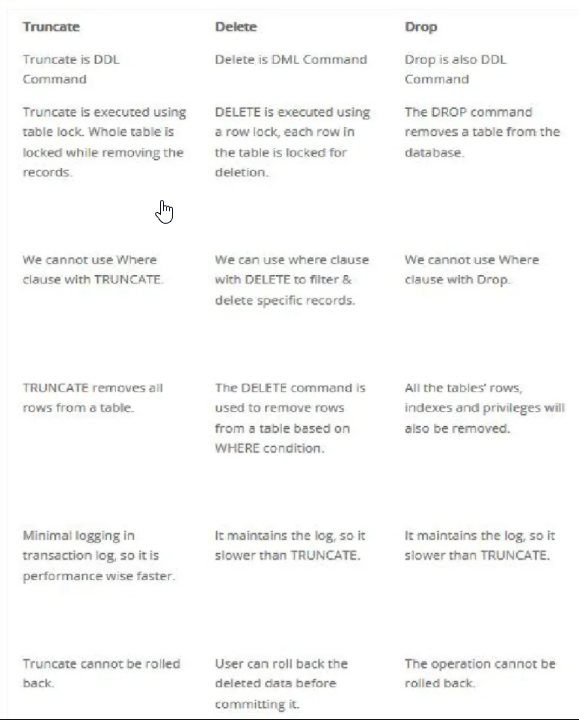
1/19/2021

Raja Umer & Saber Khan

MTBC

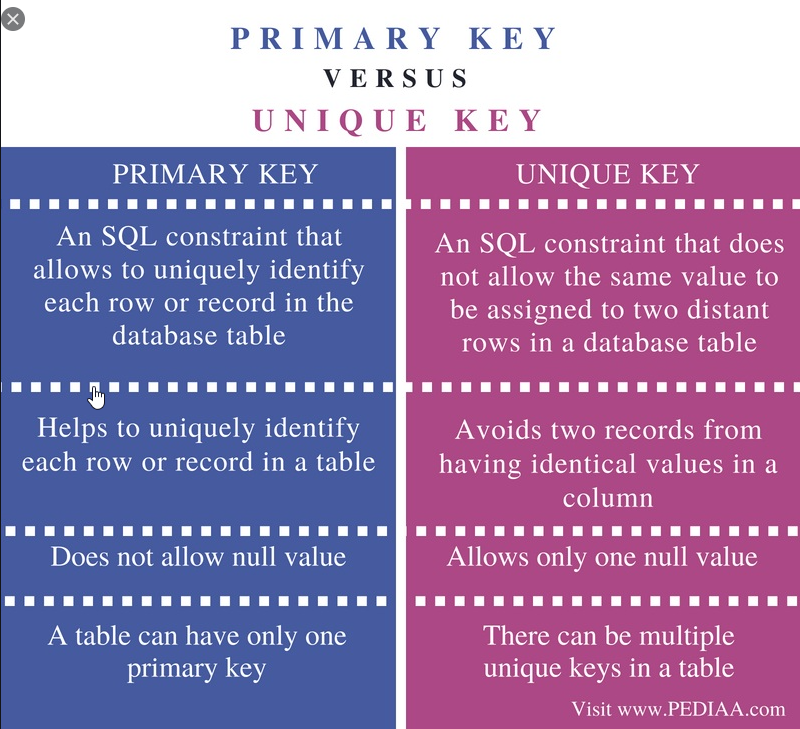
**Database Document**

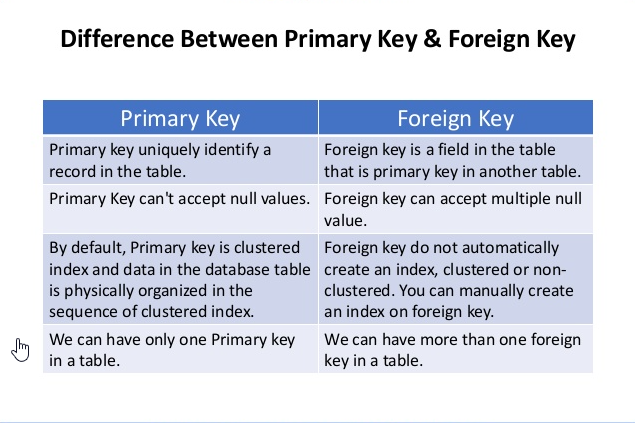
**Difference between drop, truncate and delete (1/11/2020)**



It can be rollback in sql.

But not in oracle





Creating Database, Create Table, Inserting, Deleting,Truncate(Example)

use TestDB7658(for using DB through query)

go

create database persons

sp\_rename 'Persons', 'Personal';  ---to rename table name in sql

Table Creation:

CREATE TABLE Persons (

    PersonID int not null,

    LastName varchar(255),

    FirstName varchar(255),

    Address varchar(255),

    City varchar(255),

    Mobile bigint ,

    DOB date ,

    Email\_address varchar(255)

);

select \* from Persons

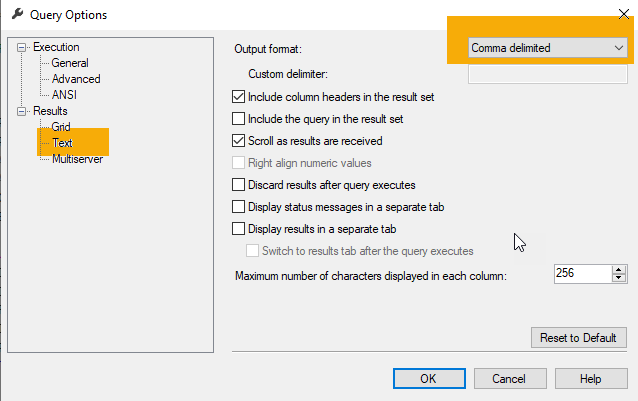
----Msg 208, Level 16, State 1, Line 15

----Invalid object name 'Persons'.

To select all columns headers at once click first write:

select top 0 \* from Persons

then click on query option on top of sql and then select below option and click ok



--PersonID,LastName,FirstName,Address,City,Mobile,DOB,Email\_address

Note: Always use begin Tran and commit\Rollback function before insert, del, update

* Insert Record:

begin Tran

insert into Persons

(PersonID,LastName,FirstName,Address,City,Mobile,DOB,Email\_address)

values ('1','Walli','Nasir','Koraal ISB','ISB',03346721665,'09/07/1984','pitafi59@gmail.com')

rollback

commit

------Msg 515, Level 16, State 2, Line 24

------Cannot insert the value NULL into column 'PersonID', table 'TestDB7658.dbo.Persons'; column does not allow nulls. INSERT fails.

------The statement has been terminated.

------Completion time: 2021-01-11T02:24:57.7401496+05:00

* UPDATE:

begin tran

update Persons

set City='Islamabad'

where PersonID='1'

rollback

commit

DROP TABLE:

drop table Persons

go

TRUNCATE:

truncate table Persons

begin tran

delete from Persons

where PersonID='1'

rollback

commit

begin tran

truncate table Persons

begin tran

drop table Persons

rollback

commit

ALTER:

ALTER TABLE Persons

ADD Father\_name varchar(255);

UPDATE:

begin tran

update Persons

set Last\_degree='BSCS'

where PersonID='1'

rollback

commit

Difference between identity and primary key

Major Difference between Primary and Identity Column

Primary Column:

* Primary Key cannot have duplicate values.
* It creates a clustered index for the Table.
* It can be set for any column type.
* We need to provide the primary column value while inserting in the table.

Identity Column:

* Identity Column can have duplicate value.
* It can only be set for Integer related columns like int, bigint, smallint, tinyint or decimal
* No need to insert values in the identity column. It is inserted automatically based on the seed.

Primary key emphasizing on uniqueness and avoid duplication value for all records on the same column, while identity provides increasing numbers in a column without inserting data. Both features could be on a single column or on difference one.

SET IDENTITY\_INSERT Persons ON---

DBCC CHECKIDENT ('Persons', RESEED, 0)---To set identity()

NOTE: If set identity on\off was not working in one session then open new session and try there its working fine.

Inserting record between two tables:

insert into Persons

select \* from Persons\_bkp\_12012021

------Msg 515, Level 16, State 2, Line 24

------Cannot insert the value NULL into column 'PersonID', table 'TestDB7658.dbo.Persons'; column does not allow nulls. INSERT fails.

------The statement has been terminated.

INTEGRITY CONSTRAINTS

Integrity Constraints

* Integrity constraints are a set of rules. It is used to maintain the quality of information.
* Integrity constraints ensure that the data insertion, updating, and other processes have to be performed in such a way that data integrity is not affected.
* Thus, integrity constraint is used to guard against accidental damage to the database.

## **Types of Integrity Constraint**

### **Domain constraints**

* Domain constraints can be defined as the definition of a valid set of values for an attribute.
* The data type of domain includes string, character, integer, time, date, currency, etc. The value of the attribute must be available in the corresponding domain.

### **Entity integrity constraints**

* The entity integrity constraint states that primary key value can't be null.
* This is because the primary key value is used to identify individual rows in relation(table) and if the primary key has a null value, then we can't identify those rows.
* A table can contain a null value other than the primary key field.

### **Referential Integrity Constraints**

* A referential integrity constraint is specified between two tables.
* In the Referential integrity constraints, if a foreign key in Table 1 refers to the Primary Key of Table 2, then every value of the Foreign Key in Table 1 must be null or be available in Table 2.

### **Key constraints**

* Keys are the entity set that is used to identify an entity within its entity set uniquely.
* An entity set can have multiple keys, but out of which one key will be the primary key. A primary key can contain a unique and null value in the relational table.

Understand SQL constraints

* [NOT NULL](https://www.w3schools.com/sql/sql_notnull.asp) - Ensures that a column cannot have a NULL value
* [UNIQUE](https://www.w3schools.com/sql/sql_unique.asp) - Ensures that all values in a column are different
* [PRIMARY KEY](https://www.w3schools.com/sql/sql_primarykey.asp) - A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table
* [FOREIGN KEY](https://www.w3schools.com/sql/sql_foreignkey.asp) - Uniquely identifies a row/record in another table
* [CHECK](https://www.w3schools.com/sql/sql_check.asp) - Ensures that all values in a column satisfies a specific condition
* [DEFAULT](https://www.w3schools.com/sql/sql_default.asp) - Sets a default value for a column when no value is specified
* [INDEX](https://www.w3schools.com/sql/sql_create_index.asp) - Used to create and retrieve data from the database very quickly

NOTE: Constraints can be drop from table by using

Command: ALTER TABLE (Table name) DROP CONSTRAINT(constraint name)

DBMS Integrity Constraints

## **NOT NULL Constraint**

**NOT NULL** constraint restricts a column from having a NULL value. Once NOT **NULL** constraint is applied to a column, you cannot pass a null value to that column. It enforces a column to contain a proper value.

One important point to note about this constraint is that it cannot be defined at table level.

### **Example using NOT NULL constraint**

CREATE TABLE Student(s\_id int NOT NULL, Name varchar(60), Age int);

## **UNIQUE Constraint**

**UNIQUE** constraint ensures that a field or column will only have unique values. A **UNIQUE** constraint field will not have duplicate data. This constraint can be applied at column level or table level.

### **Using UNIQUE constraint when creating a Table (Table Level)**

Here we have a simple CREATE query to create a table, which will have a column **s\_id** with unique values.

CREATE TABLE Student(s\_id int NOT NULL UNIQUE, Name varchar(60), Age int);

## **Primary Key Constraint**

Primary key constraint uniquely identifies each record in a database. A Primary Key must contain unique value and it must not contain null value. Usually Primary Key is used to index the data inside the table.

### **Using PRIMARY KEY constraint at Table Level**

CREATE table Student (s\_id int PRIMARY KEY, Name varchar(60) NOT NULL, Age int);

The above command will creates a PRIMARY KEY on the s\_id.

### **Using PRIMARY KEY constraint at Column Level**

ALTER table Student ADD PRIMARY KEY (s\_id);

## **Foreign Key Constraint**

FOREIGN KEY is used to relate two tables. FOREIGN KEY constraint is also used to restrict actions that would destroy links between tables.

## **CHECK Constraint**

**CHECK** constraint is used to restrict the value of a column between a range. It performs check on the values, before storing them into the database.

\

# [Difference Between Local and Global Temporary Tables in SQL Server](https://database.guide/difference-between-local-and-global-temporary-tables-in-sql-server/)

## **Naming**

* Local temporary tables’ names start with a single number sign (#). For example, #My Table.
* Local temporary tables also get a system generated numeric suffix appended to the name. This is automatically generated by SQL Server. This allows multiple sessions to create local temporary tables with the same name without causing naming conflicts. However, users can still query the table without having to know the suffix.
* Global temporary tables’ names start with a double number sign (##). For example, ##My Table.
* Global temporary tables are not appended with a suffix like with local temporary tables.

## **Visibility**

* Local temporary tables are visible only in the current session.
* Global temporary tables are visible to all sessions.

## **Lifespan**

* Local temporary tables are automatically dropped at the end of the current session. A local temporary table created in a stored procedure is dropped automatically when the stored procedure is finished. The table can be referenced by any nested stored procedures executed by the stored procedure that created the table. The table cannot be referenced by the process that called the stored procedure that created the table.
* Global temporary tables are automatically dropped when the session that created the table ends and all other tasks have stopped referencing them. The association between a task and a table is maintained only for the life of a single T-SQL statement. Therefore, a global temporary table is dropped at the completion of the last T-SQL statement that was actively referencing the table when the creating session ended.

# GROUP BY and HAVING Clause in SQL

## **Group By Clause**

The GROUP BY Clause is utilized in SQL with the SELECT statement to organize similar data into groups. It combines the multiple records in single or more columns using some functions. Generally, these functions are aggregate functions such as min(),max(),avg(), count(), and sum() to combine into single or multiple columns. It uses the **split-apply-combine** strategy for data analysis.

* In the split phase, It divides the groups with its values.
* In the apply phase, It applies the aggregate function and generates a single value.
* In the combiner phase, It combines the groups with single values into a single value.

**Points to Remember:**

* GROUP BY Clause is utilized with the SELECT statement.
* GROUP BY aggregates the results on the basis of selected column: COUNT, MAX, MIN, SUM, AVG, etc.
* GROUP BY returns only one result per group of data.
* GROUP BY Clause always follows the WHERE Clause.
* GROUP BY Clause always precedes the ORDER BY

## **Having Clause**

HAVING Clause utilized in SQL as a conditional Clause with GROUP BY Clause. This conditional clause returns rows where aggregate function results matched with given conditions only. It added in the SQL because WHERE Clause cannot be combined with aggregate results, so it has a different purpose. The primary purpose of the WHERE Clause is to deal with non-aggregated or individual records.

* HAVING Clause always utilized in combination with GROUP BY Clause.
* HAVING Clause restricts the data on the group records rather than individual records.
* WHERE and HAVING can be used in a single query.

# What is the difference between UNION and UNION ALL

The main difference between UNION and UNION ALL is that:

* **UNION:** only keeps *unique* records
* **UNION ALL:** keeps all records, including *duplicates*

### **UNION:**

UNION removes any duplicate records. UNION first performs a sorting operation and eliminates of the records that are duplicated across all columns before finally returning the combined data set.

### **UNION ALL**

If we were to now perform the UNION ALL on the same data set, the query would skip the deduplication step and return the results shown.

### **OFF SET FETCH:**

OFFSET and FETCH Clause are used in conjunction with SELECT and ORDER BY clause to provide a means to retrieve a range of records.

**OFFSET**

The OFFSET argument is used to identify the starting point to return rows from a result set. Basically, it exclude the first set of records.  
**Note:**

* OFFSET can only be used with ORDER BY clause. It cannot be used on its own.
* OFFSET value must be greater than or equal to zero. It cannot be negative, else return error.

Syntax:

SELECT column name(s)

FROM table name

WHERE condition

ORDER BY column name

OFFSET rows\_to\_skip ROWS;

### **SUB-QUERY:**

A Subquery or Inner query or a Nested query is a query within another SQL query and embedded within the WHERE clause.

A subquery is used to return data that will be used in the main query as a condition to further restrict the data to be retrieved.

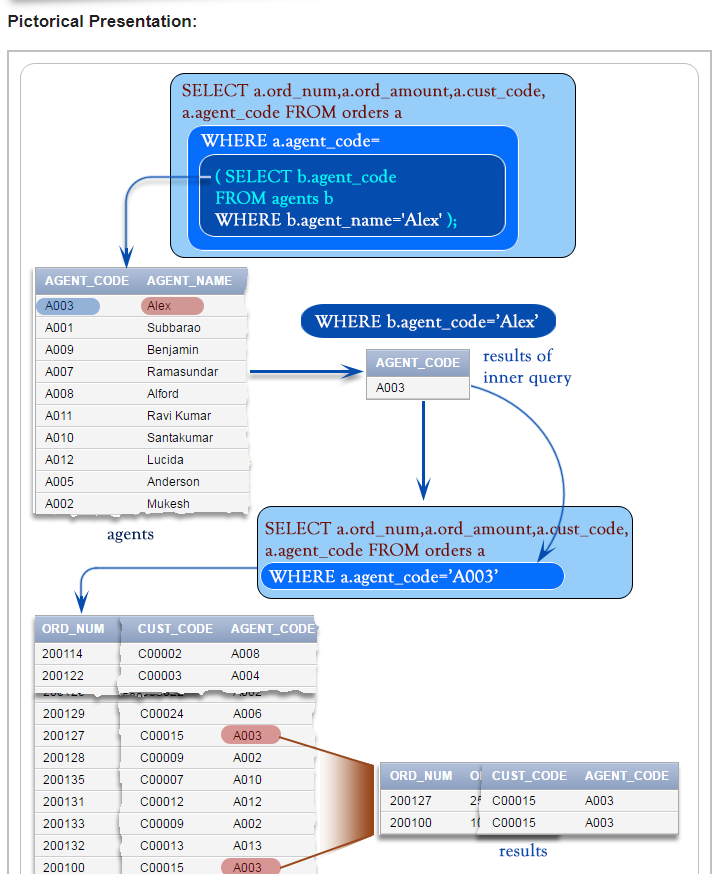
Subqueries can be used with the SELECT, INSERT, UPDATE, and DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN, etc.

### **There are a few rules that subqueries must follow −**

* Subqueries must be enclosed within parentheses.
* A subquery can have only one column in the SELECT clause, unless multiple columns are in the main query for the subquery to compare its selected columns.
* An ORDER BY command cannot be used in a subquery, although the main query can use an ORDER BY. The GROUP BY command can be used to perform the same function as the ORDER BY in a subquery.
* Subqueries that return more than one row can only be used with multiple value operators such as the iN operator.
* The SELECT list cannot include any references to values that evaluate to a BLOB, ARRAY, CLOB, or NCLOB.
* A subquery cannot be immediately enclosed in a set function.
* The BETWEEN operator cannot be used with a subquery. However, the BETWEEN operator can be used within the subquery.
* You can use upto 32 sub queries in one script.(In SQL server)

### **Correlated Subqueries**

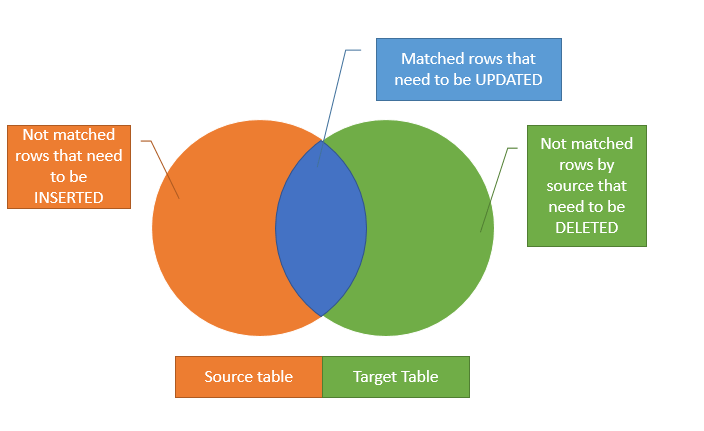
SQL Correlated Subqueries are used to select data from a table referenced in the outer query. The subquery is known as a correlated because the subquery is related to the outer query. In this type of queries, a table alias (also called a correlation name) must be used to specify which table reference is to be used.



### **SQL Server MERGE Statement**

Suppose, you have two table called source and target tables, and you need to update the target table based on the values matched from the source table. There are three cases:

1. The source table has some rows that do not exist in the target table. In this case, you need to [insert](https://www.sqlservertutorial.net/sql-server-basics/sql-server-insert/) rows that are in the source table into the target table.
2. The target table has some rows that do not exist in the source table. In this case, you need to [delete](https://www.sqlservertutorial.net/sql-server-basics/sql-server-delete/) rows from the target table.
3. The source table has some rows with the same keys as the rows in the target table. However, these rows have different values in the non-key columns. In this case, you need to [update](https://www.sqlservertutorial.net/sql-server-basics/sql-server-update/) the rows in the target table with the values coming from the source table.



SYNTAX:

**MERGE** target\_table **USING** source\_table

**ON** merge\_condition

**WHEN** **MATCHED**

**THEN** update\_statement

**WHEN** **NOT** **MATCHED**

**THEN** insert\_statement

**WHEN** **NOT** **MATCHED** **BY** **SOURCE**

**THEN** **DELETE**;

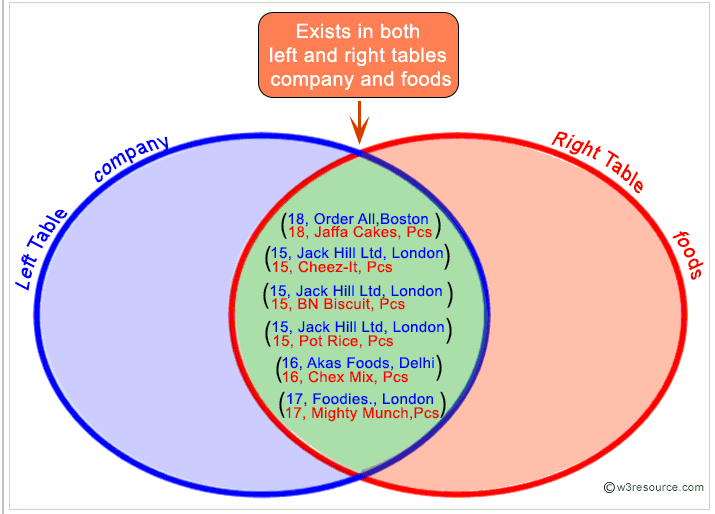
# ****JOIN:****

# JOIN returns all rows from tables where the key record of one table is equal to the key records of another table.

# **SQL INNER JOIN:**

# The INNER JOIN selects all rows from both participating tables as long as there is a match between the columns. An SQL INNER JOIN is same as JOIN clause, combining rows from two or more tables.

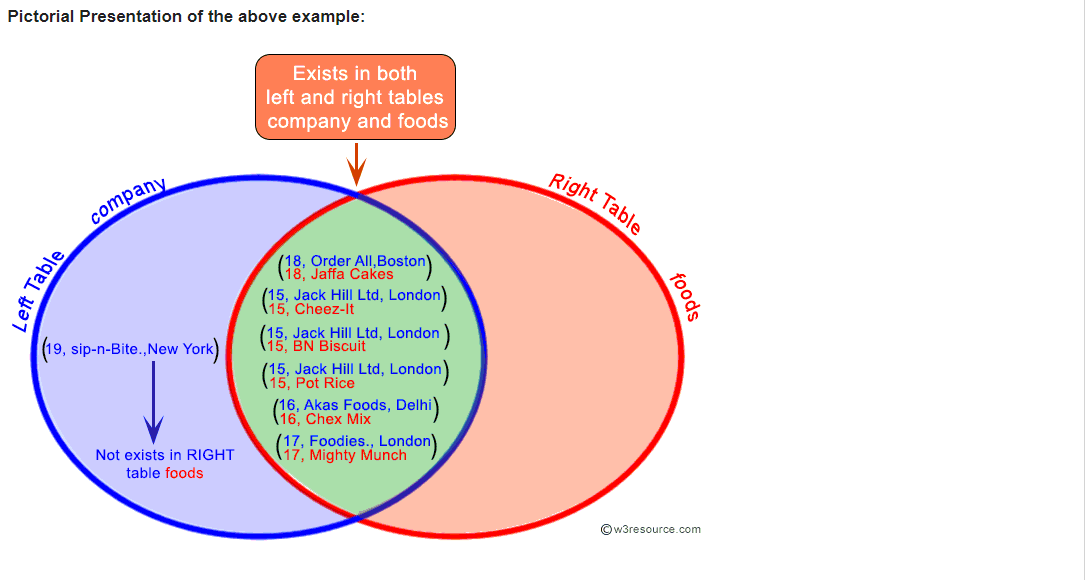
# **Pictorial Presentation:**



# ****LEFT JOIN:****

# The SQL LEFT JOIN (specified with the keywords LEFT JOIN and ON) joins two tables and fetches all matching rows of two tables for which the SQL-expression is true, plus rows from the first table (left ) that do not match any row in the second table.

LEFT JOIN Diagram:



# **RIGHT JOIN**

# The SQL RIGHT JOIN, joins two tables and fetches rows based on a condition, which is matching in both the tables (before and after the JOIN clause mentioned in the syntax below) , and the unmatched rows will also be available from the table written after the JOIN clause ( mentioned in the syntax below ).

**Syntax:**

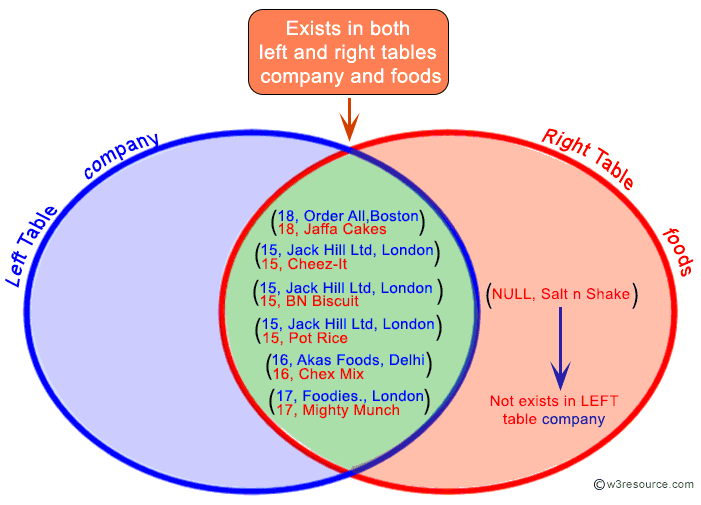
SELECT \*

FROM table1

RIGHT [ OUTER ] JOIN table2

ON table1.column\_name=table2.column\_name;

Pictorial Presentation:



# ****Full Outer Join:****

# In SQL the FULL OUTER JOIN combines the results of both [left](https://www.w3resource.com/sql/joins/perform-a-left-join.php)and [right](https://www.w3resource.com/sql/joins/perform-a-right-join.php) outer joins and returns all (matched or unmatched) rows from the tables on both sides of the join clause.

**Syntax:**

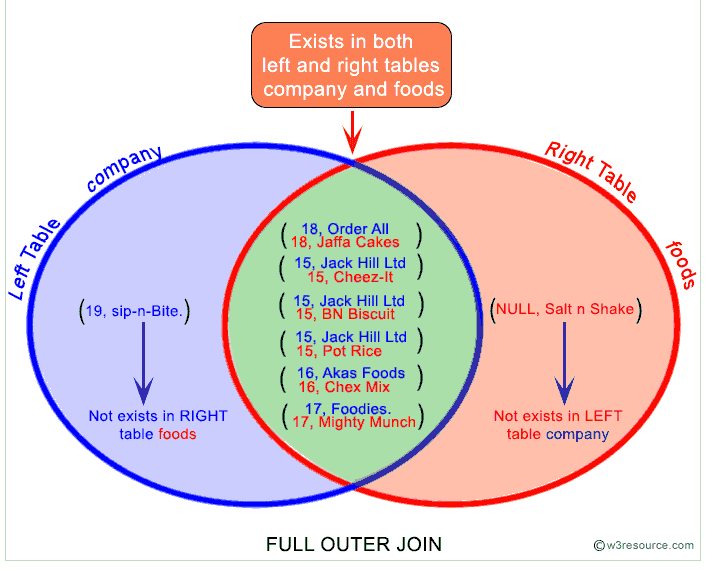
SELECT \*

FROM table1

FULL OUTER JOIN table2

ON table1.column\_name=table2.column\_name;

Pictorial Presentation:



**Self Join:**

# A self join is a join in which a table is joined with itself (which is also called Unary relationships), especially when the table has a FOREIGN KEY which references its own PRIMARY KEY. To join a table itself means that each row of the table is combined with itself and with every other row of the table.

# Syntax:

SELECT a.column\_name, b.column\_name...

FROM table1 a, table1 b

WHERE a.common\_field = b.common\_field;

# **SQL Cross Join:**

The SQL CROSS JOIN produces a result set which is the number of rows in the first table multiplied by the number of rows in the second table if no WHERE clause is used along with CROSS JOIN. This kind of result is called as Cartesian Product.

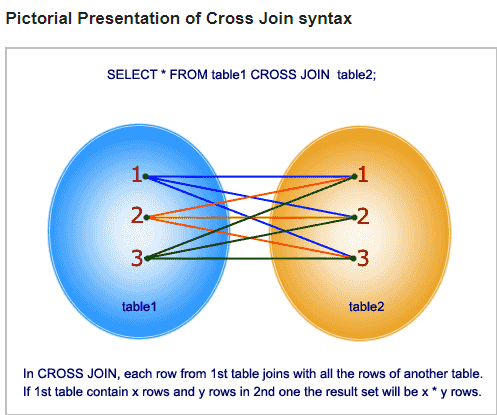
If WHERE clause is used with CROSS JOIN, it functions like an INNER JOIN.

**Syntax:**

SELECT \*

FROM table1

CROSS JOIN table2;



# **SQL Equi Join:**

SQL EQUI JOIN performs a JOIN against equality or matching column(s) values of the associated tables. An equal sign (=) is used as comparison operator in the where clause to refer equality.

You may also perform EQUI JOIN by using JOIN keyword followed by ON keyword and then specifying names of the columns along with their associated tables to check equality.

**Syntax:**

SELECT column\_list

FROM table1, table2....

WHERE table1.column\_name =

table2.column\_name;

or

SELECT \*

FROM table1

JOIN table2

[ON (join\_condition)]

# 

The difference here is for the Inner join i can have a different condition , such as “=” , <> , > , < etc, but same is not true for equi join.

Notes :

1. Inner join can have equality (=) and other operators (like <,>,<>) in the join condition.
2. Equi join only have equality (=) operator in the join condition.
3. Equi join can be an Inner join, Left Outer join, Right Outer join
4. The USING clause is not supported by SQL Server and Sybase. This clause is supported by Oracle and MySQL.

# ****NON EQUI JOIN****

The SQL NON EQUI JOIN uses comparison operator instead of the equal sign like **>, <, >=, <=**along with conditions.

**Syntax:**

SELECT \*

FROM table\_name1, table\_name2

WHERE table\_name1.column [> | < | >= | <= ] table\_name2.column;

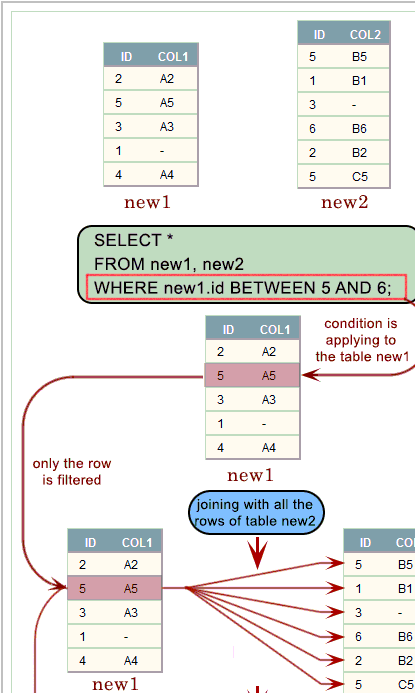
**Theta Join:**

**THETA JOIN** allows you to merge two tables based on the condition represented by **theta**. **Theta joins** work for all comparison operators. It is denoted by symbol θ. The general case of **JOIN** operation is called a **Theta join**

**THETA JOIN** allows you to merge two tables based on the condition represented by theta. Theta joins work for all comparison operators. It is denoted by symbol **θ**. The general case of JOIN operation is called a Theta join.

Syntax:

A ⋈θ B



# **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.

**Natural Join: Guidelines**

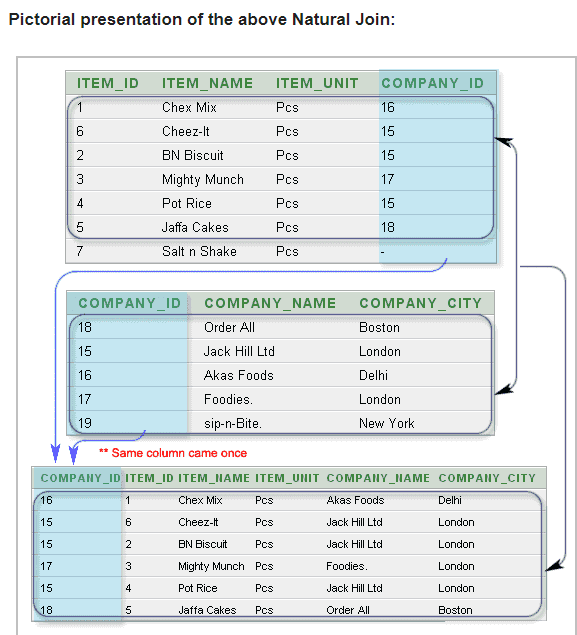
- The associated tables have one or more pairs of identically named columns.  
- The columns must be the same data type.  
- Don’t use ON clause in a natural join.

**Syntax:**

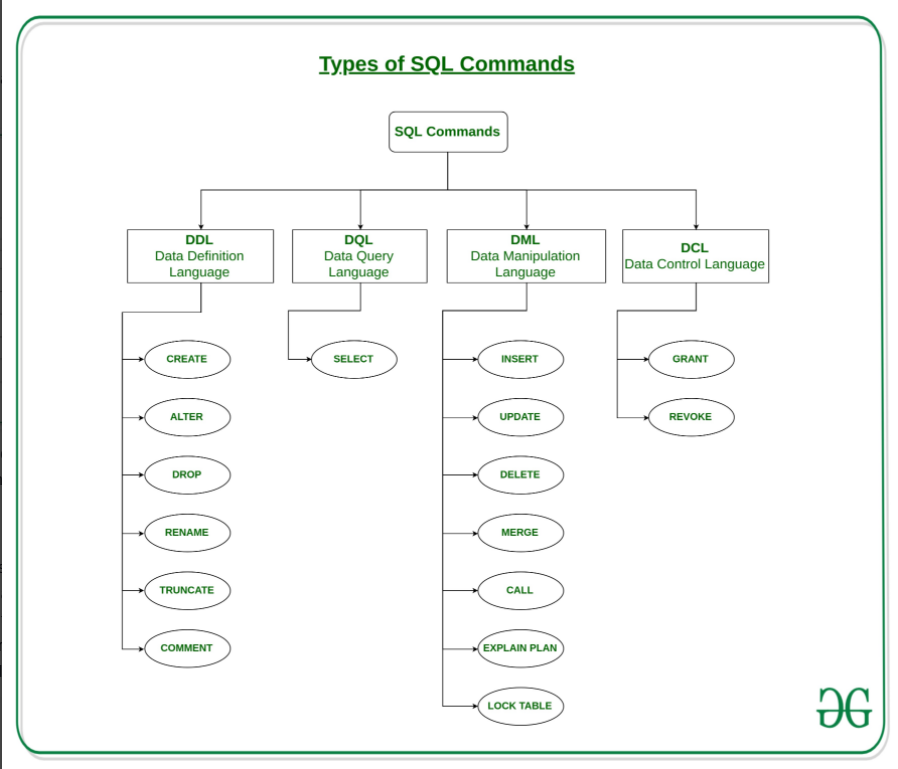
SELECT \*

FROM table1

NATURAL JOIN table2;



**SQL | DDL, DQL, DML, DCL and TCL Commands:**



**CROSS APPLY and OUTER APPLY**

* The CROSS APPLY operator returns only those rows from the left table expression (in its final output) if it matches with the right table expression. In other words, the right table expression returns rows for the left table expression match only.
* The OUTER APPLY operator returns all the rows from the left table expression irrespective of its match with the right table expression. For those rows for which there are no corresponding matches in the right table expression, it contains NULL values in columns of the right table expression.
* So you might conclude, the CROSS APPLY is equivalent to an INNER JOIN (or to be more precise its like a CROSS JOIN with a correlated sub-query) with an implicit join condition of 1=1 whereas the OUTER APPLY is equivalent to a LEFT OUTER JOIN.

**WHY:**

when do you use the APPLY operator? Although the same can be achieved with a [normal JOIN](https://www.mssqltips.com/sqlservertip/1667/sql-server-join-example/), the need of APPLY arises if you have a table-valued expression on the right part and in some cases the use of the APPLY operator boosts [performance](https://www.mssqltips.com/sql-server-tip-category/9/performance-tuning/) of your query.

# EXISTS Condition:

* The SQL Server (Transact-SQL) EXISTS condition is used in combination with a subquery and is considered to be met if the subquery returns at least one row. It can be used in a SELECT, INSERT, UPDATE, or DELETE statement.

### **Parameters or Arguments**

The subquery is a SELECT statement. If the subquery returns at least one record in its result set, the EXISTS clause will evaluate to true and the EXISTS condition will be met. If the subquery does not return any records, the EXISTS clause will evaluate to false and the EXISTS condition will not be met.

Note

* SQL statements that use the EXISTS condition are very inefficient since the sub-query is RE-RUN for EVERY row in the outer query's table. There are more efficient ways to write most queries, that do not use the EXISTS condition.

## Example

SELECT \*

FROM employees

WHERE EXISTS (SELECT \*

FROM contacts

WHERE employees.last\_name = contacts.last\_name

AND employees.first\_name = contacts.first\_name);

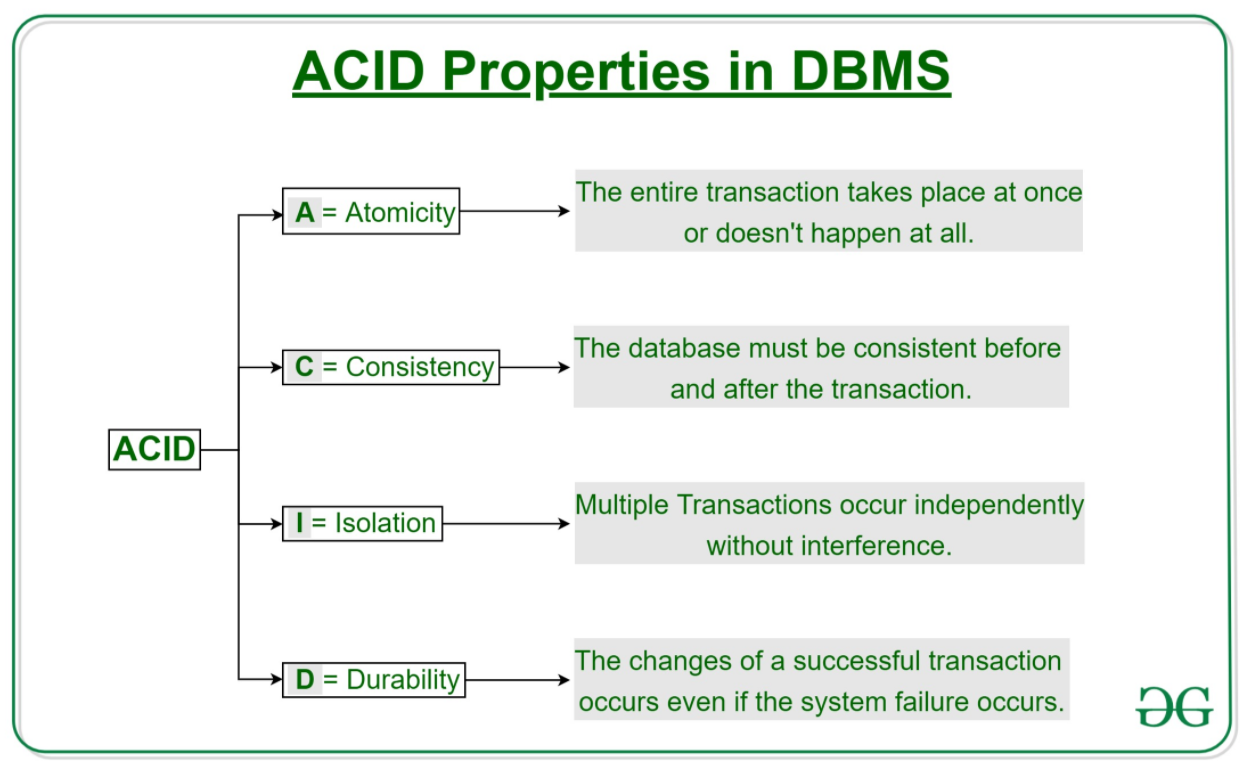
**Result:**

This SQL Server EXISTS condition example will return all records from the employees table where there is at least one record in the contacts table with a matching last\_name and first\_name.

NOT EXISTS

The SQL Server EXISTS condition can also be combined with the[NOT operator](https://www.techonthenet.com/sql_server/not.php).

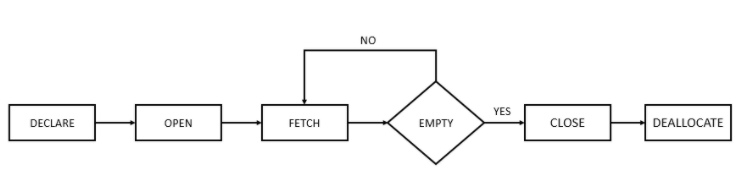
**ACID Properties in DBMS:**



* 1. **Atomicity:** The term atomicity defines that the data remains atomic. It means if any operation is performed on the data, either it should be performed or executed completely or should not be executed at all. It further means that the operation should not break in between or execute partially. In the case of executing operations on the transaction, the operation should be completely executed and not partially.
  2. **Consistency:** The word **consistency** means that the value should remain preserved always. In [DBMS](https://www.javatpoint.com/dbms-tutorial), the integrity of the data should be maintained, which means if a change in the database is made, it should remain preserved always. In the case of transactions, the integrity of the data is very essential so that the database remains consistent before and after the transaction. The data should always be correct.
  3. **Isolation:** The term 'isolation' means separation. In DBMS, Isolation is the property of a database where no data should affect the other one and may occur concurrently. In short, the operation on one database should begin when the operation on the first database gets complete. It means if two operations are being performed on two different databases, they may not affect the value of one another. In the case of transactions, when two or more transactions occur simultaneously, the consistency should remain maintained. Any changes that occur in any particular transaction will not be seen by other transactions until the change is not committed in the memory.
  4. **Durability:** Durability ensures the permanency of something. In DBMS, the term durability ensures that the data after the successful execution of the operation becomes permanent in the database. The durability of the data should be so perfect that even if the system fails or leads to a crash, the database still survives. However, if gets lost, it becomes the responsibility of the recovery manager for ensuring the durability of the database. For committing the values, the COMMIT command must be used every time we make changes.

**What is Cursor in SQL**

**Cursor** is a Temporary Memory or Temporary Work Space. It is Allocated by Database Server at the Time of Performing DML operations on Table by User. Cursors are used to store Database Tables. There are 2 types of Cursors: Implicit Cursors, and Explicit Cursors. These are explained as following below.



1. **Implicit Cursors:**  
   Implicit Cursors are also known as Default Cursors of SQL SERVER. These Cursors are allocated by SQL SERVER when the user performs DML operations.
2. **Explicit Cursors :**  
   Explicit Cursors are Created by Users whenever the user requires them. Explicit Cursors are used for Fetching data from Table in Row-By-Row Manner.
3. **Note: cursor check and matched data only row wise.**

**How to create Explicit Cursor:**

1. **Declare Cursor Object.**  
   **Syntax :** DECLARE cursor name CURSOR FOR SELECT \* FROM table name

DECLARE s1 CURSOR FOR SELECT \* FROM stud Details

1. **Open Cursor Connection.**  
   **Syntax :** OPEN cursor connection

OPEN s1

1. **Fetch Data from cursor.**  
   There are total 6 methods to access data from cursor. They are as follows :  
   **FIRST** is used to fetch only the first row from cursor table.  
   **LAST** is used to fetch only last row from cursor table.  
   **NEXT** is used to fetch data in forward direction from cursor table.  
   **PRIOR** is used to fetch data in backward direction from cursor table.  
   **ABSOLUTE n** is used to fetch the exact nth row from cursor table.  
   **RELATIVE n** is used to fetch the data in incremental way as well as decremental way.  
   **Syntax :** FETCH NEXT/FIRST/LAST/PRIOR/ABSOLUTE n/RELATIVE n FROM cursor\_name

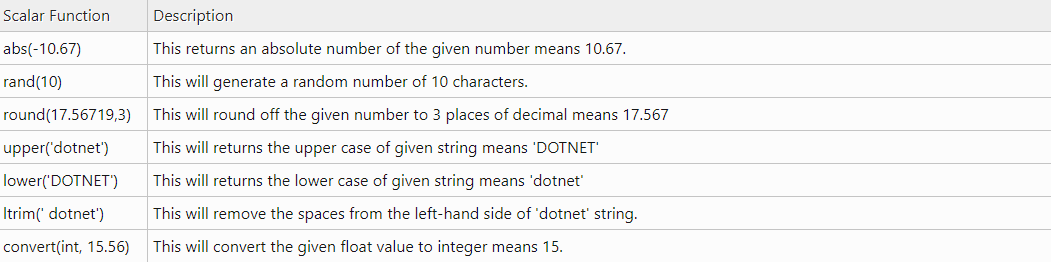
## **Types of Function**

## **System Defined Function**

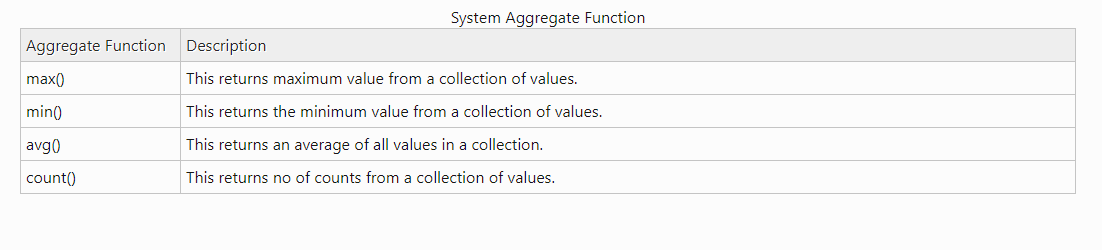
These functions are defined by [**SQL Server**](https://www.dotnettricks.com/learn/sqlserver/introduction-to-sql-server) for a different purpose. We have two types of system defined function in SQL Server

## **Scalar Function**

Scalar functions operate on a single value and return a single value. Below is the list of some useful **SQL Server Scalar functions.**



## **Aggregate Function**



## **User Defined Function**

These functions are created by the user in the system database or in a user-defined database. We three types of user-defined functions.

--Create a table

CREATE TABLE Employee

(

EmpID int PRIMARY KEY,

FirstName varchar(50) NULL,

LastName varchar(50) NULL,

Salary int NULL,

Address varchar(100) NULL,

)

--Insert Data

Insert into Employee(EmpID,FirstName,LastName,Salary,Address) Values(1,'Mohan','Chauahn',22000,'Delhi');

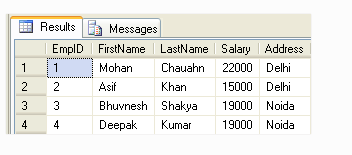
Insert into Employee(EmpID,FirstName,LastName,Salary,Address) Values(2,'Asif','Khan',15000,'Delhi');

Insert into Employee(EmpID,FirstName,LastName,Salary,Address) Values(3,'Bhuvnesh','Shakya',19000,'Noida');

Insert into Employee(EmpID,FirstName,LastName,Salary,Address) Values(4,'Deepak','Kumar',19000,'Noida');

--See created table

Select \* from Employee



--Create function to get emp full name

Create function fnGetEmpFullName

(

@FirstName varchar(50),

@LastName varchar(50)

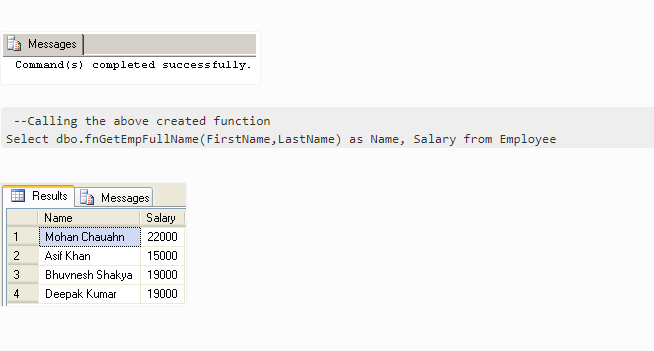
)

returns varchar(101)

As

Begin return (Select @FirstName + ' '+ @LastName);

end



## **Inline Table-Valued Function**

The user-defined inline table-valued function returns a table variable as a result of actions performed by the function. The value of the table variable should be derived from a single SELECT statement.

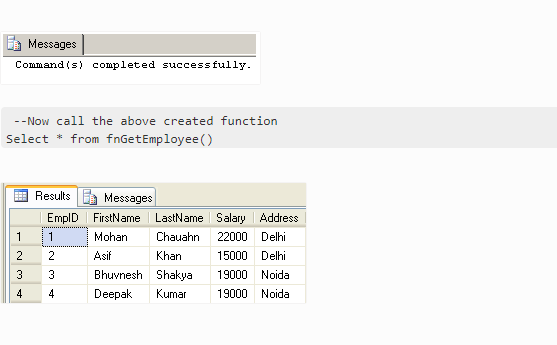
--Create function to get employees

Create function fnGetEmployee()

returns Table

As

return (Select \* from Employee)



## **Multi-Statement Table-Valued Function**

A user-defined multi-statement table-valued function returns a table variable as a result of actions performed by the function. In this, a table variable must be explicitly declared and defined whose value can be derived from multiple **SQL statements.**

--Create function for EmpID,FirstName and Salary of Employee

Create function fnGetMulEmployee()

returns @Emp Table

(

EmpID int,

FirstName varchar(50),

Salary int

)

As

begin

Insert into @Emp Select e.EmpID,e.FirstName,e.Salary from Employee e;

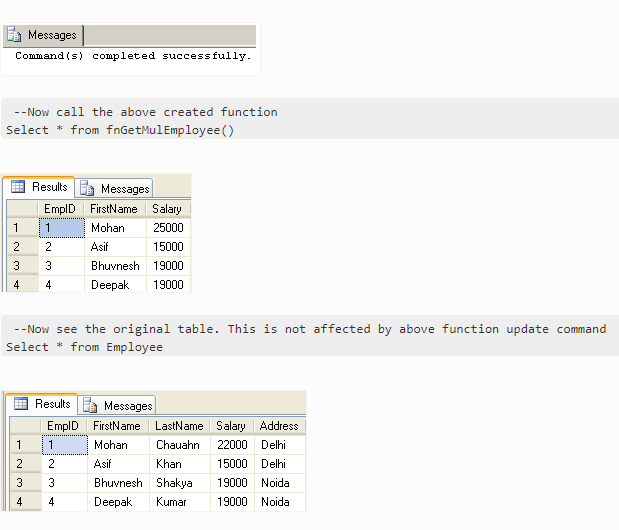
--Now update salary of first employee

update @Emp set Salary=25000 where EmpID=1;

--It will update only in @Emp table not in Original Employee table

return

end



#### **Note**

1. Unlike Stored Procedure, Function returns an only single value.
2. Unlike Stored Procedure, Function accepts only input parameters.
3. Unlike Stored Procedure, Function is not used to Insert, Update, Delete data in a database table(s).
4. Like Stored Procedure, Function can be nested up to 32 levels.
5. User Defined Function can have upto 1023 input parameters while a Stored Procedure can have upto 2100 input parameters.
6. User Defined Function can't return XML Data Type.
7. User Defined Function doesn't support Exception handling.
8. User Defined Function can call only Extended Stored Procedure.
9. User Defined Function doesn't support set options like set ROWCOUNT etc.

## **Stored procedure**

## **Benefits of using a stored procedure**

**It can be easily modified**: We can easily modify the code inside the stored procedure without the need to restart or deploying the application. For example, If the T-SQL queries are written in the application and if we need to change the logic, we must change the code in the application and re-deploy it. SQL Server Stored procedures eliminate such challenges by storing the code in the database. so, when we want to change the logic inside the procedure we can just do it by simple ALTER PROCEDURE statement.

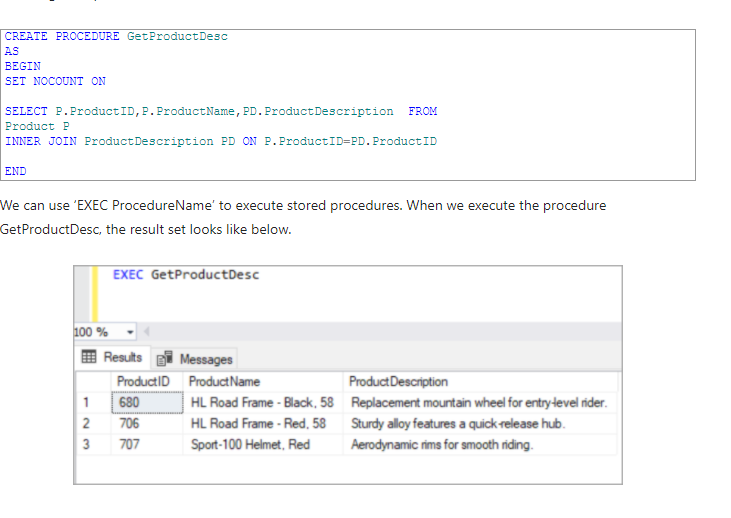
**Reduced network traffic:** When we use stored procedures instead of writing T-SQL queries at the application level, only the procedure name is passed over the network instead of the whole T-SQL code.

**Reusable:**Stored procedures can be executed by multiple users or multiple client applications without the need of writing the code again.

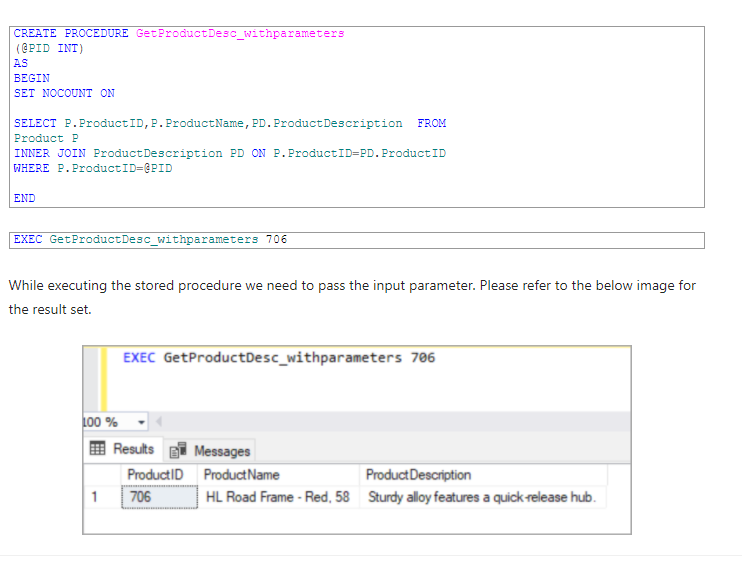
**Security:**Stored procedures reduce the threat by eliminating direct access to the tables. we can also encrypt the stored procedures while creating them so that source code inside the stored procedure is not visible. Use third-party tools like [ApexSQL Decrypt](https://www.apexsql.com/sql-tools-decrypt.aspx" \t "_blank) to decrypt the encrypted stored procedures.

**Performance:**The SQL Server stored procedure when executed for the first time creates a plan and stores it in the buffer pool so that the plan can be reused when it executes next time.

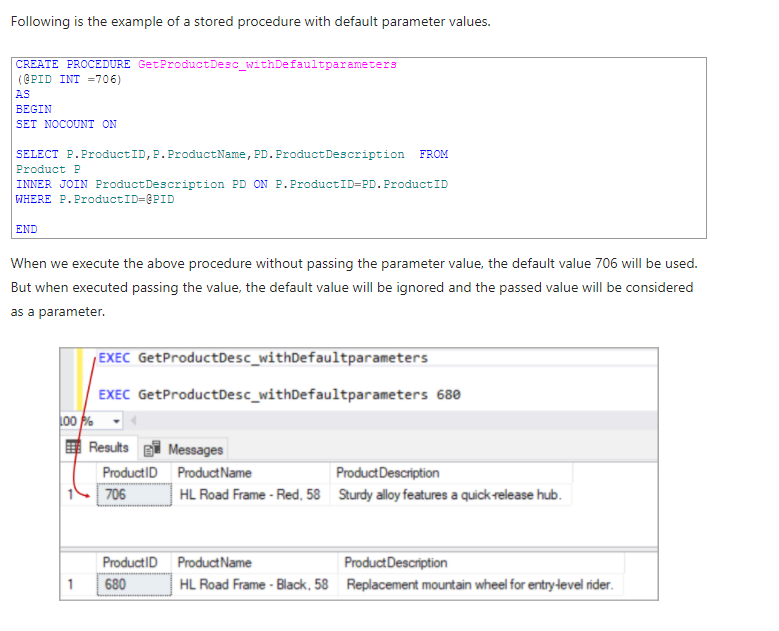
## **Creating a simple stored procedure**



## **Creating a stored procedure with parameters**

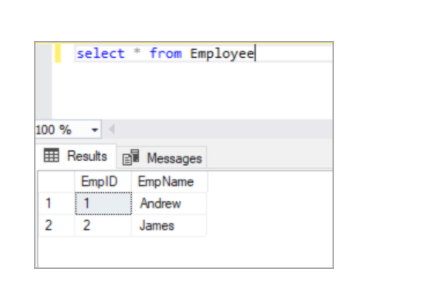


## **Creating a stored procedure with default parameters values**

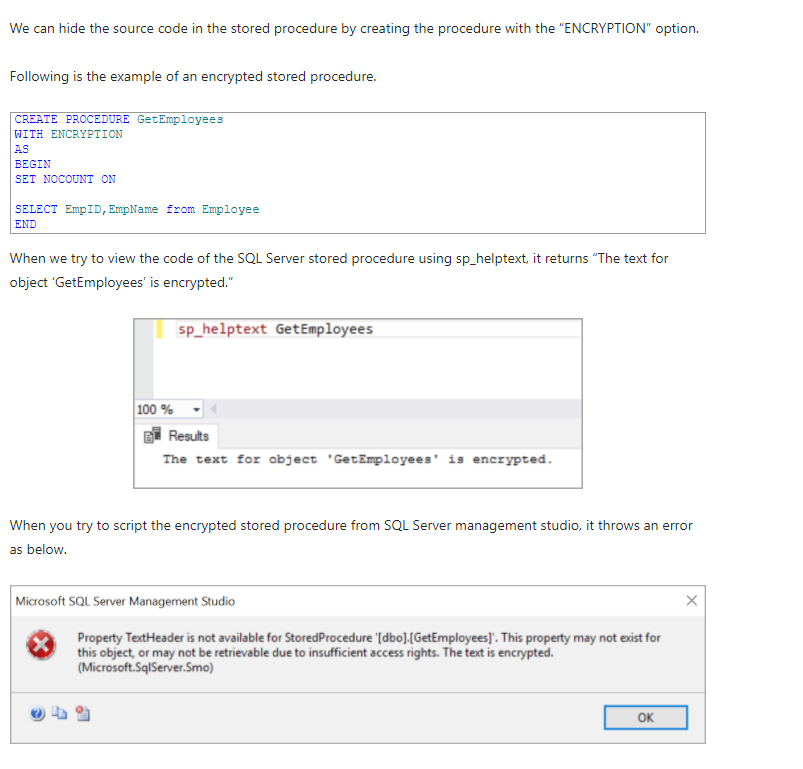


## **Creating a stored procedure with an output parameter**





## **Creating an encrypted stored procedure**



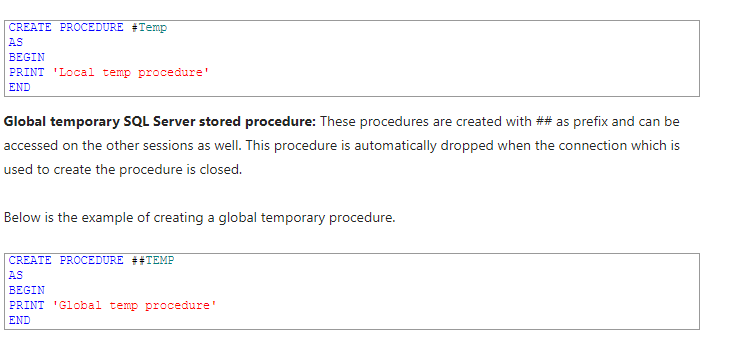
## **Creating a temporary procedure**

Like the temporary table, we can create temporary procedures as well. There are two types of temporary procedures, one is a local temporary stored procedure and another one is a global temporary procedure.

These procedures are created in the **tempdb** database.

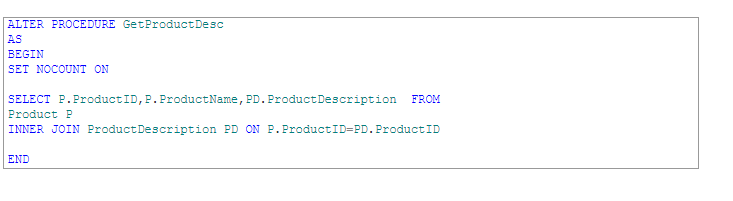
**Local temporary SQL Server stored procedures**: These are created with # as prefix and can be accessed only in the session where it created. This procedure is automatically dropped when the connection is closed.

Following is the example of creating a local temporary procedure.



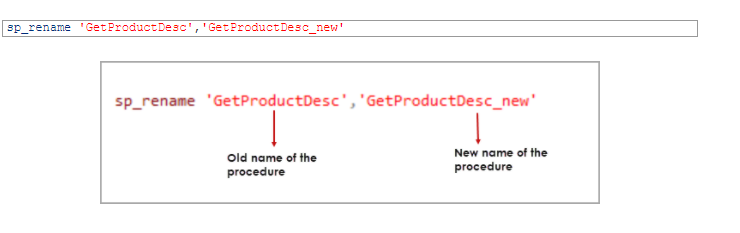
## **Modifying the stored procedure**

Use the **ALTER PROCEDURE** statement to modify the existing stored procedure. Following is the example of modifying the existing procedure.



## **Renaming the stored procedure**

To rename a stored procedure using T-SQL, use system stored procedure sp\_rename. Following is the example that renames the procedure “GetProductDesc” to a new name “GetProductDesc\_new”.



**Basic Differences between Stored Procedure and Function in SQL Server**

1. The function must return a value but in**Stored Procedure** it is optional. Even a procedure can return zero or n values.
2. Functions can have only input parameters for it whereas Procedures can have input or output parameters.
3. Functions can be called from Procedure whereas Procedures cannot be called from a Function.

## **Advance Differences between Stored Procedure and Function in SQL Server**

1. The procedure allows SELECT as well as DML(INSERT/UPDATE/DELETE) statement in it whereas Function allows only SELECT statement in it.
2. Procedures cannot be utilized in a SELECT statement whereas Function can be embedded in a SELECT statement.
3. Stored Procedures cannot be used in the [SQL](https://en.wikipedia.org/wiki/SQL) statements anywhere in the WHERE/HAVING/SELECT section whereas Function can be.
4. Functions that return tables can be treated as another rowset. This can be used in JOINs with other tables.
5. Inline Function can be thought of as views that take parameters and can be used in JOINs and other Rowset operations.
6. An exception can be handled by try-catch block in a Procedure whereas try-catch block cannot be used in a Function.
7. We can use Transactions in Procedure whereas we can't use Transactions in Function.

## **Difference Between Clustered and Non-Clustered Index**

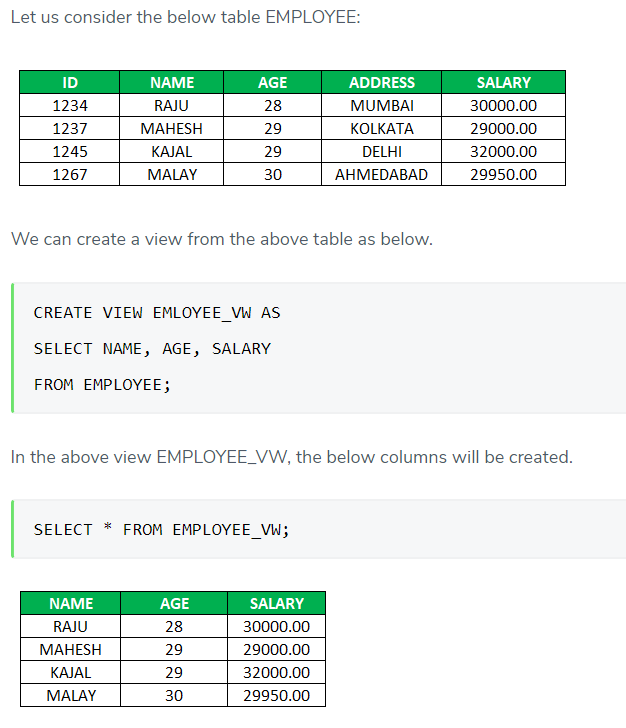
|  |  |
| --- | --- |
| **Clustered Index** | **Non-Clustered Index** |
| It physically sorts the rows of a table based on the primary key or on a column that is unique and not null (generally we use primary key). | This is an index structure that is separate from the actual table which sorts one or more selected columns. Think of it as another table with few columns sorted together. For example, if you have a phonebook table where it is sorted by the name of a person and you want to know the count of people living in a specific country. What you’ll do? You’ll make another table with columns ‘country’ and ‘count\_of\_people’ which is sorted by the country name. Now finding the number of people in a given country will be much faster otherwise you will have to do the full table scan to get the answer. This is a non-clustered index. |
| Querying data is fast. A typical use case can be where there are range-based queries on the primary key. | Querying data is faster when there is no index in a table. A scenario where it can be used is when you want to find no. of people from a particular country in the phonebook. Even if the phonebook is sorted by name, you would like to have some kind of mapping of country with no. of people living there for a faster answer to such queries. |
| There can only be one clustered index per table. | There can be many non-clustered indexes per table. |
| It doesn’t need extra disk space. | It requires extra space to store those indexes. |
| It is faster than the non-clustered index. | It is slower than the clustered index in terms of SELECT queries. |
| Updation and Insertion are slow as the sorted order has to be maintained (can be faster when insertion always happens at the last, e.g.: Index on ID col). | Updation and Insertion are slow as the sorted order has to be maintained. |

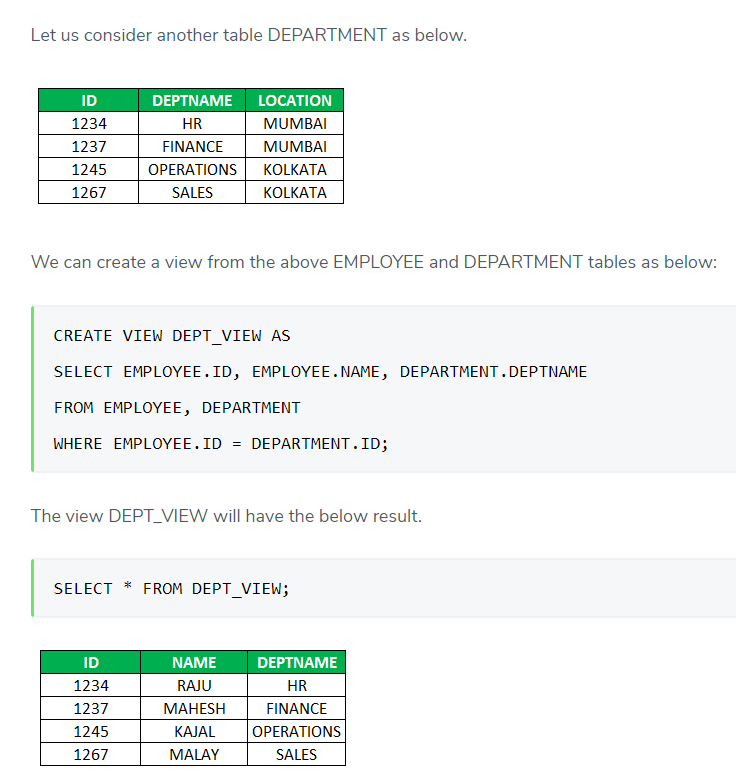
## **Introduction to SQL Views**

Views in SQL are defined as those result sets that work as a stored query on data, and acts as a pre-established query command which is stored by the SQL server in the database dictionary. A view representing a subset of the data present in a table, and performing functions such as joining and simplification of multiple tables into a single table, data aggregation, handling data complexity, providing security etc., The most important aspect being that they taking very minimal memory as a storage space.

### Syntax to Create View

CREATE VIEW VIEW\_NAME AS  
SELECT column1, column2, column3.......  
FROM table\_name WHERE [condition];





### Different view Operations in SQL Views

### Update in View

Though a view can be updated, we need to keep a few conditions in the notice. Such as, while updating a view the select statement should not contain a DISTINCT keyword, set functions, order by clause, Group By or Having, sub-queries, etc. Also, the FROM clause should not contain multiple tables. In addition to the above, the view should have NOT NULL values if it needs to be updated. So when we want to update the view EMPLOYEE\_VW keeping the above points in focus, the table EMPLOYEE will be updated.

CREATE OR REPLACE VIEW statement is used to add or remove fields from a view.

**SYNTAX for Update**

CREATE OR REPLACE VIEW view\_name AS  
SELECT column1, column2....  
FROM table\_name  
WHERE [condition];

### INSERTING into VIEW

Syntax to insert into a view

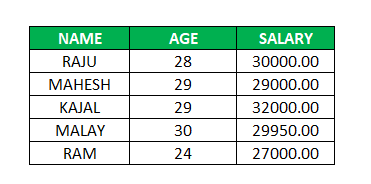
INSERT INTO view\_name(column1, column 2, column3,....) VALUES(value1, value2, value3,...);

A row can be inserted to view EMPLOYEE\_VW by using insert into a statement as below:

INSERT INTO EMPLOYEE\_VW (NAME, AGE, SALARY) VALUES(‘RAM’, ‘24’, ‘27000.00’);

After insertion, we can see the result by the below select query.

SELECT \* FROM EMPLOYEE\_VW;



### DELETING FROM A VIEW

Deleting a row from a view deletes the row from the table on which the view was created.

The syntax for deleting from a view

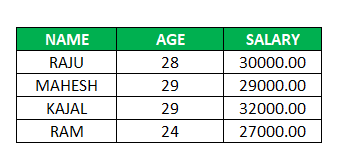
DELETE FROM view\_name WHERE [condition];

We can delete the row from view as below:

DELETE FROM EMPLOYEE\_VW WHERE NAME = ‘MALAY’;

After deletion, the result can be displayed by the below query.

SELECT \* FROM EMPLOYEE\_VW;



### DROPPING A VIEW

The views can be dropped by using the below syntax:

DROP VIEW view\_name;

If we want to delete the view EMPLOYEE\_VW, it can be deleted as below:

DROP VIEW EMPLOYEE\_VW;

### Advantages and Disadvantages of SQL Views

Below are some pros and cons as follows:

#### Advantages

Below are the advantages of using views:

* If we need to maintain any sensitive information by providing limited access to the users, views are used for that purpose. Views are used to only display the required data to the users by keeping sensitive data safe.
* As a database view is associated with many tables upon which the view is created, it simplifies the complexity of the query.
* The view is used to hide the complexity of the underlying tables used in a database from the end-users.
* Views are useful in case of re-designing the database so as not to affect any other applications using the same database.
* The data of the computed columns can be calculated very easily when we query the data from the view, as views enable computed columns.

#### Disadvantages

Despite the many advantages that the views offer, it still has some disadvantages stated as below:

* One of the major disadvantages of using view comes into the picture when we change the table structures frequently upon which the view is created. So when the table structures are changed, the view also needs to be changed.
* Also, the usage of view slows down the performance of the queries.

### Conclusion

Views are widely used for their many advantages. They add an extra security layer to the database which is very essential for any relational database management system. Views are flexible in case of exposure of the data to the end-users by showing only the data necessary for example using read-only views to limit the privileges to the users. But also views can be disadvantageous if the underlying table structures change much frequently thereby increasing the complexity of changing the views according to the table structures.

###### So, in short, the following are the limitations and Dis-Advantages of Views in SQL Server

1. We cannot pass parameters to SQL Server views
2. Cannot use an Order by clause with views without specifying FOR XML, OFFSET or TOP
3. The Views cannot be created based on Temporary Tables in SQL Server
4. We cannot associate Rules and Defaults with SQL Server views

# SQL Server Triggers

The SQL Server trigger is a special type of stored procedures that is automatically executed when an event occurs in a specific database server. SQL Server provides us with two main types of triggers: the **DML** Triggers and the **DDL** triggers. The DDL triggers will be fired in response to different Data Definition Language (DDL) events, such as executing CREATE, ALTER, DROP, GRANT, DENY, and REVOKE T-SQL statements. The DDL trigger can respond to the DDL actions by preventing these changes from affecting the database, perform another action in response to these DDL actions or recording these changes that are executed against the database.

# AFTER… DML Trigger

# Assume that we need to track the DML actions that are performed on a specific table and write these logs in a history table, where the ID of the inserted, updated or deleted record and the action that is performed will be written to the history table. The CREATE TABLE T-SQL statements below can be used to create both the source and history tables:

CREATE TABLE TriggerDemo\_Parent

(

ID INT IDENTITY (1,1) PRIMARY KEY,

Emp\_First\_name VARCHAR (50),

Emp\_Last\_name VARCHAR (50),

Emp\_Salary INT

)

GO

CREATE TABLE TriggerDemo\_History

(

ID INT IDENTITY (1,1) PRIMARY KEY,

ParentID INT,

PerformedAction VARCHAR (50),

)

GO

To track the **INSERT operation**, we will create a DML trigger that will be fired after performing an INSERT operation on the parent table. This trigger will retrieve the last inserted ID value to that parent table from the virtual inserted table, as in the CREATE TRIGGER T-SQL statement below:

CREATE TRIGGER AfterInsertTrigger

ON TriggerDemo\_Parent

AFTER INSERT

AS

INSERT INTO TriggerDemo\_History VALUES ((SELECT TOP 1 inserted.ID FROM inserted), 'Insert')

GO

Tracking the **DELETE operation** can be achieved by creating a DML trigger that is fired after performing the DELETE operation on the parent table. Again, the trigger will retrieve the ID value of the last deleted record from that parent table from the virtual deleted table, as in the CREATE TRIGGER T-SQL statement below:

CREATE TRIGGER AfterDeleteTrigger

ON TriggerDemo\_Parent

AFTER DELETE

AS

INSERT INTO TriggerDemo\_History VALUES ((SELECT TOP 1 deleted.ID FROM deleted), 'Delete')

GO

Finally, we will track also the **UPDATE operation** by creating a DML trigger that will be fired after performing an UPDATE operation on the parent table. Within this trigger, we will retrieve the last updated ID value from that parent table from the virtual inserted table, taking into consideration that the UPDATE process is performed by deleting the record and inserting a new record with the updated values, as in the CREATE TRIGGER T-SQL statement below:

CREATE TRIGGER AfterUPDATETrigger

ON TriggerDemo\_Parent

AFTER UPDATE

AS

INSERT INTO TriggerDemo\_History VALUES ((SELECT TOP 1 inserted.ID FROM inserted), 'UPDATE')

GO

The tables and the triggers are ready now for our testing. If you try to insert a new record into the parent table using the INSERT INTO T-SQL statement below:

INSERT INTO TriggerDemo\_Parent VALUES ('AAA','BBB',500)

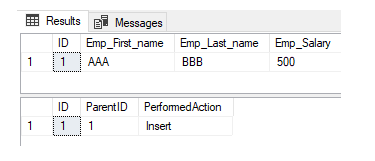
It is also clear when you check the data inserted into both the parent and history tables using the SELECT statements below:

SELECT \* FROM TriggerDemo\_Parent

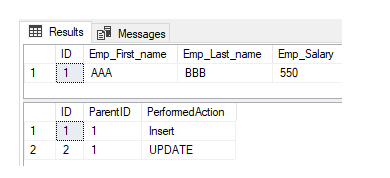
GO

SELECT \* FROM TriggerDemo\_History

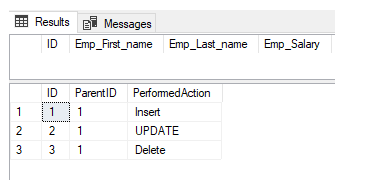
AFTER INSERT trigger:



AFTER UPDATE trigger:



AFTER DELETE trigger



# INSTEAD OF… DML Trigger

The second type of DML triggers is the INSTEAD OF DML trigger. As mentioned previously, the INSTEAD OF trigger will override the statement of the action that fires the trigger with the statement provided in the trigger. Assume that we need to log the DML actions that users are trying to perform on a specific table, without allowing them to perform that action. The CREATE TABLE T-SQL statements below can be used to create both the source and alternative tables:

CREATE TABLE TriggerDemo\_NewParent

(

ID INT IDENTITY (1,1) PRIMARY KEY,

Emp\_First\_name VARCHAR (50),

Emp\_Last\_name VARCHAR (50),

Emp\_Salary INT

)

GO

CREATE TABLE TriggerDemo\_InsteadParent

(

ID INT IDENTITY (1,1) PRIMARY KEY,

ParentID INT,

PerformedAction VARCHAR (50),

)

GO

After creating the two tables, we will insert a single record into the source table for our demo using the INSERT INTO statement below:

INSERT INTO TriggerDemo\_NewParent VALUES ('AA','BB', 500)

For this demo, we will create three triggers to override the **INSERT, UPDATE, and DELETE** operations. The first trigger will be used to prevent any insert operation on the parent table and the log that change into the alternative table. The trigger is created using the CREATE TRIGGER T-SQL statement below:

CREATE TRIGGER InsteadOfInsertTrigger

ON TriggerDemo\_NewParent

INSTEAD OF INSERT

AS

INSERT INTO TriggerDemo\_InsteadParent VALUES ((SELECT TOP 1 inserted.ID FROM inserted), 'Trying to Insert new ID')

GO

The second trigger is used to prevent any update operation on the parent table and the log that change into the alternative table. This trigger is created as below:

CREATE TRIGGER InsteadOfUpdateTrigger

ON TriggerDemo\_NewParent

INSTEAD OF UPDATE

AS

INSERT INTO TriggerDemo\_InsteadParent VALUES ((SELECT TOP 1 inserted.ID FROM inserted), 'Trying to Update an existing ID')

GO

The last trigger will be used to prevent any **delete operation** on the parent table and the log that change into the alternative table. This trigger is created as follows:

CREATE TRIGGER InsteadOfDeleteTrigger

ON TriggerDemo\_NewParent

INSTEAD OF DELETE

AS

INSERT INTO TriggerDemo\_InsteadParent VALUES ((SELECT TOP 1 inserted.ID FROM inserted), 'Trying to Delete an existing ID')

GO

The two tables and the three triggers are ready now. If you try to insert a new value into the parent table using the INSERT INTO T-SQL statement below:

INSERT INTO TriggerDemo\_NewParent VALUES ('CCC','DDD',500)

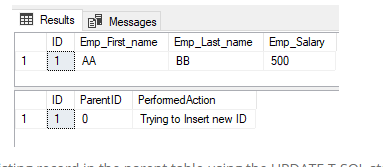
Then check both the parent and the alternative table records using the SELECT statements below:

SELECT \* FROM TriggerDemo\_NewParent

GO

SELECT \* FROM TriggerDemo\_InsteadParent

Due to the fact that we have the **INSTEAD OF INSERT trigger in the parent table**, you will see from the result that no new record is inserted into the parent table, and a log for the insert operation is inserted into the alternative table, as shown in the result below:



Trying to update an existing record in the parent table using the UPDATE T-SQL statement below:

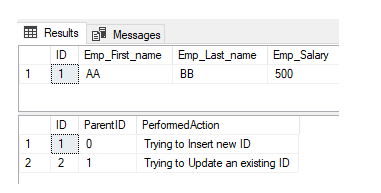
UPDATE TriggerDemo\_NewParent SET Emp\_Salary=550 WHERE ID=1

Then checking both the parent and the alternative table records using the SELECT statements below:

SELECT \* FROM TriggerDemo\_NewParent

GO

SELECT \* FROM TriggerDemo\_InsteadParent



## Advantages and Disadvantages of Triggers

**Advantages**

* Forcing **security** approvals on the table that are present in the database
* Triggers provide another way to check the **integrity of data**
* **Counteracting invalid** exchanges
* Triggers **handle errors** from the database layer
* Normally triggers can be useful for **inspecting the data** changes in tables
* Triggers give an alternative way to run s**cheduled tasks**. Using triggers, we don’t have to wait for the scheduled events to run because the triggers are invoked automatically before or after a change is made to the data in a table

**Disadvantages**

* Triggers can only provide extended **validations**, i.e., not all kind validations. For simple validations, you can use the NOT NULL, UNIQUE, CHECK and FOREIGN KEY constraints
* Triggers may increase the **overhead** of the database
* Triggers can be difficult to**troubleshoot** because they execute automatically in the database, which may not invisible to the client applications

# Clustered Vs Non Clustered Index

# What Is A Clustered Index?

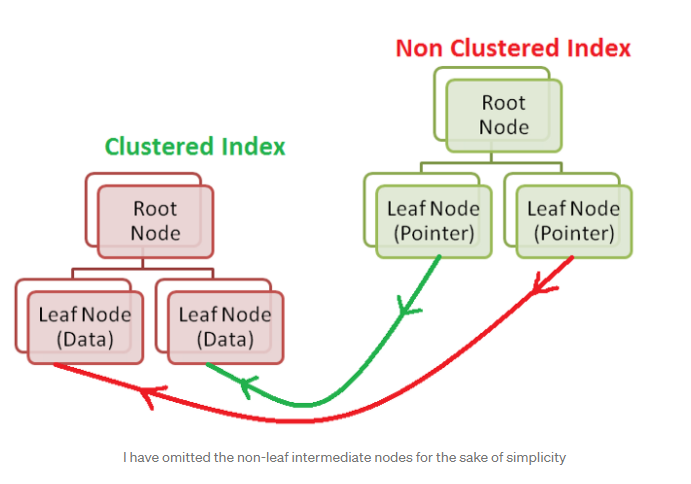
A clustered index:

* Physically orders the data according to the columns of the index.
* Therefore, when we create a clustered index, we are essentially physically ordering the rows of the table. As a result, the clustered index contains all of the columns of a row.
* There can only be one clustered index on a table as the rows in the table will be sorted by the order specified in the clustered index.
* This means that whenever you insert or update a record, the clustered index ensures that the order is maintained. This process can be a hit on the performance of your application.
* For the tables without a clustered index, data is stored in an unordered heap. Heaps do not have an ordered structure, they do not have a natural order. This can get problematic as the table grows in size and data gets fragmented over time.
* Clustered indexes are stored as trees. With clustered index, the actual data is stored in the leaf nodes. This can speed up getting the data when a lookup is performed on the index. As a consequence, a lower number of IO operations are required.
* Additionally the index can be reorganised and rebuilt without moving the data out into a new table.
* SQL Server automatically creates a Primary key constraint on a clustered index. Having said that, the primary key constraint can be removed. Primary key should uniquely identify a row

# What Is A Non-Clustered Index?

Non-Clustered Index is:

* Also known as B-Tree index.
* The data is ordered in a logical manner in a non-clustered index. The rows can be stored physically in a different order than the columns in a non-clustered index. Therefore, the index is created and the data in the index is ordered logically by the columns of the index.
* Non clustered indexes are created on top of the actual data.
* Unlike clustered index, the leaf pages of the index do not contain the actual data. The leaf pages of the non clustered index contain the pointers.



Pointers are like the page numbers in the Index page of a book.

* Remember the leaf nodes of a clustered index store the actual data.
* If a clustered index does not exist in a table then the leaf nodes of a non clustered index points to the heap page where the data is stored. Heap is essentially an unordered (randomly ordered) rows of data.
* Therefore when we query for data, first the non-clustered index is searched to get the address of the data and then the lookup is performed on the clustered index to get the data. Hence this makes the non-clustered index usually slower than the clustered index.
* There can be multiple non-clustered indexes in a table.

**KEY POINT:**

Although the rows in the table are physically ordered as specified by the order of the clustered index but the non-clustered index contains unique values of columns in the order that is specified for the index, and they contain pointers to the actual data. Think of non-clustered index as a dictionary of a table.

# When Should We Create A Clustered Index?

Note, always execute your query and analyse the execution plan to understand where the performance bottle-necks are but it is wise to create clustered index when:

1. Your query contains filters on specific columns such as WHERE clauses or on JOINS and you are required to query those columns over and over again. Additionally, the requirement is that these columns are not updated as frequently.
2. When you are always required to return data in an ordered manner then you can create a clustered index with the required columns of the ORDER BY clause. As a consequence, all of the rows of the table are not required to be scanned.
3. If your application performs a large number of reads on a table then the clustered index will make the operation extremely fast.
4. Your queries SELECT all or most of the columns of a table then you can think of creating a clustered index.

# When Should We Create A Non-Clustered Index?

Note, always execute your query and analyse the execution plan to understand where the performance bottle-necks are but it is wise to create a non-clustered index when:

* Multiple queries are required to filter rows on a table and there are different set of columns which are used in the WHERE clause and JOINS. Additionally, these columns are updated frequently.
* Again, if you are constantly returning data in specific order then create a non-clustered index to speed up the performance. It will also reduce the memory footprint as you will not be required to perform an additional sorting.
* If certain columns are used more frequently in the queries then you can create a non clustered index on the tables. This is known as the Cover Non Clustered Index.
* If you want to create an index on only those rows that meet a specific criteria then you can add a WHERE clause in the non clustered index. This is known as a Filter Non Clustered Index.
* We can also INCLUDE the columns that we need in the SELECT statement to be added as leaf nodes in the non clustered index. It speeds up the retrieval time.

# How to create a clustered index

Creating a clustered index can be achieved in two ways:

1. Create a primary key constraint, or
2. Use the create index statement.

-- Via primary key constraint  
ALTER TABLE FinTechExplained.Trade ADD CONSTRAINT PK\_Trade\_TradeId  
PRIMARY KEY CLUSTERED (TradeId ASC, TradeType ASC);-- using create index statement  
CREATE CLUSTERED INDEX IX\_Trade\_TradeId ON FinTechExplained.Trade(TradeId ASC, TradeType ASC);

# How to create a non-clustered index

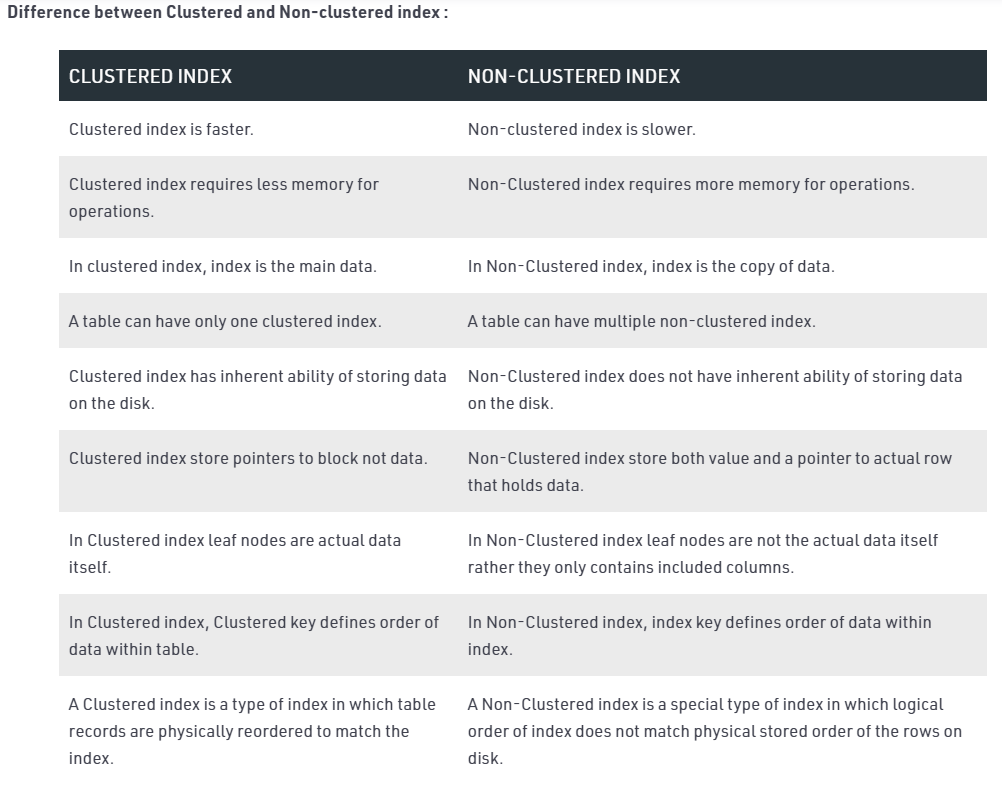
You can create a non-clustered index in two ways:

1. By using the NonClustered keyword
2. Without using the NonClustered Index

-- By using the non-clustered index  
CREATE NONCLUSTERED INDEX IX\_Trade\_CreatedAtCreatedBy ON FinTechExplained.Trade(CreatedAt ASC,CreatedBy ASC);CREATE INDEX IX\_Trade\_CreatedAtCreatedBy ON FinTechExplained.Trade (CreatedAt ASC,CreatedBy ASC);

## **Difference Between Clustered And Non-Clustered Index In Tabular Form**

|  |  |  |
| --- | --- | --- |
| **BASIS OF COMPARISON** | **CLUSTERED INDEX** | **NON-CLUSTERED INDEX** |
| **Description** | A clustered index is a kind of file that sorts the information in the table on their key qualities. | A non-clustered index stores the meta-information in one area and files in another area. The file comprise of the pointers to the location of that information. |
| **Function** | It can be used to sort the record and store the index in physical memory. | It creates a logical ordering of data rows and uses pointers for accessing the physical data files. |
| **Size** | The size of the clustered index is quite large. | The size of the non-clustered index is small compared to the clustered index. |
| **Storage of Records** | It stores records in the leaf node of the index. | It does not store records in the leaf node of an index that means it takes extra space for data. |
| **Keys** | In clustered index, clustered key defines order of data within table. | It can work with unique constraints that act as a composite key. |
| **Speed Of Reading Operations** | Clustered indexes are faster to read than non-clustered indexes as data is physically stored in index order. | Non-clustered indexes are slower in read operation as compared to clustered indexes. |
| **Storage Of Pointers** | Clustered index store pointers to block not data. | Non-clustered index store both value and a pointer to actual row that holds data. |
| **Table Composition** | A table can have only one clustered index. | A table can have multiple non-clustered indexes. |
| **Storage Of Logical Structure** | No extra space is required to store logical structure. | Extra space is required to store logical structure. |
| **Advantage** | A clustered index can improve the performance of data retrieval. | Non-clustered index creates the columns that are used in joins. |
| **Insert And Update Operations** | It is slower for insert and update operations than non-clustered index. | It is quicker for insert and update operations than clustered index. |
| **Table** | If the table does not have clustered index, it is referred to as a ‘’Heap’’. | A table may not have any non-clustered indexes. |
| **Index Id** | A clustered index always has index Id of 0. | A non-clustered index always contains an index Id>0. |

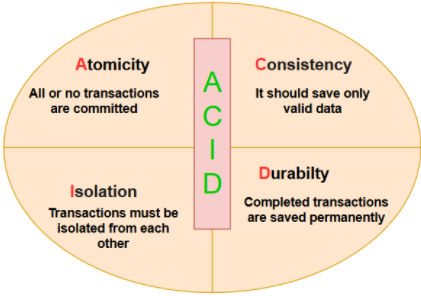


## [SQL Server Deadlocks and the Best Ways to Avoid](https://www.quest.com/community/blogs/b/database-management/posts/the-anatomy-of-sql-server-deadlocks-and-the-best-ways-to-avoid-them)

Database professionals are routinely confronted with [database performance](https://www.quest.com/community/blogs/b/database-management/posts/3-easy-ways-to-improve-your-database-performance)issues like improper indexing and poorly written code in production SQL instances. Suppose you updated a transaction and SQL Server reported the following deadlock message. For DBAs just starting out, this might come as a shock.

### **What is a SQL Server deadlock?**

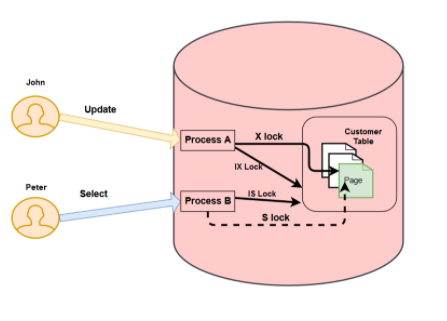
SQL Server is a highly transactional database. For example, suppose you are supporting the database for an online shopping portal where you receive new orders from customers around the clock. Multiple users are likely performing the same activity at the same time. In this case, your database should follow the Atomicity, Consistency, Isolation, Durability (ACID) properties in order to be consistent, reliable and protect data integrity.



To follow the ACID properties, SQL Server uses locking mechanisms, constraints and write-ahead logging. Various lock types include: exclusive lock(X), shared lock(S), update lock (U), intent lock (I), schema lock (SCH) and bulk update lock (BU). These locks can be acquired on the key, table, row, page and database level.

Suppose you have two users, John and Peter who are connected to the customer database.

* John wants to update the records for the customer having [customerid] 1.
* At the same time, Peter wants to retrieve the value for the customer having [customerid] 1.



In this case, SQL Server uses the following locks for both John and Peter.

**Locks for John**

* It takes an intent exclusive (IX) lock on the customer table and page that contains the record.
* It further takes an exclusive (X) lock on the row that John wants to update. It prevents any other user from modifying the row data until process A releases its lock.

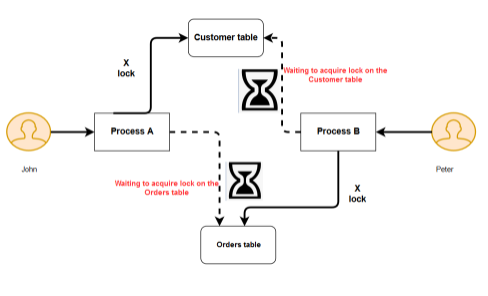
**Locks for Peter**

* It acquires an intent shared (IS) lock on the customer table and the page that contains the record as per the where clause.
* It tries to take a shared lock to read the row. This row already has an exclusive lock for John.

In this case, Peter needs to wait until John finishes his work and releases the exclusive lock. This situation is known as blocking.

Now, suppose in another scenario, John and Peter have the following locks.

* John has an exclusive lock on the customer table for the customer id 1.
* Peter has an exclusive lock on the orders table for the customer id 1.
* John requires an exclusive lock on the orders table to finish his transaction. Peter already has an exclusive lock on the orders table.
* Peter requires an exclusive lock on the customer table to finish his transaction. John already has an exclusive lock on the customer table.



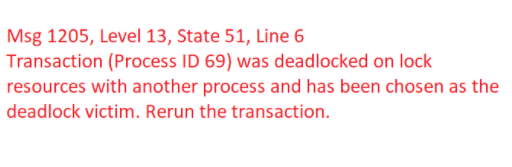
In this case, neither of the transactions can proceed because each transaction requires a resource held by the other transaction. This situation is known as a SQL Server deadlock.

### **SQL Server deadlock monitoring mechanisms**

SQL Server monitors deadlock situations periodically using the deadlock monitor thread. This checks the processes involved in a deadlock and identifies if a session has become a deadlock victim. It uses an internal mechanism to identify the deadlock victim process. By default, the transaction with the least amount of resources required for rollback is considered a victim.

SQL Server kills the victim session so that another session can acquire the required lock to complete its transaction. By default, SQL Server checks the deadlock situation every 5 seconds using the deadlock monitor. If it detects a deadlock, it might reduce the frequency from 5 seconds to 100 milliseconds depending upon the deadlock occurrence. It again resets the monitoring thread to 5 seconds if frequent deadlocks do not occur.

Once the SQL Server kills a process as a deadlock victim, you will receive the following message. In this session, process ID 69 was a deadlock victim.



### **The impacts of using SQL Server deadlock priority statements**

By default, SQL Server marks the transaction with the least expensive rollback as a [deadlock victim](https://www.sqlshack.com/sql-server-selects-deadlock-victim/). Users can set the deadlock priority in a transaction using the DEADLOCK\_PRIORITY statement.

SET DEADLOCK\_PRIORITY

It uses the following arguments:

* Low: It is equivalent to deadlock priority -5
* Normal: It is the default deadlock priority 0
* High: It is the highest deadlock priority 5.

We can also set numeric values for the deadlock priority from -10 to 10 (total 21 values).

Let’s look at a few examples of deadlock priority statements.

**Example 1:**

Session 1 with deadlock priority: Normal (0) > Session 2 with deadlock priority: Low (-5)

**Deadlock Victim:**Session 2



**Example 2:**

Session 1 with deadlock priority: Normal (0) < Session 2 with deadlock priority: High (+5)

**Deadlock Victim:**Session 1



You can capture SQL Server deadlock information in the following ways:

SQL Server profiler

SQL Server extended events

SQL Server error logs

Default traces in SQL Server

### **5 Types of SQL Server deadlocks**

**1) Bookmark lookup deadlock**

Bookmark lookup is a commonly found deadlock in SQL Server. It occurs due to a conflict between the select statement and the DML (insert, update and delete) statements. Usually, SQL Server chooses the select statement as a deadlock victim because it does not cause data changes and the rollback is quick. To avoid the bookmark lookup, you can use a covering index. You can also use a NOLOCK query hint in the select statements, but it reads uncommitted data.

**2) Range scan deadlock**

Sometimes, we use a SERIALIZABLE isolation level at the server level or the session level. It is a restrictive isolation level for concurrency control and can create range scan locks instead of a page or row level locks. In the SERIALIZABLE isolation level, users cannot read data if it is modified but waiting to be committed in a transaction. Similarly, if a transaction reads data, another transaction cannot modify it. It provides the lowest concurrency so we should use this isolation level in specific application requirements.

**3) Cascading constraint deadlock**

SQL Server uses the parent-child relationship among tables using the foreign key constraints. In this scenario, if we update or delete a record from the parent table, it takes necessary locks on the child table to prevent orphan records. To eliminate these deadlocks, you should always modify data in a child table first followed by the parent data. You can also work directly with the parent table using the DELETE CASCADE or UPDATE CASCADE options. You should also create appropriate indexes on the foreign key columns.

**4) Intra-query parallelism deadlock**

Once a user submits a query to the SQL query engine, query optimizer builds an [optimized execution plan](https://www.quest.com/community/blogs/b/database-management/posts/sql-server-execution-plan-what-is-it-and-how-does-it-help-with-performance-problems). It can execute the query in a serial or parallel order depending upon the query cost, the maximum degree of parallelism (MAXDOP) and cost threshold for parallelism.

In a parallelism mode, SQL Server assigns multiple threads. Sometimes for a large query in a parallelism mode, these threads start blocking each other. Eventually, it converts into deadlocks. In this case, you need to review the execution plan and your MAXDOP and cost threshold for parallelism configurations. You can also specify the MAXDOP at the session level to troubleshoot the deadlock scenario.

**5) Reverse object order deadlock**

In this type of deadlock, multiple transactions access objects in a different order in the T-SQL. This causes blocking among the resources for each session and converts it into a deadlock. You always want to access objects in a logical order so that it does not lead to a deadlock situation.

### **Useful ways to avoid and minimize SQL Server deadlocks**

* Try to keep transactions short; this will avoid holding locks in a transaction for a long period of time.
* Access objects in a similar logical manner in multiple transactions.
* Create a covering index to reduce the possibility of a deadlock.
* [Create indexes](https://www.quest.com/community/blogs/b/database-management/posts/sql-server-indexes-key-requirements-performance-impacts-and-considerations) to match the foreign key columns. This way, you can eliminate deadlocks due to cascading referential integrity.
* Set deadlock priorities using the SET DEADLOCK\_PRIORITY session variable. If you set the deadlock priority, SQL Server kills the session with the lowest deadlock priority.
* Utilize the error handling using the try-catch blocks. You can trap the deadlock error and rerun the transaction in the event of a deadlock victim.
* Change the isolation level to the READ COMMITTED SNAPSHOT ISOLATION or SNAPSHOT ISOLATION. This changes the SQL Server locking mechanism. Although, you should be careful in changing the isolation level, as it might impact other queries negatively.

### **SQL Server deadlock considerations**

Deadlocks are a natural mechanism in SQL Server to avoid the session holding locks and waiting for other resources. You should capture deadlock queries and optimize them so that they do not conflict with one other. It’s important to capture the lock for a short span and release it, so that other queries can effectively use it.

SQL Server deadlocks happen, and while SQL Server internally handles deadlock situations, you should try to minimize them whenever possible. Some of the best ways to eliminate deadlocks are by creating an [index](https://www.quest.com/community/blogs/b/database-management/posts/sql-server-indexes-key-requirements-performance-impacts-and-considerations), applying application code changes or carefully inspecting the resources in a deadlock graph

### **Transaction Isolation Levels in DBMS**

**Prerequisite**

As we know that, in order to maintain consistency in a database, it follows ACID properties. Among these four properties (Atomicity, Consistency, Isolation and Durability) Isolation determines how transaction integrity is visible to other users and systems. It means that a transaction should take place in a system in such a way that it is the only transaction that is accessing the resources in a database system.  
Isolation levels define the degree to which a transaction must be isolated from the data modifications made by any other transaction in the database system. A transaction isolation level is defined by the following phenomena –

* **Dirty Read –**A Dirty read is the situation when a transaction reads a 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 same row twice, and get 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:

1. **Read Uncommitted –**Read Uncommitted is the lowest isolation level. In this level, one transaction may read not yet committed changes made by other transaction, thereby allowing dirty reads. In 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 allows dirty read. The transaction holds a read or write lock on the current row, and thus prevent 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 all rows it inserts, updates, or deletes. Since other transaction cannot read, update or delete these rows, consequently it avoids non-repeatable read.
4. **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.

The Table is given below clearly depicts the relationship between isolation levels, read phenomena and locks :

