema\_id=t.schema\_id

inner join sys.data\_spaces ds

on ds.data\_space\_id = ix.data\_space\_id

left join sys.partition\_schemes ps

on ps.data\_space\_id = ix.data\_space\_id

left join sys.partition\_functions pf

on pf.function\_id = ps.function\_id

left join sys.partition\_range\_values rv

on rv.function\_id = pf.function\_id AND rv.boundary\_id = prt.partition\_number

left join sys.destination\_data\_spaces dds

on dds.partition\_scheme\_id = ps.data\_space\_id AND dds.destination\_id = prt.partition\_number

left join sys.filegroups fg

on fg.data\_space\_id = ISNULL(dds.data\_space\_id,ix.data\_space\_id)

inner join (select str(sum(total\_pages)\*8./1024,10,2) as [TotalMB],str(sum(used\_pages)\*8./1024,10,2) as [UsedMB],container\_id from sys.allocation\_units group by container\_id) au

on au.container\_id = prt.partition\_id

where pf.is\_system=0

order by ps.name,rv.value,fg.name

# Conditional Statement

## IF ELSE

IF expression

**BEGIN**

     Statement block -- It executes when the IF clause expression is TRUE.

**END**

**ELSE**

**BEGIN**

    Statement block -- It executes when the IF clause expression is FALSE.

**END**

Note: Oracle has THEN key; in the last oracle has end if

### Example 1

DECLARE @age INT;

SET @age = 20;

IF @age < 25

PRINT 'Yong man';

ELSE

### PRINT 'Old man'

### Example 2

DECLARE @age INT;

SET @age = 20;

IF @age < 25

BEGIN

PRINT 'Yong man';

PRINT 'Age is less than 25';

END

ELSE

BEGIN

PRINT 'Old man'

PRINT 'Age is greater than 25';

END

## Nested IF ELSE

DECLARE @age INT;

SET @age = 80;

IF @age < 18

PRINT 'You are underage';

ELSE

BEGIN

IF @age < 50

PRINT 'You are below 50';

ELSE

PRINT 'You are senior cetizen';

END;

# While Loop

**WHILE** condition

**BEGIN**

{SQL\_statement | statement\_block | BREAK | **CONTINUE**}

**END**;

### Example1

DECLARE @i INT;

SET @i = 1;

WHILE @i <= 5

BEGIN

PRINT 'Itertion'+CAST(@i AS varchar(10))

SET @i = @i + 1;

END;

## PRINT 'While loop ended';

## BREAK Statement

SQL Server also allows us to use the BREAK statement in the WHILE loop like programming languages. This statement is used to **immediately stop the current iteration of the loop**, and control flow resumes with the next statement after the loop.

DECLARE @i INT;

SET @i = 1;

WHILE @i <= 5

BEGIN

if @i=4

break;

PRINT 'Itertion'+CAST(@i AS varchar(10))

SET @i = @i + 1;

END;

## CONTINUE Statement

SQL Server also allows us to use the CONTINUE statement in the WHILE loop like programming languages. This statement immediately **terminates the current execution of the loop when the specified condition is met**, and control flow returns to the beginning of the loop.

DECLARE @i INT;

SET @i = 1;

WHILE @i <= 5

BEGIN

SET @i = @i + 1;

if @i=4

continue;

PRINT 'Itertion'+CAST(@i AS varchar(10))

END;

## For Loop

SQL server does not support for loop

# Function

## Built-in Function

1. Aggregate Function—Count,Sum,AVG,MAX,MIN
2. Numeric Function
3. Conversion Function
4. String Function
5. Date and Time Function

## User Defined Function

## Numeric Function

ABS(number)—Absolute value of number

SELECT ABS(-35.6) ---35.6

SELECT ROUND(35.6) –36

SELECT FLOOR(35.6) —36 which ignore decinmal value

SELECT MOD(11,3) --2 Remainder MOD(dividend,divisor)

SELECT POWER(3,2) --9 POWER(number,power)

SELECT SQRT(16) --4 Square root

SELECT GREATEST(16,25,3) –-25 maximum among list GREATED(col1,col2,…)

SELECT LEAST(16,25,3) –-3 minimm among list LEAST(col1,col2,…)

SELECT RAND() --Random number for every execution,value from 0 to 1, e.g 0.988888989

## String Function

## CHARINDEX

Not available in MySql

CHARINDEX( Searching\_string, string, [start\_position] ) –Similar in oracle: INSTR

SELECT CHARINDEX('t','Teachert',1),CHARINDEX('t','Teachert',7)

SELECT name,CHARINDEX(',',name) FRom Data --index of first occurance of , ; index start at 1

## LEN, Replace,..

LEN(STRING) -- similar in oracle/MySql LENGTH(string)

<https://www.w3schools.com/mysql/mysql_ref_functions.asp>

SELECT LEN('Teachert')

SELECT LEFT('SQL Tutorial.com', 3) AS "LFT",RIGHT('SQL Tutorial.com', 3)"RHT"

SELECT LOWER('THE khalid')"lower",UPPER('THE khalid')"upper"

SELECT REPLACE('Hello Mr Khalid', 'Mr', 'Mrs');-- Hello mrs Khalid

SELECT COALESCE(col1, col2, col3);-- if col1 is null then col2 will be returned

SELECT CONCAT("SQL ", "Tutorial ", "is ", "fun!") AS ConcatenatedString;

SELECT CONCAT\_WS(" ","SQL ", "Tutorial ", "is ", "fun!")

SELECT SUBSTRING('MdKhalid', 1, 3);-- start at position 1, extract 3 characters

SELECT CONCAT(LTRIM(" SQL"),RTRIM("Tutorial "),TRIM(" Fun "))

|  |  |
| --- | --- |
| **Function** | **Description** |
| ASCII | Returns the number code that represents the specific character |
| CHAR | Returns the ASCII character based on the number code |
| CHARINDEX | Returns the location of a substring in a string |
| Concat | Concatenates two or more strings together;Concat(str1,str2,str3,…) |
| Concat\_WS | Concate with separator;concat(separator,str1,str2,str3) |
| DATALENGTH | Returns the length of an expression (in bytes) |
| LEN | Returns the length of the specified string |
| LOWER | Converts a string to lower-case |
| UPPER | Converts a string to upper-case |
| LTRIM | Removes leading spaces from a string |
| RTRIM | Removes trailing spaces from a string |
| NCHAR | Returns the Unicode character based on the number code |
| PATINDEX | Returns the location of a pattern in a string |
| REPLACE | Replaces a sequence of characters in a string with another set of characters  REPLACE(string1, **string\_to\_replace**, **string\_with**) |
| REVERSE | It will reverse string; reverse(“StringValue”) |
| SPACE | Returns a string with a specified number of spaces |
| STR | Returns a string representation of a number |
| LEFT | Extracts a substring from a string (starting from left) |
| RIGHT | Extracts a substring from a string (starting from right) |
| SUBSTRING | Extracts a substring from a string |

### [How to count number of occurrence of character in string field](https://stackoverflow.com/questions/1860457/how-to-count-instances-of-character-in-sql-column)

This will return number of occurance of N

select ColumnName, **LEN**(ColumnName)- **LEN**(**REPLACE**(ColumnName, 'N', '')) from Table

## IFNULL(), ISNULL(), COALESCE(), and NVL()

**My SQL** --IFNULL and COALESC is same, if first expression is null then second value

**Oracle**—NVL works similar

**SQL Server**—ISNULL works similar

DECLARE @age INT;

SET @age = null;

SELECT isnull(@age,10) --print 10

## ISNUMERIC in SQL Server

ISNUMERIC returns 1 when the input expression evaluates to a valid numeric data type; otherwise it returns 0

# Conversion Function

Number🡨🡪Character🡨🡪date

## CAST

The CAST() function converts a value (of any type) into the specified datatype.

CAST( expression **AS** Data\_type)

SELECT CAST(14.85 AS int);

SELECT CAST(14.85 AS float);

SELECT CAST(15.6 AS varchar);

SELECT CAST(15.6 AS varchar(4));

SELECT CAST('15.6' AS float);

SELECT CAST('2014-05-02' AS datetime);-- Format: "YYYY-MM-DD"

SELECT CAST("14:06:10" AS TIME); --

## CONVERT

The CONVERT function converts an expression from one datatype to another datatype

CAST( expression **,** Data\_type)

SELECT CAST(14.85 AS int),CONVERT(int, 14.85);

SELECT CAST(14.85 AS float),CONVERT(float,14.85)

SELECT CAST(15.6 AS varchar), CONVERT(varchar, 15.6)

SELECT CAST(15.6 AS varchar(4)), CONVERT(varchar(4), 15.6)

SELECT CAST('15.6' AS float), CONVERT(float, '15.6')

SELECT CAST('2014-05-02' AS datetime),CONVERT(datetime, '2014-05-02')

**Different from CAST**

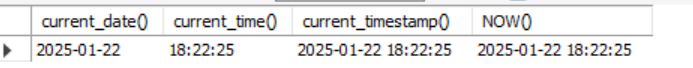
CONVERT accepts an optional style parameter used for formatting.  Other than that, they are pretty much the same as CAST.

SELECT CONVERT(VARCHAR, getdate(), 101) as MMDDYYYY,

CONVERT(VARCHAR, getdate(), 111) as YYYYMMDD;

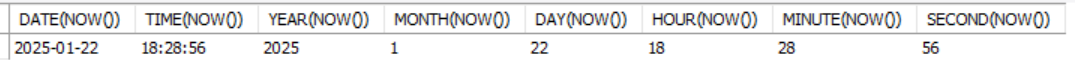
# DATE function

In MySql

SELECT current\_date(),current\_time(),current\_timestamp(),NOW()

SELECT **DATE**(NOW()),**TIME**(NOW()),**YEAR**(NOW()),**MONTH**(NOW()),

**DAY**(NOW()),**HOUR**(NOW()),**MINUTE**(NOW()),**SECOND**(NOW()),**MONTHNAME**(NOW()) --January



## DATEADD

DATEADD( interval, number, date )

SELECT DATEADD(year, 1, '2014/04/28'),DATEADD(quarter, 1, '2014-04-28')

SELECT DATEADD(month, 1, '2014/04/28'),DATEADD(week, 1, '2014-04-28'),DATEADD(week, 1, '2014-03-28')

SELECT DATEADD(day, 1, '2014/04/28'),DATEADD(hour, 1, '2014-04-28'),DATEADD(minute, 1, '2014-04-28')

In My SQL we have Date\_Add(dateColumn, INTERVAL unit Interval\_unit

SELECT **Date\_add**(now(),**INTERVAL** 1 month)

|  |  |
| --- | --- |
| Interval Value (any one of) | Explanation |
| year, yyyy, yy | Year interval |
| quarter, qq, q | Quarter interval |
| month, mm, m | Month interval |
| dayofyear | Day of year interval |
| day, dy, y | Day interval |
| week, ww, wk | Week interval |
| weekday, dw, w | Weekday interval |
| hour, hh | Hour interval |
| minute, mi, n | Minute interval |
| second, ss, s | Second interval |
| millisecond, ms | Millisecond interval |

## DATEDIFF

DATEDIFF( interval, date1, date2 ) -- date2 minus date1

SELECT DATEDIFF(**year**,'2014/04/28','2015/04/28'),DATEDIFF(**quarter**,'2014-04-28','2015-04-28')

SELECT DATEDIFF(**month**,'2014-04-28','2015/04/28'),DATEDIFF(**week**, '2014-04-28', '2015-04-28')

SELECT DATEDIFF(**day**, '2014-04-28', '2014/04/30'),DATEDIFF(**hour**, '2014-04-28 00:00:00', '2014-04-28 01:00:00'),DATEDIFF(**minute**,'2014-04-28 00:00:00', '2014-04-28 01:00:00')

In **MySQL** , we have DATEDIFF (date1,date2) -- Default interval is day .

## DATEPART

In MSSQL

the DATEPART function returns a specified part of a given date, as an integer value.

DATEPART( interval, date )

SELECT DATEPART(month, '2014/04/28'),DATEPART(mm, '28 APR 2014')

SELECT DATEPART(minute, '2014/04/28 09:49'),DATEPART(second, '2014/04/28 09:49:12')

SELECT DATEPART(m, '2014/04/28 09:49'),DATEPART(s, '2014/04/28 09:49:12')

In **MySQL** , we have EXTRACT(component from date)

SELECT **EXTRACT**(WEEK FROM NOW()),**EXTRACT**(QUARTER FROM NOW())

Componenet as below

|  |  |  |
| --- | --- | --- |
| YEAR |  | HOUR |
| MONTH |  | MINUTE |
| DAY |  | SECOND |
| WEEK |  | QUARTER |

## DATENAME

the DATEPART function returns a specified part of a given date, as a string value.

SELECT DATENAME(month, '2014/04/28')

Result=April

## MONTH(date)

In both MS SQL and MySQL

It is equivalent to DATEPART(MONTH,date)

SELECT MONTH('2014/04/28');

Result=4

SELECT MONTH('2014/03/31 10:05');

Result=3

## DAY(date)

In both **MS SQL** and **MySQL**

It is equivalent to DATEPART(DAY,date)

SELECT DAY('2014/04/28');

Result: 28

SELECT DAY('2014/03/31 10:05');

Result: 31

SELECT DAY('2014/04/01 10:05:18.621');

Result: 1

## YEAR(date)

In both **MS SQL** and **MySQL**

It is equivalent to DATEPART(YEAR,date)

SELECT YEAR('2014/04/28');

Result: 2014

SELECT YEAR('2013/03/31 10:05');

Result: 2013

SELECT YEAR('2015/12/01 10:05:18.621');

Result: 2015

## Calculate Age

SELECT a.\*,

CASE WHEN MONTH(today\_date)>MONTH(dob) OR

(

MONTH(today\_date)=MONTH(dob) AND DAY(today\_date)>DAY(dob)

) THEN

DATEDIFF(year,dob,today\_date) ELSE DATEDIFF(year,dob,today\_date)-1 END AS 'AGE'

FROM t\_Date a

## Calculate days

SELECT datediff(day, getdate(), dateadd(month, 1, getdate()))—current month

SELECT datediff(day, '2018-03-15', dateadd(month, 1, '2018-03-15'))—particular

Caution

SELECT DATEDIFF(DAY,'2018-01-30',DATEADD(MONTH,1,'2018-01-30'))—this gives 29 as february is of 28. So it is better to be careful while selecting that. It should be 1 to 28 .

# CASE

## Case-Type1

CASE expression

WHEN value\_1 THEN result\_1

WHEN value\_2 THEN result\_2

WHEN value\_n THEN result\_n

ELSE result

END

## Case-Type2

CASE

WHEN condition\_1 THEN result\_1

WHEN condition\_2 THEN result\_2

WHEN condition\_n THEN result\_n

ELSE result

END

# COALESCE

the COALESCE function returns the first non-null expression in the list. If all expressions evaluate to null, then the COALESCE function will return null.

COALESCE( expression1, expression2, ... expression\_n )—exp must be in same data type

SELECT COALESCE(NULL, NULL, 1, 2, 3, NULL, 4);

SELECT COALESCE(NULL,'Hello',1) --Gives error because expression must be same data type

# ****ISDATE****

* The ISDATE function returns 1, if the *expression* is a valid date.
* The ISDATE function returns 0, if the *expression* is NOT a valid date

SELECT ISDATE('2014-05-01'),ISDATE('2014-05-01 10:03:32.001'),ISDATE('techonthenet.com'),ISDATE(123)

# ISNULL(oracle=NVL)

the ISNULL function lets you return an alternative value when an expression is NULL.

ISNULL( expression, alternative\_value )

SELECT ISNULL(NULL, 'Te'),ISNULL('xyz', 'yzx')

# ISNUMERIC

 the ISNUMERIC function returns 1 if the expression is a valid number. Otherwise, it returns 0.

ISNUMERIC( expression )

SELECT ISNUMERIC(1234),ISNUMERIC('1234'),ISNUMERIC('te'),ISNUMERIC('2014-05-01');

For both integer and decimal datatype, isnumeric returns 1

# Like

|  |  |  |
| --- | --- | --- |
| **Character** | **Description** | **Example** |
| % | Any string of zero or more characters. | LIKE '5[%]' means 5% |
| \_ | Any single character. | LIKE '[\_]n' means \_n |
| [] | Any single character within the specified range [a-f] or set [abcdef] | LIKE '[ [ ]' means [ |
| [^] | Any single character not within the specified range [^a-f] or set [^abcdef] | LIKE 'abc[def]' means abcd, abce, or abcf |

# CEILING FLOOR

The CEILING function returns the smallest integer greater than or equal to the specified numeric expression. The FLOOR function returns the largest integer less than or equal to the specified numeric expression. For example, in considering a numeric expression of 12.9273, CEILING returns 13 and FLOOR returns 12. The return value of both FLOOR and CEILING has the same data type as the input numeric expression.

|  |  |  |
| --- | --- | --- |
| EID | CEILING | FLOOR |
| 11 | 11 | 11 |
| 11.11 | 12 | 11 |
| 13 | 13 | 13 |
| 13.13 | 14 | 13 |
| 13.59 | 14 | 13 |
| 13.99 | 14 | 13 |
| 13.01 | 14 | 13 |
| 13.5 | 14 | 13 |
| -13.59 | -13 | -14 |
| -13.99 | -13 | -14 |
| -13.01 | -13 | -14 |
| -13.5 | -13 | -14 |

# NULLIF

 the NULLIF function compares expression1 and expression2. If expression1 and expression2 are equal, the NULLIF function returns NULL. Otherwise, it returns the first expression which is expression1.

If exp1==exp2 then return **NULL** else **exp1**

NULLIF( expression1, expression2 )

SELECT NULLIF(12, 12),NULLIF(12, 45),NULLIF('2014-05-01', '2014-05-01'),NULLIF('A','a')

# GREATEST(NA in SQL) --in MySql

There is no such function in SQL server. We can use **case** function for workaround.

Maximum of all subject as below.

SELECT CASE

WHEN MATH>=phy AND MATH>=Che AND MATH>=Hindi THEN MATH

WHEN PHY>=MATH AND PHY>=Che AND PHY>=Hindi THEN PHY

WHEN CHE>=MATH AND CHE>=PHY AND CHE>HINDI THEN CHE

WHEN HINDI>=MATH AND HINDI>=PHY AND HINDI>CHE THEN HINDI

END AS maxsubject FROM STUDENt2

# LEAST(NA in SQL) --In MySql

There is no such function in SQL server. We can use **case** function for workaround.

SELECT CASE

WHEN MATH<=phy AND MATH<=Che AND MATH<=Hindi THEN MATH

WHEN PHY<=MATH AND PHY<=Che AND PHY<=Hindi THEN PHY

WHEN CHE<=MATH AND CHE<=PHY AND CHE<HINDI THEN CHE

WHEN HINDI<=MATH AND HINDI<=PHY AND HINDI<CHE THEN HINDI

END AS minsubject FROM STUDENt2

# ANY

The ANY operator returns TRUE if any of the subquery values meet the condition. The ANY comparison condition is used to compare a value to a list or subquery. It must be preceded by =, !=, >, <, <=, >= and followed by a list or subquery.

If ANY= then it is equivalent to IN as below

SELECT ProductName FROM Products WHERE ProductID = ANY (SELECT ProductID FROM OrderDetails WHERE Quantity = 10);

SELECT ProductName FROM Products WHERE ProductID >= ANY (SELECT ProductID FROM OrderDetails WHERE Quantity = 10);

# ALL

The ALL operator returns TRUE if all of the subquery values meet the condition. The ALL comparison condition is used to compare a value to a list or subquery. It must be preceded by =, !=, >, <, <=, >= and followed by a list or subquery.

SELECT ProductName FROM Products WHERE ProductID >= ALL (SELECT ProductID FROM OrderDetails WHERE Quantity = 10);

# UNION

In order to perform a UNION, the data being combined must meet two conditions.First, the number and the order of the columns must be the same. Next, the data types must be compatible.  They need not be the same exact same type, but they must be of a type that SQL Server can implicitly convert.

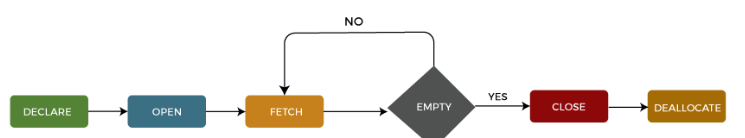
* Compatible like smallint,Float,Decimal,INT are compatible
* **Either the datatype used should be same in two table**(or)**you should use cast or convert function to match the datatypes in those two tables**.

# CURSOR

Cursor is a Database object which allows us to process each row at a timeand manipulate its data. A **Cursor is always associated with a Select Query** and it will process each row returned by the Select Query one by one.

* **cursor's purpose is to update the data row by row, change it, or perform calculations that are not possible when we retrieve all records at once**.

### Life Cycle of the cursor



**DECLARE** cursorName **CURSOR**  **FOR** select\_statement;

**OPEN** cursorName ;

**FETCH** **NEXT** **FROM** cursorName  **INTO** variable\_list;

**CLOSE** cursorName ;

**DEALLOCATE** cursorName ;

### @@FETCH\_STATUS

We can also use the **@@FETCHSTATUS function** in SQL Server to get the status of the most recent FETCH statement cursor that was executed against the cursor. The **FETCH** statement was successful when the @@FETCHSTATUS gives **zero output**. Otherwise gives **-1**

### Example1

DECLARE @id int,@name varchar(20),@sal int

DECLARE c CURSOR FOR SELECT \* FROM emp

open c

Fetch next from c into @id,@name,@sal

while @@FETCH\_STATUS=0

BEGIN

update emp set id=@id+1,name=@name+'A',salary=@sal+1 WHERE id=@id and name=@name and salary=@sal

Fetch next from c into @id,@name,@sal

END

close c

### deallocate c

# Filter Operator

|  |  |
| --- | --- |
| **Condition** | **Description** |
| **=** | Tests for an equal condition. |
| **<>** | Tests for a not-equal condition. |
| **!=** | Tests for a not-equal condition. |
| **>** | Tests for a greater-than condition. |
| **>=** | Tests for a greater-than or equal-to condition. |
| **!>** | Tests for a not-greater-than condition. |
| **<** | Tests for a less-than condition. |
| **<=** | Tests for a less-than or equal-to condition. |
| **!<** | Tests for a not-less-then condition. |
| **[ NOT ] LIKE** | Tests for a matching pattern. |
| **ESCAPE ‘escape\_character’** | Allows a wildcard character to be searched for. |
| **[ NOT ] BETWEEN** | Tests for a between condition. The AND keyword separates the starting and ending values. |
| **IS [ NOT ] NULL** | Tests for a null or optionally a not-null condition. |
| **CONTAINS** | Tests for fuzzy matching or words or phrases. |
| **[ NOT ] IN** | Tests if a value is included or excluded from a list. The list can be a set of constants enclosed in parentheses or a subquery. |

# ESCAPE

Wildcard Character s

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| % | \_ | [ | ] | [] | ^ |

You can search for character strings that include one or more of the special wildcard characters. For example, the discounts table in a customers database may store discount values that include a percent sign (%). To search for the percent sign as a character instead of as a wildcard character, the ESCAPE keyword and escape character must be provided. For example, a sample database contains a column named comment that contains the text 30%. To search for any rows that contain the string 30% anywhere in the comment column, specify a WHERE clause such as WHERE comment LIKE '%30!%%' ESCAPE '!'. If ESCAPE and the escape character are not specified, the Database Engine returns any rows with the string 30.

If there is no character after an escape character in the LIKE pattern, the pattern is not valid and the LIKE returns FALSE. If the character after an escape character is not a wildcard character, the escape character is discarded and the character following the escape is treated as a regular character in the pattern. This includes the percent sign (%), underscore (\_), and left bracket ([) wildcard characters when they are enclosed in double brackets ([ ]). Also, within the double bracket characters ([ ]), escape characters can be used and the caret (^), hyphen (-), and right bracket (]) can be escaped.

To search string containg %

SELECT \* FROM wildcard where name like '%|%%' ESCAPE '|'

SELECT \* FROM wildcard where name like '%[%]%'

**In MySQL**

For searching % character, we can use escape character **\**

where name like '%**\%**%'

or name like '%**\.**%' # searching .

or name like '%**\\_**%' # searching \_

# Using Wildcard Characters As Literals

|  |  |
| --- | --- |
| Symbol | Meaning |
| LIKE '5[%]' | 5% |
| LIKE '[\_]n' | \_n |
| LIKE '[a-cdf]' | a, b, c, d, or f |
| LIKE '[-acdf]' | -, a, c, d, or f |
| LIKE '[ [ ]' | [ |
| LIKE ']' | ] |
| LIKE 'abc[\_]d%' | abc\_d and abc\_de |
| LIKE 'abc[def]' | abcd, abce, and abcf |

# System stored procedure

## SP\_Help

It returns about bject details in tabular form with few column. View,DML Trigger or table in current database.

**Sp\_help** object\_name -- object name could be table,view,function,trigger,index

# SP\_HELPTRIGGER

Returns the type or types of DML triggers defined on the specified table for the current database.

sp\_helptrigger cannot be used with DDL triggers.

**SP\_HELPTRIGGER** tableName

# SP\_HELPTEXT

Displays the definition of a user-defined “**Procedure**”,**Function**,**View**,**trigger**

**SP\_HELPTEXT** Tr\_AFTER

# SP\_EXECUTESQL

It accept sql statement in data type ntext/nchar/nvarchar

EXEC SP\_EXECUTESQL N'SELECT \* FROM DATA2'

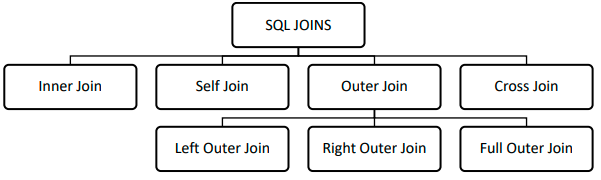
--

DECLARE @str nvarchar(100);

SET @str='SELECT ''khalid'' as kk UNION SELECT ''Anwar'' as kk'

EXEC SP\_EXECUTESQL @str

# JOINS



**Explicit Join--** without using join keyword

select \* from inventory,sale where sale.inventoryid=inventory.inventoryid

**inner join**

select \* from inventory inner join sale on sale.inventoryid=inventory.inventoryid

**Full outer join** (shows everything)

select sale.inventoryid,inventoryname from inventory full outer join sale on sale.inventoryid=inventory.inventoryid

**left join** (might have NULL value, since some inventory might not have sales)

select inventory.inventoryid,inventoryname from inventory left join sale on sale.inventoryid=inventory.inventoryid

**right join**

select sale.inventoryid,inventoryname from inventory right join sale on sale.inventoryid=inventory.inventoryid

**Self Join**

SELECT \* From staff E inner join staff M on E.managerID = M.employeeID

# Natural Join

The SQL NATURAL JOIN is a type of EQUI JOIN and is structured in such a way that, columns with the same name of associated tables will appear once only.

* The associated tables have one or more pairs of identically named columns.
* The columns must be the same data type.
* **Don’t use ON** clause in a natural join.

SELECT \* FROM table1 NATURAL JOIN table2;

*The INNER JOIN using ON clause do the same job as natural Join.*

A NATURAL JOIN can be an INNER join, a LEFT OUTER join, or a RIGHT OUTER join. The default is INNER join.

## Syntax

[***TableExpression***](https://docs.oracle.com/javadb/10.6.2.1/ref/rreftableexpression.html#rreftableexpression) **NATURAL [ { LEFT | RIGHT } [ OUTER ] | INNER ] JOIN {** [**TableViewOrFunctionExpression**](https://docs.oracle.com/javadb/10.6.2.1/ref/rrefsqlj33215.html#rrefsqlj33215) **| (** [***TableExpression***](https://docs.oracle.com/javadb/10.6.2.1/ref/rreftableexpression.html#rreftableexpression) **) }**

Difference between natural join and inner join

There is one significant difference between INNER JOIN and NATURAL JOIN is the number of columns returned. Joining key column will not come twice as it come in inner join.

# SQL UNIONS

### Union

* to combine two tables together (but the no. of columns & each column’s data types for 2 tables must be match)
* don't need common key, only need common attributes
* merge/combine, not showing duplicate record

SELECT \* FROM empTgt

UNION

SELECT \* FROM empTgt1

### Union All

* all union rules except return duplicate record

SELECT \* FROM empTgt --20 record

UNION All

SELECT \* FROM empTgt1 --11 record

--above query is returned with 31 record

### Intersect

* keep only the first table rows in common to both query
* not showing duplicate record
* removing duplicate record from first table

SELECT \* FROM empTgt

INTERSECT

SELECT \* FROM empTgt1

### Except --similar to minus

* Works similar to MINUS query
* removing duplicate record from first table

## Find duplicate record

SELECT \* FROM

(SELECT \*,ROW\_NUMBER() OVER(PARTITION BY sn,name,school ORDER BY (SELECT 0)) AS RN

FROM DUPLICATE) TMP

WHERE RN>1

-

SELECT \* FROM

(SELECT \*,ROW\_NUMBER() OVER(PARTITION BY sn,name,school ORDER BY sn) AS RN

FROM DUPLICATE) TMP

WHERE RN>1

## To remove duplicate record

**In MS SQL**

DELETE TMP FROM

(SELECT \*,ROW\_NUMBER() OVER(PARTITION BY sn,name,school ORDER BY sn) AS RN

FROM DUPLICATE) TMP

WHERE RN>1

**Using CTE**

WITH TMP AS

(SELECT \*,ROW\_NUMBER() OVER(PARTITION BY sn,name,school ORDER BY sn) AS RN

FROM DUPLICATE

)

DELETE FROM TMP WHERE RN>1

## Show All Rows with an Above-Average Value

SELECT

  first\_name,

  last\_name,

  salary

FROM employee

WHERE salary > ( SELECT AVG(salary) FROM employee )

# CTE -- Common Table Expression

A Common Table Expression (CTE) in SQL is a temporary result set that

* can be referenced within a **SELECT**, **INSERT**, **UPDATE**, or **DELETE** statement.
* an essential tool for simplifying **complex queries** and making them more readable.
* used within the execution scope of a **SELECT**, **INSERT**, **UPDATE**, or **DELETE** statement.
* Breaking down complex queries into smaller, reusable components.
* Improving code readability and modularity.
* Enabling recursive operations for hierarchical data.

**Syntax**

WITH **cteName** AS (

SELECT query

)

SELECT \*

FROM **cteName**;

### Limitation

* **Temporary Scope**: A CTE exists only during the execution of the query. Once the query completes, the CTE is discarded.
* **Performance Issues**: For very large datasets, CTEs can sometimes lead to performance degradation due to multiple references to the same CTE.
* **Not Allowed in All Database Operations**: Some operations, such as **INSERT** and [UPDATE](https://www.geeksforgeeks.org/sql-update-statement/), may have restrictions when using CTEs in certain databases.

# Sub Query

### In select column

SELECT \*, (SELECT AVG(Salary) from empTgt) As avgSalary FROM empTgt

### In Derived Table

SELECT \* FROM

(SELECT \* FROM empTgt WHERE id>100) AS tt

### In Where Clause

SELECT \* FROM empTgt WHERE ID IN (SELECT ID FROM empTgt WHERE id>100)

# Correlated Subqueries

A correlated subquery is evaluated once for each row processed by the outer query. It is sub query that uses values from outer query. It is top to down approach

The parent statement can be a SELECT, UPDATE, or DELETE statement.

## Select Correlated

SELECT column1, column2,FROM table1 outer WHERE column1 operator

(SELECT column1, column2 FROM table2

WHERE expr1 = outer.expr2);

## Update Correlated

UPDATE table1 **alias1**

SET column = (SELECT expression FROM table2 **alias2** WHERE **alias1**.column =**alias2**.column);

## Delete Correlated

DELETE FROM table1 **alias1**

WHERE column1 operator (SELECT expressionFROM table2 **alias2** WHERE **alias1**.column = **alias2**.column);

<https://www.geeksforgeeks.org/sql-correlated-subqueries/>

## To find Employee with Second Highest Salary

1.

SELECT MAX(SALARY) FROM EMPloyees WHERE SALARY not in (SELECT MAX(SALARY) FROM EMPLOYEES)

2.

SELECT name, MAX(salary) AS salary FROM employee WHERE salary < (SELECT MAX(salary) FROM employee);

3.

SELECT \* FROM

(SELECT \*,DENSE\_RANK() OVER(ORDER BY SALARY DESC) AS rnk FROM EMPloyees) T

where rnk=2

4.

Select name,salary from employee A

where n-1 = (Select **count**(**distinct** salary) from employee **B** where **B.salary**>A.salary)

**Department wise maximum salary**

SELECT DEPT,MAX(SALARY) FROM employee2 GROUP BY DEPT

**Employee with Department ‘s maximum salary**

SELECT \* FROM TEST a WHERE Salary in

(SELECT MAX(SALARY) FROM TEST b WHERE a.GROUP\_ID=B.GROUP\_ID)

**Department wise 2nd high salary**

SELECT \* FROM EMPLOYEE2 ORDER BY DEPT,SALARY

(

SELECT \*, **dense\_rank**() **OVER**(**PARTITION BY** DEPT **ORDER BY** Salary desc) as drnk FROM EMPLOYEE2 ) A WHERE drnk=2

--Using correlated sub Query

SELECT a.deptno, a.salary FROM Emp a

WHERE 1 = (SELECT COUNT(DISTINCT salary)

FROM Emp b WHERE b.salary > a.salary AND b.deptno = a.deptno)

group by a.deptno

**Find manager list**

SELECT Distinct **e2.employeeNumber** As Mgr FROM emp e1 inner JOIN emp e2

on **e2.employeeNumber**=**e1.reportsTo**

**find department with highest average salary for employees who is for more than 2 years.**

SELECT d.DepartmentName, AVG(e.Salary) AS AverageSalary FROM Employees e

INNER JOIN Departments d ON e.DepartmentID = d.DepartmentID

WHERE e.HireDate < DATEADD(year, -2, GETDATE())

GROUP BY d.DepartmentName ORDER BY AverageSalary DESC LIMIT 1;

## Show All Rows with an Above-Average Value

SELECT first\_name,last\_name,salary

FROM employee

WHERE salary > (SELECT **AVG**(salary) FROM employee )

**Employees with Salaries Higher Than Their Departmental Average**

SELECT first\_name,last\_name,salary

FROM employee e1

WHERE salary >

    (SELECT **AVG**(salary) FROM employee e2

     WHERE e1.departmet\_id = e2.department\_id)

You can add one more condition like dept,project wise maxm salary

name of the employee and of his/her department belonging to a department having less than 2 i.e. 1 employee

SELECT EMPLOYEE\_NAME, DEPARTMENT\_NAME

FROM COMPANY WHERE DEPARTMENT\_NAME IN

(SELECT DEPARTMENT\_NAME FROM COMPANY

GROUP BY DEPARTMENT\_NAME HAVING COUNT(\*)<2);

**Conditional Sum**

Find sum of salary for group by department

SELECT

  SUM (

CASE WHEN dept\_id IN ('SALES','HUMAN RESOURCES') THEN salary

     ELSE 0 END

) AS **total\_salary\_sales\_and\_hr**,

  SUM (

CASE WHEN dept\_id IN ('IT','SUPPORT') THEN salary

    ELSE 0 END

) AS **total\_salary\_it\_and\_support**

FROM employee

**Group Rows by a Range**

SELECT

CASE

WHEN salary <= 20000 THEN 'low'

WHEN salary > 20000 AND salary <= 50000 THEN 'medium'

WHEN salary > 50000 THEN 'high'

END AS salary\_category,

COUNT(\*) AS number\_of\_employees

FROM Emp

GROUP BY

CASE

WHEN salary <= 20000 THEN 'low'

WHEN salary > 20000 AND salary <= 50000 THEN 'medium'

WHEN salary > 50000 THEN 'high'

END

## EXISTS/NOT EXISTS

* The EXISTS operator is used to test for the existence of any record in a subquery/correlated sub query.
* The EXISTS operator returns TRUE if the subquery returns one or more records.

**Syntax**

SELECT column\_name(s) FROM table\_name  
WHERE EXISTS  
(SELECT column\_name FROM table\_name WHERE condition);

**Example**

If there is no correlated query then it will fetch all record from outer query. So better to use correlated sub query.

SELECT SupplierName FROM Suppliers

WHERE EXISTS (SELECT ProductName FROM Products WHERE Products.SupplierID = Suppliers.supplierID);

Output// above query will fetch all matching record where SupplierID is matching in both table

SELECT SupplierName FROM Suppliers

WHERE NOT EXISTS (SELECT ProductName FROM Products WHERE Products.SupplierID = Suppliers.supplierID AND Price < 20);

Output// above query will fetch all matching record where SupplierID is not matching in both table

## ****How can you optimize a slow-running query?****

* There are several techniques, including:
* Using appropriate indexes
* Avoiding unnecessary joins and subqueries
* Using efficient functions and operators
* Analyzing execution plans to identify bottlenecks

# Ranking Function

* The ranking functions always assign rank on basis of **ORDER BY** clause.
* These functions are always used with **OVER()** clause.
* The assignment of rank to rows always start with 1 for every new partition.

ROW\_NUMBER will always generate unique values without any gaps, even if there are ties.

RANK can have gaps in its sequence and when values are the same, they get the same rank. **With gaps**

DENSE\_RANK also returns the same rank for ties, but doesn’t have any gaps in the sequence. **Without gaps**

SELECT \*,

**row\_number**() **OVER**(ORDER BY MARKS) As rn,

**RANK**() **OVER**(ORDER BY MARKS) As rnk,

**dense\_rank**() **OVER**(ORDER BY MARKS) As drnk

FROM pivot1

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| NAME | COLLEGE | ROLL\_NO | SUBJECT | MARKS | rn | rnk | drnk |
| PUSHKAR | MSIT | 898888 | GRAPHICS | 78 | 1 | 1 | 1 |
| ROMY | BVP | 261150 | NETWORKING | 87 | 2 | 2 | 2 |
| ROMY | BVP | 261150 | DBMS | 90 | 3 | 3 | 3 |
| PUSHKAR | MSIT | 898888 | NETWORKING | 90 | 4 | 3 | 3 |
| PUSHKAR | MSIT | 898888 | DBMS | 91 | 5 | 5 | 4 |
| ROMY | BVP | 261150 | GRAPHICS | 95 | 6 | 6 | 5 |

# Regular Expression in MySql-- REGEXP

**Case 1** − If you want only those rows which have exactly 10 digits and all must be only digit, use the below regular expression.

SELECT \* FROM yourTableName WHERE yourColumnName **REGEXP** '^[0-9]{10}$';

**Case 2 –** only integer value of **any length**

SELECT \*FROM yourTableName WHERE yourColumnName REGEXP '**^**[0-9]\*$';

**Case3-** mobile number start with 8 or 9 and should be of 10 digit REGEXP '[89][0-9]{9}';

**Case4**- password contains First capital letter, followed by small letter and one integer in between

[A-B][a-z]\*[0-9][a-z]\*

**Case5:** EmailId like [khalid0142759@gmail.com](mailto:khalid0142759@gmail.com) and contains \_ . –

[a-zA-Z0-9\\_\.\-]\*[@][a-z]\*[\.][a-z]{2,3}

**Case6**  search .(dot) in a string or %

Regexp ‘[\.]’

Regexp ‘[\%]’

**Case6**  search \(backsalsh) in a string.. need to use thrice because two \\ will be escaped.

Regexp ‘\\\’

|  |  |  |
| --- | --- | --- |
| **Pattern** | **What the Pattern matches** | **Example** |
| ^ | caret(^) matches **Beginning** of string | '^man' == Like 'man%' ;beginning with this word '^[0-9]\*$' == first character will be from 0 to 9 and remaing could be any character of any length |
| $ | **End** of string | 'khan$' == Like '%khan' '[ab]$' == Like '%a' or '%b' |
| [abc] | Any character listed between the square brackets | '[abc]'== string contains either a or b or c like '%a%' or '%b%' or '%c%' '**[ab]c'** ==Like '%ac%' or '%bc%' ; all combination '^[ab] == Like 'a%' or 'b%' |
| [^abc] | Any character **not listed** between the square brackets | string does not contains either a or b or c |
| [A-Z] | match any upper case letter. | range A,B,C,…Z [a-zA-Z] ==[abc…zABC…Z] |
| [a-z] | match any lower case letter | range a,b,c,…z |
| [0-9] | match any digit from 0 through to 9 | range 0,1,2,…9 |
| | | multiple pattern; 'ram|khan|sri' | 'ram|khan|sri' ;string containing **ram** **Or** **Khan** **Or** **sri** '^ram|khan$|sri' ;string begin with ram Or ends with Khan or contains sri |
| \* | **Quantifier:** Matches zero or more times | 'man' == Like '%man% |
| + | Previous character must occur **at least once** or more times. |  |
| . | Any **single** character |  |
| ? | preceding character is optional. Match 0 or 1 time |  |
| {n} | Matches exactly n occurrences of the preceding character. |  |
| {n,} | Matches n or more occurrences of the preceding character. |  |
| {n,m} | Matches between n and m occurrences of the preceding element |  |

## Quantifiers

In regular expressions, quantifiers are metacharacters that specify how many times the previous character or group should be matched. Above green colors are quantifiers.

E.g [a-z]\* [a-z]+ [a-z]?