# SQL Server

<http://www.pragimtech.com/download_sql_server_videos.aspx>

<https://www.softwaretestinghelp.com/50-popular-sql-interview-questions-for-testers/>

<https://www.studytonight.com/dbms/set-operation-in-sql.php>

<https://www.codeproject.com/Articles/620086/CASCADE-in-SQL-Server-with-example>

# Execution Plan

* Execution plan will be generated by Query optimizer with the help of statistics and Algebrizer\processor tree. It is the result of Query optimizer and tells how to do\perform your work\requirement.
* There are two different execution plans - Estimated and Actual.
* Estimated execution plan indicates optimizer view.
* Actual execution plan indicates what executed the query and how was it done.
* Execution plans are stored in memory called plan cache, hence can be reused. Each plan is stored once unless optimizer decides parallelism for the execution of the query.
* There are three different formats of execution plans available in SQL Server - Graphical plans, Text plans, and XML plans.
* SHOWPLAN is the permission which is required for the user who wants to see the execution plan.

# Create table

CREATE TABLE Employee

( Id INT NOT NULL CONSTRAINT [PK\_Employee] PRIMARY KEY IDENTITY,

FirstName VARCHAR(100) NOT NULL,

LastName VARCHAR(100) NOT NULL,

Gender VARCHAR(100) NOT NULL,

Age VARCHAR(100) NULL,

Mobile VARCHAR(100) NULL,

Email VARCHAR(100) NULL

)

CREATE TABLE SystemUser

( Id INT NOT NULL CONSTRAINT [PK\_SystemUser] PRIMARY KEY IDENTITY,

Email VARCHAR(100) NOT NULL CONSTRAINT [UK\_SystemUser\_Email] UNIQUE,

Code VARCHAR(10) NOT NULL CONSTRAINT [UK\_SystemUser\_Code] UNIQUE

)

# INSERT INTO

INSERT INTO [Employee]

(FirstName, LastName, Mobile, Age, Gender)

VALUES ('Shailendra', 'Patil', '12345678978', 25, 'MALE' ),

('Aditya', 'Mundle', '98765432145', 29, 'MALE' ),

('Chetan', 'More', '54321987654', 28, 'MALE' ),

('Kirti', 'Bandal', '54321987654', 28, 'FEMALE' )

INSERT INTO [SystemUser]

(Email, Code)

VALUES ('spatil@xpanxion.com', 'spatil' )

# SELECT INTO

* SELECT Customers.CustomerName, Orders.OrderID  
  INTO CustomersOrderBackup2017  
  FROM Customers  
  LEFT JOIN Orders ON Customers.CustomerID = Orders.CustomerID;

**Tip:** SELECT INTO can also be used to create a new, empty table using the schema of another. Just add a WHERE clause that causes the query to return no data:

* SELECT \* INTO *newtable*  
  FROM *oldtable*  
  WHERE 1 = 0;

# ALTER TABLE

Alter TABLE Employee drop COLUMN Email;

ALTER TABLE Employee

DROP COLUMN SystemUserId

# INSERT INTO SELECT

* INSERT INTO Customers (CustomerName, City, Country)  
  SELECT SupplierName, City, Country FROM Suppliers  
  WHERE Country='Germany';

# Constraints

* ALTER TABLE Employee

ADD CONSTRAINT Mobile CHECK (LEN(Mobile) <=12);

* ALTER TABLE Employee

ADD SystemUserId VARCHAR(10) NOT NULL CONSTRAINT [DF\_Employee\_SystemUserId] DEFAULT(NULL)

* ALTER TABLE Employee

DROP CONSTRAINT [DF\_Employee\_SystemUserId]

* ALTER TABLE Employee

DROP COLUMN SystemUserId

# Rename Column

EXEC sp\_RENAME 'TAble2.LastName' , 'MiddleName', 'COLUMN'

# Order by

SELECT column1, column2,...  
FROM table\_name  
ORDER BY column1, column2, ... ASC|DESC;

# UPADATE

UPDATE Employee

SET Mobile = '12345678978'

WHERE Id = 1

# DELETE

DELETE FROM Employee

WHERE Id = 101;

# SQL Joins

* **In SQL server, there are different types of JOINS.**  
  1. CROSS JOIN  
  2. INNER JOIN   
  3. OUTER JOIN
* **Outer Joins are again divided into 3 types**  
  1. Left Join or Left Outer Join  
  2. Right Join or Right Outer Join  
  3. Full Join or Full Outer Join
* **(INNER) JOIN**: Returns records that have matching values in both tables
* **LEFT (OUTER) JOIN**: Return all records from the left table, and the matched records from the right table
* **RIGHT (OUTER) JOIN**: Return all records from the right table, and the matched records from the left table
* **FULL (OUTER) JOIN**: Return all records when there is a match in either left or right table      

# Practice joins

CREATE TABLE Table1

( Id INT NOT NULL CONSTRAINT [PK\_Table1] PRIMARY KEY IDENTITY,

FirstName VARCHAR(100) NOT NULL, );

CREATE TABLE Table2

( Id INT NOT NULL CONSTRAINT [PK\_Table2] PRIMARY KEY IDENTITY,

UserId INT NOT NULL CONSTRAINT [FK\_Table2\_Table1\_Id] FOREIGN KEY (Id) REFERENCES [Table1] ,

MiddleName VARCHAR(100) NOT NULL,

);

CREATE TABLE Table3

( Id INT NOT NULL CONSTRAINT [PK\_Table3] PRIMARY KEY IDENTITY,

Table2Id INT NOT NULL CONSTRAINT [FK\_Table3\_Table2\_Id] FOREIGN KEY (Id) REFERENCES [Table2] ,

LastName VARCHAR(100) NOT NULL,

);

select \* from Table1 t1

inner join Table2 t2 on t1.Id = t2.UserId

full join Table3 t3 on t2.Id = t3.Table2Id

select \* from Table1 t1

inner join Table2 t2 on t1.Id = t2.UserId

inner join Table3 t3 on t2.Id = t3.Table2Id

select \* from Table1 t1

inner join Table2 t2 on t1.Id = t2.UserId

left join Table3 t3 on t2.Id = t3.Table2Id

select \* from Table1 t1

inner join Table2 t2 on t1.Id = t2.UserId

right join Table3 t3 on t2.Id = t3.Table2Id

select \* from Table1 t1

inner join Table2 t2 on 1=1

select \* from Table1 t1

left join Table2 t2 on 1=1

select \* from Table1 t1

right join Table2 t2 on 1=1

SELECT \* FROM table1

CROSS JOIN table2

# Practice Joins 2

Create table tblDepartment

(

ID int primary key,

DepartmentName nvarchar(50),

Location nvarchar(50),

DepartmentHead nvarchar(50)

)

Go

Insert into tblDepartment values (1, 'IT', 'London', 'Rick')

Insert into tblDepartment values (2, 'Payroll', 'Delhi', 'Ron')

Insert into tblDepartment values (3, 'HR', 'New York', 'Christie')

Insert into tblDepartment values (4, 'Other Department', 'Sydney', 'Cindrella')

Go

Create table tblEmployee

(

ID int primary key,

Name nvarchar(50),

Gender nvarchar(50),

Salary int,

DepartmentId int foreign key references tblDepartment(Id)

)

Go

Insert into tblEmployee values (1, 'Tom', 'Male', 4000, 1)

Insert into tblEmployee values (2, 'Pam', 'Female', 3000, 3)

Insert into tblEmployee values (3, 'John', 'Male', 3500, 1)

Insert into tblEmployee values (4, 'Sam', 'Male', 4500, 2)

Insert into tblEmployee values (5, 'Todd', 'Male', 2800, 2)

Insert into tblEmployee values (6, 'Ben', 'Male', 7000, 1)

Insert into tblEmployee values (7, 'Sara', 'Female', 4800, 3)

Insert into tblEmployee values (8, 'Valarie', 'Female', 5500, 1)

Insert into tblEmployee values (9, 'James', 'Male', 6500, NULL)

Insert into tblEmployee values (10, 'Russell', 'Male', 8800, NULL)

Go

select \* from tblDepartment

select \* from tblEmployee

# ALL, ANY AND EXIST Operator

* SELECT ProductName  
  FROM Products  
  WHERE ProductID = ALL (SELECT ProductID FROM OrderDetails WHERE Quantity = 10);
* SELECT ProductName  
  FROM Products  
  WHERE ProductID = ANY (SELECT ProductID FROM OrderDetails WHERE Quantity > 99);
* SELECT SupplierName  
  FROM Suppliers  
  WHERE EXISTS (SELECT ProductName FROM Products WHERE SupplierId = Suppliers.supplierId AND Price < 20);

# CASE WHEN ELSE

* SELECT OrderID, Quantity,  
  CASE  
      WHEN Quantity > 30 THEN "The quantity is greater than 30"  
      WHEN Quantity = 30 THEN "The quantity is 30"  
      ELSE "The quantity is under 30"  
  END  
  FROM OrderDetails;

# The difference between TRUNCATE, DELETE and DROP

* **TRUNCATE**
* TRUNCATE is a DDL command
* TRUNCATE is executed using a table lock and whole table is locked for remove all records.
* We cannot use Where clause with TRUNCATE.
* TRUNCATE removes all rows from a table.
* Minimal logging in transaction log, so it is performance wise faster.
* TRUNCATE TABLE removes the data by deallocating the data pages used to store the table data and records only the page deallocations in the transaction log.
* Identify column is reset to its seed value if table contains any identity column.
* To use Truncate on a table you need at least ALTER permission on the table.
* Truncate uses the less transaction space than Delete statement.
* Truncate cannot be used with indexed views.
* **DELETE**
* DELETE is a DML command.
* DELETE is executed using a row lock, each row in the table is locked for deletion.
* We can use where clause with DELETE to filter & delete specific records.
* The DELETE command is used to remove rows from a table based on WHERE condition.
* It maintain the log, so it slower than TRUNCATE.
* The DELETE statement removes rows one at a time and records an entry in the transaction log for each deleted row.
* Identity of column keep DELETE retain the identity.
* To use Delete you need DELETE permission on the table.
* Delete uses the more transaction space than Truncate statement.
* Delete can be used with indexed views.
* **DROP**
* The DROP command removes a table from the database.
* All the tables' rows, indexes and privileges will also be removed.
* No DML triggers will be fired.
* The operation cannot be rolled back.
* DROP and TRUNCATE are DDL commands, whereas DELETE is a DML command.
* DELETE operations can be rolled back (undone), while DROP and TRUNCATE operations cannot be rolled back.

# Normalization

<https://www.dotnettricks.com/learn/sqlserver/database-normalization-basics>

* 1NF
  + A database table is said to be in 1NF if it contains no repeating fields/columns.
* 2NF
  + database table is said to be in 2NF if it is in 1NF and contains only those fields/columns that are functionally dependent(means the value of field is determined by the value of another field(s)) on the primary key
* 3NF
  + A database table is said to be in 3NF if it is in 2NF and all non keys fields should be dependent on primary key or We can also said a table to be in 3NF if it is in 2NF and no fields of the table is transitively functionally dependent on the primary key
* 4NF
  + A database table is said to be in 4NF if it is in BCNF and primary key has one-to-one relationship to all non keys fields or We can also said a table to be in 4NF if it is in BCNF and contains no multi-valued dependencies
* 5NF
  + A database table is said to be in 5NF if it is in 4NF and contains no redundant values or We can also said a table to be in 5NF if it is in 4NF and contains no join dependencies

# Stored Procedure with Output Parameter

* Create Procedure spGetEmployeeCountByGender  
  @Gender nvarchar(20),  
  @EmployeeCount int Output  
  as  
  Begin  
   Select @EmployeeCount = COUNT(Id)   
   from tblEmployee   
   where Gender = @Gender  
  End
* **Ececute SP**

Declare @EmployeeTotal int  
Execute spGetEmployeeCountByGender 'Female', @EmployeeTotal output  
Print @EmployeeTotal

# Stored Procedure with return value

* Create Procedure spGetTotalCountOfEmployees2  
  as  
  Begin  
   return (Select COUNT(ID) from Employees)  
  End
* **Ececute SP**

Declare @TotalEmployees int  
Execute @TotalEmployees = spGetTotalCountOfEmployees2  
Select @TotalEmployees

* So, using return values, we can only return integers, and that too, only one integer.
* Functions
  1. **Built-in functions**
     + Declare @Number int  
       Set @Number = 97  
       While(@Number <= 122)  
       Begin  
        Print CHAR(@Number)  
        Set @Number = @Number + 1  
       End
     + Declare @Number int  
       Set @Number = 65  
       While(@Number <= 90)  
       Begin  
        Print LOWER(CHAR(@Number))  
        Set @Number = @Number + 1  
       End
     + Select LTRIM('   Hello')
     + Select RTRIM('Hello   ')
     + Select LOWER('CONVERT This String Into Lower Case')
     + Select UPPER('CONVERT This String Into upper Case')
     + Select REVERSE('ABCDEFGHIJKLMNOPQRSTUVWXYZ')
     + Select LEN('SQL Functions   ')
     + Select LEFT('ABCDE', 3)
     + Select RIGHT('ABCDE', 3)
     + Select CHARINDEX('@','sara@aaa.com',1)
     + Select SUBSTRING('John@bbb.com',6, 7)
     + SELECT REPLICATE('Pragim', 3)
     + Select FirstName + SPACE(5) + LastName as FullName  
       From tblEmployee
     + Select Email, PATINDEX('%@aaa% ', Email) as FirstOccurence   
       from tblEmployee  
       Where PATINDEX('%@aaa.%', Email) > 0
  2. **User Defined functions**
  3. **Inline table valued functions**
     + CREATE FUNCTION fn\_EmployeesByGender(@Gender nvarchar(10))  
       RETURNS TABLE  
       AS  
       RETURN (Select Id, Name, DateOfBirth, Gender, DepartmentId  
            from tblEmployees  
            where Gender = @Gender)
  4. **Multi-statement Table Valued function(MSTVF)**
     + Create Function fn\_MSTVF\_GetEmployees()  
       Returns @Table Table (Id int, Name nvarchar(20), DOB Date)  
       as  
       Begin  
        Insert into @Table  
        Select Id, Name, Cast(DateOfBirth as Date)  
        From tblEmployees  
          
        Return  
       End
  5. **Differences between Inline Table Valued functions and Multi-statement Table Valued functions**
  6. In an Inline Table Valued function, the RETURNS clause cannot contain the structure of the table, the function returns. Where as, with the multi-statement table valued function, we specify the structure of the table that gets returned
  7. Inline Table Valued function cannot have BEGIN and END block, where as the multi-statement function can have.
  8. Inline Table valued functions are better for performance, than multi-statement table valued functions. If the given task, can be achieved using an inline table valued function, always prefer to use them, over multi-statement table valued functions.
  9. It's possible to update the underlying table, using an inline table valued function, but not possible using multi-statement table valued function.
     + **Reason for improved performance of an inline table valued function:**  
       Internally, SQL Server treats an inline table valued function much like it would a view and treats a multi-statement table valued function similar to how it would a stored procedure.

# INDEXES

**The following are the different types of indexes in SQL Server**

1. **Clustered**

* Primary key, constraint create **clustered indexes automatically** if no clustered index already exists on the table
* a table can contain only one clustered index
* Create Clustered Index IX\_tblEmployee\_Name  
  ON tblEmployee(Name)

1. **Nonclustered**

* The data is stored in one place, the index in another place
* Create NonClustered Index IX\_tblEmployee\_Name  
  ON tblEmployee(Name)

1. **Unique**
   * UNIQUENESS is a property of an Index, and both CLUSTERED and NON-CLUSTERED indexes can be UNIQUE.
   * This unique non clustered index, ensures that no 2 entires in the index has the same first and last names
   * So creating a UNIQUE constraint, actually creates a UNIQUE index
   * CREATE UNIQUE INDEX IX\_tblEmployee\_City  
     ON tblEmployee(City)  
     WITH IGNORE\_DUP\_KEY
2. Filtered
3. XML
4. Full Text
5. Spatial
6. Columnstore
7. Index with included columns
8. Index on computed columns

* **Advantages of index**
* Not only, the SELECT statement, even the following DELETE and UPDATE statements can also benefit from the index.
* Indexes can also help queries, that ask for sorted results.
* **Diadvantages of Indexes**
* Additional Disk Space
* Insert Update and Delete statements can become slow (update index for each query)

# Synonyms and Views SQL

They are different things. A synonym is an alias for the object directly, a view is a construct over one or more tables.

* **Some reasons to use a view:**
* May need to filter, join or otherwise frig with the structure and semantics of the result set
* May need to provide legacy support for an underlying structure that has changed but has dependencies that you do not want to re-work.
* May provide security where some of the contents of the table should be visible to a class of users but not all. This could involve removing columns with sensitive information or filtering out a subset of the records.
* May wish to encapsulate some business logic in a form that is accessible to users for reporting purposes.
* You may wish to unify data from more than one source.
* **Reasons to use a synonym:**
* You may wish to alias an object in another database, where you can't (or don't want to) hard code the reference to the specific database.
* You may wish to redirect to a source that changes over time, such as an archive table.
* You want to alias something in a way that does not affect the query optimiser.

# Get the last generated identity column value

* **SCOPE\_IDENTITY()** - returns the last identity value that is created in the same session and in the same scope.

Select SCOPE\_IDENTITY()

* **@@IDENTITY** - returns the last identity value that is created in the same session and across any scope.

Select @@IDENTITY

* **IDENT\_CURRENT('TableName')** - returns the last identity value that is created for a specific table across any session and any scope.

Select IDENT\_CURRENT('tblPerson')

# DELETE Duplicate rows from table

WITH EmployeesCTE AS

(

SELECT \*, ROW\_NUMBER()OVER(PARTITION BY ID ORDER BY ID) AS RowNumber

FROM Employees

)

DELETE FROM EmployeesCTE WHERE RowNumber > 1

# SQL MERGE Table

MERGE targetTable  
Using sourceTable  
ON mergeCondition  
WHEN MATCHED  
THEN updateStatement  
WHEN NOT MATCHED BY TARGET  
THEN insertStatement  
WHEN NOT MATCHED BY SOURCE  
THEN deleteStatement

# Nth highest salary

1. SELECT \*

FROM Employee Emp1

WHERE (1) = (

SELECT COUNT(DISTINCT(Emp2.Salary))

FROM Employee Emp2

WHERE Emp2.Salary > 200)

2. WITH CommonTableExpression AS (

SELECT EmpID, EmpName, EmpSalary,

RN = ROW\_NUMBER() OVER (ORDER BY EmpSalary DESC)

FROM dbo.Salary )

SELECT EmpID, EmpName, EmpSalary

FROM CommonTableExpression

WHERE RN = @NthRow

# SQL query to find employees hired in last n months

-- Replace N with number of months

Select \*

FROM Employees

Where DATEDIFF(MONTH, HireDate, GETDATE()) Between 1 and N

# Transform rows to columns

Select Country, City1, City2, City3

From

(

  Select Country, City,

    'City'+

      cast(row\_number() over(partition by Country order by Country)

             as varchar(10)) ColumnSequence

  from Countries

) Temp

pivot

(

  max(City)

  for ColumnSequence in (City1, City2, City3)

) Piv

# Query to find rows that contain only numerical data

Select Value from TestTable Where ISNUMERIC(Value) = 1

# Querie retrieve all student rows whose Name starts with letter 'M'. without using the LIKE operator.

SELECT \* FROM Students WHERE CHARINDEX('M',Name) = 1

SELECT \* FROM Students WHERE LEFT(Name, 1) = 'M'

SELECT \* FROM Students WHERE SUBSTRING(Name, 1, 1) = 'M'