**INTERVIEW QUESTIONS ASKED ME:**

**TSQL QUESTIONS**

**WHAT IS NORMALIZED DATA MEAN?**

**Database normalization**, or simply **normalization**, is the process of organizing the [columns](https://en.wikipedia.org/wiki/Column_(database)) (attributes) and [tables](https://en.wikipedia.org/wiki/Table_(database)) (relations) of a [relational database](https://en.wikipedia.org/wiki/Relational_database) to reduce [data redundancy](https://en.wikipedia.org/wiki/Data_redundancy) and improve data integrity.

Normalization involves arranging attributes in tables based on [dependencies](https://en.wikipedia.org/wiki/Dependency_theory_(database_theory)) between attributes, ensuring that the dependencies are properly enforced by database integrity constraints. Normalization is accomplished through applying some formal rules

|  |  |
| --- | --- |
| **Level** | **Rule** |
| [First normal form (1NF)](http://agiledata.org/essays/dataNormalization.html#1NF) | An entity type is in 1NF when it contains no repeating groups of data. |
| [Second normal form (2NF)](http://agiledata.org/essays/dataNormalization.html#2NF) | An entity type is in 2NF when it is in 1NF and when all of its non-key attributes are fully dependent on its [primary key](http://www.agiledata.org/essays/keys.html). |
| [Third normal form (3NF)](http://agiledata.org/essays/dataNormalization.html#3NF) | An entity type is in 3NF when it is in 2NF and when all of its attributes are directly dependent on the [primary key](http://www.agiledata.org/essays/keys.html). |

**Name the Microsoft System Databases**

* **Master**
* **Model**
* **Msdb**
* **Tempdb**

**How would I use and name some WINDOWS FUNCTIONS ?**

* OVER PARTITION
  + Row\_Number(), RANK(),DENSE\_RANK()

**How to print only odd numbers between 1 to 100**

Declare  
NUM1 number:=1;  
begin  
loop  
NUM1 := NUM1+2;  
dbms\_output.put\_line (NUM1||’,’);  
exit when NUM1=100;  
end loop;  
end;

1. **What is the basic Syntax of a Recursive CTE**
   * WITH UserCTE AS (
   * SELECT userId, userName, managerId,0 AS steps
   * FROM dbo.Users
   * WHERE userId = 7
   * UNION ALL
   * SELECT mgr.userId, mgr.userName, mgr.managerId, usr.steps +1 AS steps
   * FROM UserCTE AS usr
   * INNER JOIN dbo.Users AS mgr
   * ON usr.managerId = mgr.userId
   * )
   * SELECT \* FROM UserCTE AS u;

|  |  |  |
| --- | --- | --- |
| **EmployeeID** | **ManagerID** | **EmployeeName** |
| 1 | 0 | Hiren Solanki |
| 2 | 1 | Gaurang Devluk |
| 3 | 2 | Matthew Dave |
| 4 | 5 | Dhaval Raninga |
| 5 | 3 | Altaf Khan |
| 6 | 2 | Virang Patel |
| 7 | 3 | Rajesh Thakur |
| 8 | 4 | Tusshar Surve |

Now what if you want to find a hierarchy for an employee no. 8, with the following result:

|  |  |  |
| --- | --- | --- |
| **EmployeeName** | **EmployeeID** | **ManagerID** |
| Tusshar Surve | 8 | 4 |
| Dhaval Raninga | 4 | 2 |
| Gaurang Devluk | 2 | 1 |
| Hiren Solanki | 1 | 0 |

Write your T-SQL query using Recursive CTE, like:

Hide   Copy Code

WITH CTE(EmployeeName,empcode,managercode) AS

(

SELECT EmployeeName,empcode,managercode from EMP where empcode=8

UNION ALL

SELECT e.EmployeeName,e.empcode,e.managercode

from EMP e

INNER JOIN CTE c ON e.empcode = c.managercode

)

SELECT \* FROM CTE

1. In the example above you can see the recursive part on the highlighted rows. A step counter is used to calculate degrees of separation from the user to the managers. From just a few people in a hierarchy you can get plenty of relationships, so we also added a restriction to only show user with id 7.
2. The result of this query would be:
3. Hide   Copy Code
4. userId userName managerId steps
5. ----------- ---------------- ----------- -----------
6. 7 Claire 6 0
7. 6 Vince 5 1
8. 5 Lynn 1 2
9. 1 John NULL 3
10. (4 row(s) affected)

**/\* This is an example of a recursive CTE. Google query hints to view how to set the MAX recursion \*/**

begin with

DateRange as (

select 1 as DateRangeID,@StartingDate as StartingDate,

dateadd(millisecond, -3, dateadd(day, 1, @EndingDate)) as EndingDate,

dateadd(millisecond, -3, dateadd(day, 1, @AsOfDate)) as AsOfDate

**/\* I need to verify with Don why he uses Millisecond \*/**

union all

select dr.DateRangeID + 1 as DateRangeID,

dateadd(year, -1, dr.StartingDate) as StartingDate,

dateadd(year, -1, dr.EndingDate) as EndingDate,

dateadd(year, -1, dr.AsOfDate) as AsOfDate

from DateRange as dr

where dr.DateRangeID <= @LookBackPeriod

),

1. **What is the benefit of using a CTE**
   * When you have smaller hierarchical sets of data, CTEs can be handy to produce nice sets of data. But the recursive element of this processing can be heavy for larger tables with deep hierarchies and should thus be used carefully. But in smaller sets, or when later processing is difficult, you’ve got a key tool in the CTE’s ability to call itself.
2. **What makes a CTE recursive**
   * It allows you to join to itself
3. **Should there be a limit to a Recursive CTE and why**
   * Without the limitation the result would be chaotic and unordered.
4. **What causes a Cartesian product**

SELECT name, gender,

CONCAT('$', FORMAT(salary, 2)) AS 'Monthly Salary'

FROM employees, shops;

The problem here is that the query selects from multiple tables. Without any explicit table joins, we wind up with a kind of default join called a Cartesian Join (or Cross Join). The Cross Join name refers to the fact that it joins every row of the first table to every row of the second table.

1. **How do you fix a Cartesian product**
   * Limit the records in the where clause.
   * A Cartesian product is an explosion of records
2. **What is a JUNK dimension**
   * Junk Dimensions Transactional business processes typically produce a number of miscellaneous, low-cardinality flags and indicators. Rather than making separate dimensions for each flag and attribute, you can create a single junk dimension combining them together. This dimension, frequently labeled as a transaction profile dimension in a schema, does not need to be the Cartesian product of all the attributes’ possible values, but should only contain the combination of values that actually occur in the source data.
3. **Can an Index be applied to a #Temp table**
   * YES
4. **Write an one line update scrip to switch the values in column MF**

Id age gender

1 33 M

2 25 F

3 24 M

4 31 F

SELECT CASE

WHEN GENDER = ‘M’ then ‘F’

WHEN GENDER = ‘F’ then ‘M’

END

1. **What is the difference between UNION and UNION ALL?**
   * The UNION command is used to select related information from two tables, much like the JOIN command. However, when using the UNION command all selected columns need to be of the same data type. With UNION, only distinct values are selected.
   * The UNION ALL command is equal to the UNION command, except that UNION ALL selects all values.
   * The difference between Union and Union all is that Union all will not eliminate duplicate rows, instead it just pulls all rows from all tables fitting your query specifics and combines them into a table.
   * **A UNION statement effectively does a SELECT DISTINCT on the results set. If you know that all the records returned are unique from your union, use UNION ALL instead, it gives faster results.**
2. **What is the difference between UNION and a JOIN?**

* Think of joins as horizontal and unions as vertical. A join adds fields to the length(horizontal) and a Union appends records(Vertical)
* [UNION](http://msdn.microsoft.com/en-us/library/ms180026.aspx) combines the results of two or more queries into a single result set that includes all the rows that belong to all queries in the union.
* By using [JOINs](http://msdn.microsoft.com/en-us/library/ms191517.aspx), you can retrieve data from two or more tables based on logical relationships between the tables. Joins indicate how SQL should use data from one table to select the rows in another table.
* Union:

Combines the results of two or more queries into a single result set that includes all the rows that belong to all queries in the union. The UNION operation is different from using joins that combine columns from two tables.

The following are basic rules for combining the result sets of two queries by using UNION:

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

**Full Outer Join:**

Specifies that a row from either the left or right table that does not meet the join condition is included in the result set, and output columns that correspond to the other table are set to NULL. This is in addition to all rows typically returned by the INNER JOIN.

1. **Name the joins**
   * see MAIN TSQL notes
   * Left Join, Right Join, Inner Join, Full Outer Join, Self Join, Cross Join.
2. **What would 3/2 do (data type question)?**

Select 3/2 would return 1

Select 3/2.0 returns 1.5

Select 15/4 would return 3

1. **Tell me about profiler.**
2. **Tell me about the execution plan.**
   * Table Scan means needs index or better index or remove function from where clause
   * Covered Index improves performance
3. **What is the difference between a CTE and a temporary table?**

**CTE**

WITH cte (Column1, Column2, Column3)

AS

(

SELECT Column1, Column2, Column3

FROM SomeTable

)

SELECT \* FROM cte

**Temp Table**

SELECT Column1, Column2, Column3

INTO #tmpTable

FROM SomeTable

SELECT \* FROM #tmpTable

**CTEs...**

* Are unindexable (but can use existing indexes on referenced objects)
* Cannot have constraints
* **Are essentially disposable VIEWs**
* Persist only until the next query is run
* Can be recursive
* Do not have dedicated stats (rely on stats on the underlying objects)
* one really nice feature with CTEs is that you can let it call itself recursively and nest up trees of **parent-child relationships.**

**#Temp Tables...**

* Are real materialized tables that exist in tempdb
* Can be indexed
* Can have constraints
* Persist for the life of the current CONNECTION
* Can be referenced by other queries or subprocedures
* Have dedicated stats generated by the engine

**As far as when to use each, they have very different use cases. If you will have a very large result set, or need to refer to it more than once, put it in a #temp table. If it needs to be recursive, is disposable, or is just to simplify something logically, a CTE is preferred.**

**Also, a CTE should never be used for performance. You will almost never speed things up by using a CTE, because, again, it's just a disposable view. You can do some neat things with them but speeding up a query isn't really one of them.**

**CTE**

A CTE creates the table being used in memory, but is only valid for the specific query following it. When using recursion, this can be an effective structure, but bear in mind that it will need to be recreated everytime it's needed.

You might also consider here a table variable. This is used as a temp table is used, but is also in-memory only, but can be used multiple times without needing to be recreated every time. Also, if you need to persist a few records now, add a few more records after the next select, add a few more records after another op, then return just those handful of records, then this is a handy in-memory structure.

**Temp Table**

A temp table is literally a table created on disk, just in a specific database that everyone knows can be deleted. It is the responsibility of a good dev to destroy those tables when they are no longer needed, but a DBA can also wipe them.

Temporary tables come in two variety: Local and global. In terms of MS Sql Server you use a #tableName designation for local, and ##tableName designation for global (note the use of a single or double # as the identifying characteristic).

Notice that with temp tables, as opposed to table variables or CTE, you can apply indexes and the like, as these are legitimately tables in the normal sense of the word.

**Generally I would use temp tables for longer or larger queries, and CTEs or table variables if I had a small dataset already and wanted to just quickly script up a bit of code for something small.**

1. **What is the difference between a PRIMARY KEY and a UNIQUE CONSTRAINT?**

**Primary key:**

1. Primary key is nothing but it uniquely identifies each row in a table.
2. Primary key does not allow duplicate values, nor NULL.
3. Primary key by default is a clustered index.
4. A table can have only one primary key.

**Unique Constraint:**

1. Unique key is nothing but it uniquely identifies each row in a table.
2. Unique key does not allow duplicate values, but it allows (at most one) NULL.
3. Unique key by default is a non-clustered index.
4. **WHAT are the differences between a Stored Procedure and a Function?**
   * See main list on TSQL list.
5. **What type of Functions are there?**
   * User Defined Scalar Function
   * Table Function
   * Multi Valued Table function.
6. **How would you seperate the UserID from an email address programmatically.**

SELECT RIGHT(Email, LEN(Email) - CHARINDEX('@', email)) Domain ,  
COUNT(Email) EmailCount  
FROM   dbo.email  
WHERE  LEN(Email) > 0  
GROUP BY RIGHT(Email, LEN(Email) - CHARINDEX('@', email))  
ORDER BY EmailCount DESC

Above script will select the domain after @ character. Please note, if there is more than one @ character in the email, this script will not work as that email address is already invalid.

OR

set @domain = substring(@email, charindex(['@'](mailto:'@'), @email)+1, (len(@email) - charindex('.', reverse(@email))) - charindex(['@'](mailto:'@'), @email))

1. **Tell me about the String functions.**
   * See main list on TSQL list.
2. **How would you use a LEFT join in a real world business situation.**
   * Select c.CustomerName, i.InvoiceAmount

From Customer c

LEFT JOIN Invoice i

* + On c.CustomerID = i.CustomerID

1. **What is the difference between a variable and a parameter?**

**Parameter** - Represents a constant value that cannot be changed throughout the session run.

**Variable** - Represents a value that can be changed during session run. There are some functions available to change the variable value like setvariable(), setmaxvariable(),..

* Transact-SQL local variables.

A Transact-SQL variable is an object in Transact-SQL batches and scripts that can hold a data value. After the variable has been declared, or defined, one statement in a batch can set the variable to a value and a later statement in the batch can get the value from the variable. For example:

[Copy](javascript:if%20(window.epx.codeSnippet)window.epx.codeSnippet.copyCode('CodeSnippetContainerCode_b5772073-092b-4485-9f65-f4be48de650f');)

USE AdventureWorks2008R2;

GO

DECLARE @EmpIDVar int;

SET @EmpIDVar = 1234;

SELECT \*

FROM HumanRresources.Employee

WHERE BusinessEntityID = @EmpIDVar;

|  |
| --- |
| **NoteNote** |
| The maximum number of local variables that can be declared in a batch is 10,000. |

* Transact-SQL parameters.

A parameter is an object used to pass data between a stored procedure and the batch or script that executes the stored procedure. Parameters can be either input or output parameters. For example:

[Copy](javascript:if%20(window.epx.codeSnippet)window.epx.codeSnippet.copyCode('CodeSnippetContainerCode_88bf068d-d5df-4396-80e6-67db9880642b');)

USE AdventureWorks2008R2;

GO

CREATE PROCEDURE ParmSample @EmpIDParm int AS

SELECT BusinessEntityID, JobTitle

FROM HumanResources.Employee

WHERE BusinessEntityID = @EmpIDParm ;

GO

EXEC ParmSample @EmpIDParm = 109 ;

GO

1. **What is the difference between a Table Variable and a TMP Table**
2. **Can you use both a Table Variable and a TMP Table**

**in a SP? If yes how would you use them.**

# Difference between CTE and Temp Table and Table Variable

Temp Table or Table variable or CTE are commonly used for storing data temporarily in SQL Server. In this article, you will learn the differences among these three.

## CTE

CTE stands for Common Table expressions. It was introduced with SQL Server 2005. It is a temporary result set and typically it may be a result of complex sub-query. **Unlike temporary table its life is limited to the current query.** It is defined by using WITH statement. CTE improves readability and ease in maintenance of complex queries and sub-queries. Always begin CTE with semicolon.

### A sub query without CTE is given below :

1. **SELECT \* FROM (**
2. **SELECT Addr.Address, Emp.Name, Emp.Age From Address Addr**
3. **Inner join Employee Emp on Emp.EID = Addr.EID) Temp**
4. **WHERE Temp.Age > 50**
5. **ORDER BY Temp.NAME**

### By using CTE above query can be re-written as follows :

1. **;With CTE1(Address, Name, Age)--Column names for CTE, which are optional**
2. **AS**
3. **(**
4. **SELECT Addr.Address, Emp.Name, Emp.Age from Address Addr**
5. **INNER JOIN EMP Emp ON Emp.EID = Addr.EID**
6. **)**
7. **SELECT \* FROM CTE1 --Using CTE**
8. **WHERE CTE1.Age > 50**
9. **ORDER BY CTE1.NAME**

### When to use CTE

1. This is used to store result of a complex sub query for further use.
2. This is also used to create a recursive query.

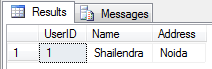
## Temporary Tables

In SQL Server, temporary tables are created at run-time and you can do all the operations which you can do on a normal table. These tables are created inside Tempdb database. Based on the scope and behavior temporary tables are of two types as given below-

1. **Local Temp Table**

Local temp tables are only available to the SQL Server session or connection (means single user) that created the tables. These are automatically deleted when the session that created the tables has been closed. Local temporary table name is stared with single hash ("#") sign.

* 1. **CREATE TABLE #LocalTemp**
  2. **(**
  3. **UserID int,**
  4. **Name varchar(50),**
  5. **Address varchar(150)**
  6. **)**
  7. **GO**
  8. **insert into #LocalTemp values ( 1, 'Shailendra','Noida');**
  9. **GO**
  10. **Select \* from #LocalTemp**

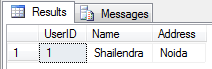
****

The scope of Local temp table exist to the current session of current user means to the current query window. If you will close the current query window or open a new query window and will try to find above created temp table, it will give you the error.

1. **Global Temp Table**

Global temp tables are available to all SQL Server sessions or connections (means all the user). These can be created by any SQL Server connection user and these are automatically deleted when all the SQL Server connections have been closed. Global temporary table name is started with double hash ("##") sign.

* 1. **CREATE TABLE ##GlobalTemp**
  2. **(**
  3. **UserID int,**
  4. **Name varchar(50),**
  5. **Address varchar(150)**
  6. **)**
  7. **GO**
  8. **insert into ##GlobalTemp values ( 1, 'Shailendra','Noida');**
  9. **GO**
  10. **Select \* from ##GlobalTemp**

****

Global temporary tables are visible to all SQL Server connections while Local temporary tables are visible to only current SQL Server connection.

The best way to use a temporary table is to create it and then fill it with data. This goes something like this:

CREATE TABLE #TibetanYaks(

YakID int,

YakName char(30) )

INSERT INTO #TibetanYaks (YakID, YakName)

SELECT YakID, YakName

FROM dbo.Yaks

WHERE YakType = 'Tibetan'

-- Do some stuff with the table

drop table #TibetanYaks

Obviously, this DBA knows their yaks as they're selecting the famed Tibetan yaks, the Cadillac of yaks. Temporary tables are usually pretty quick. Since you are creating and deleting them on the fly, they are usually only cached in memory.

## Table Variables

If you are using SQL Server 2000 or higher, you can take advantage of the new TABLE variable type. These are similar to temporary tables except with more flexibility and they always stay in memory.  The code above using a table variable might look like this:

DECLARE @TibetanYaks TABLE (

YakID int,

YakName char(30) )

INSERT INTO @TibetanYaks (YakID, YakName)

SELECT YakID, YakName

FROM dbo.Yaks

WHERE YakType = 'Tibetan'

-- Do some stuff with the table

Table variables don't need to be dropped when you are done with them.

## Which to Use

* **If you have less than 100 rows generally use a table variable.  Otherwise use  a temporary table.  This is because SQL Server won't create statistics on table variables.**
* **If you need to create indexes on it then you must use a temporary table.**
* **When using temporary tables always create them and create any indexes and then use them.  This will help reduce recompilations.  The impact of this is reduced starting in SQL Server 2005 but it's still a good idea.**

- See more at: http://www.sqlteam.com/article/temporary-tables#sthash.Pa6vc3Jo.dpuf

## Table Variable

This acts like a variable and exists for a particular batch of query execution. It gets dropped once it comes out of batch. This is also created in the Tempdb database but not the memory. This also allows you to create primary key, identity at the time of Table variable declaration but not non-clustered index.

1. **GO**
2. **DECLARE @TProduct TABLE**
3. **(**
4. **SNo INT IDENTITY(1,1),**
5. **ProductID INT,**
6. **Qty INT**
7. **)**
8. **--Insert data to Table variable @Product**
9. **INSERT INTO @TProduct(ProductID,Qty)**
10. **SELECT DISTINCT ProductID, Qty FROM ProductsSales ORDER BY ProductID ASC**
11. **--Select data**
12. **Select \* from @TProduct**
14. **--Next batch**
15. **GO**
16. **Select \* from @TProduct --gives error in next batch**

#### Note

1. Temp Tables are physically **created in the Tempdb database**. These tables act as the normal table **and also can have constraints, index like normal tables.**
2. CTE is a named temporary result set which is used to manipulate the complex sub-queries data. This exists for the scope of statement. **This is created in memory rather than Tempdb database**. You cannot create any index on CTE.
3. Table Variable acts like a variable and exists for a particular batch of query execution. It gets dropped once it comes out of batch. This is also created in the Tempdb database but not the memory.
4. **What is the difference between DELETE and TRUNCATE?**
5. **What is the difference between ISNULL and IS NULL and NULL IF?**
6. **What does COELESCE do?**
7. **What is a recursive query?**
8. **How would you create a recursive query?**

The following example shows the semantics of the recursive CTE structure by returning a hierarchical list of employees, starting with the highest ranking employee, in the Adventure Works Cycles company. A walkthrough of the code execution follows the example.

Transact-SQL

[Copy](javascript:if%20(window.epx.codeSnippet)window.epx.codeSnippet.copyCode('CodeSnippetContainerCode_f98711e0-a02d-4db5-b1d7-f964e1d745ac');" \o "Copy to clipboard.)

-- Create an Employee table.

CREATE TABLE dbo.MyEmployees

(

EmployeeID smallint NOT NULL,

FirstName nvarchar(30) NOT NULL,

LastName nvarchar(40) NOT NULL,

Title nvarchar(50) NOT NULL,

DeptID smallint NOT NULL,

ManagerID int NULL,

CONSTRAINT PK\_EmployeeID PRIMARY KEY CLUSTERED (EmployeeID ASC)

);

-- Populate the table with values.

INSERT INTO dbo.MyEmployees VALUES

(1, N'Ken', N'Sánchez', N'Chief Executive Officer',16,NULL)

,(273, N'Brian', N'Welcker', N'Vice President of Sales',3,1)

,(274, N'Stephen', N'Jiang', N'North American Sales Manager',3,273)

,(275, N'Michael', N'Blythe', N'Sales Representative',3,274)

,(276, N'Linda', N'Mitchell', N'Sales Representative',3,274)

,(285, N'Syed', N'Abbas', N'Pacific Sales Manager',3,273)

,(286, N'Lynn', N'Tsoflias', N'Sales Representative',3,285)

,(16, N'David',N'Bradley', N'Marketing Manager', 4, 273)

,(23, N'Mary', N'Gibson', N'Marketing Specialist', 4, 16);

Transact-SQL

[Copy](javascript:if%20(window.epx.codeSnippet)window.epx.codeSnippet.copyCode('CodeSnippetContainerCode_2f0c176d-61e8-47fc-8fad-5722f10b7905');)

USE AdventureWorks2008R2;

GO

WITH DirectReports (ManagerID, EmployeeID, Title, DeptID, Level)

AS

(

-- Anchor member definition

SELECT e.ManagerID, e.EmployeeID, e.Title, edh.DepartmentID,

0 AS Level

FROM dbo.MyEmployees AS e

INNER JOIN HumanResources.EmployeeDepartmentHistory AS edh

ON e.EmployeeID = edh.BusinessEntityID AND edh.EndDate IS NULL

WHERE ManagerID IS NULL

UNION ALL

-- Recursive member definition

SELECT e.ManagerID, e.EmployeeID, e.Title, edh.DepartmentID,

Level + 1

FROM dbo.MyEmployees AS e

INNER JOIN HumanResources.EmployeeDepartmentHistory AS edh

ON e.EmployeeID = edh.BusinessEntityID AND edh.EndDate IS NULL

INNER JOIN DirectReports AS d

ON e.ManagerID = d.EmployeeID

)

-- Statement that executes the CTE

SELECT ManagerID, EmployeeID, Title, DeptID, Level

FROM DirectReports

INNER JOIN HumanResources.Department AS dp

ON DirectReports.DeptID = dp.DepartmentID

WHERE dp.GroupName = N'Sales and Marketing' OR Level = 0;

GO

### Example Code Walkthrough

1. The recursive CTE, DirectReports, defines one anchor member and one recursive member.
2. The anchor member returns the base result set T0. This is the highest ranking employee in the company; that is, an employee who does not report to a manager.

Here is the result set returned by the anchor member:

[Copy](javascript:if%20(window.epx.codeSnippet)window.epx.codeSnippet.copyCode('CodeSnippetContainerCode_e163cc95-7cd5-4798-ba10-6b19c6ac7297');)

ManagerID EmployeeID Title                         Level

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

NULL      1          Chief Executive Officer        0

1. The recursive member returns the direct subordinate(s) of the employee in the anchor member result set. This is achieved by a join operation between the Employee table and the DirectReports CTE. It is this reference to the CTE itself that establishes the recursive invocation. Based on the employee in the CTE DirectReports as input (Ti), the join (MyEmployees.ManagerID = DirectReports.EmployeeID) returns as output (Ti+1), the employees who have (Ti) as their manager. Therefore, the first iteration of the recursive member returns this result set:

[Copy](javascript:if%20(window.epx.codeSnippet)window.epx.codeSnippet.copyCode('CodeSnippetContainerCode_ef535c2b-9a7a-4baf-a5c4-75f67e3933aa');)

ManagerID EmployeeID Title                         Level

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

1         273        Vice President of Sales       1

1. The recursive member is activated repeatedly. The second iteration of the recursive member uses the single-row result set in step 3 (containing EmployeeID 273) as the input value, and returns this result set:

[Copy](javascript:if%20(window.epx.codeSnippet)window.epx.codeSnippet.copyCode('CodeSnippetContainerCode_a5576c80-492b-474b-a46f-fbeb79eb40a8');)

ManagerID EmployeeID Title                         Level

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

273       16         Marketing Manager             2

273       274        North American Sales Manager  2

273       285        Pacific Sales Manager         2

The third iteration of the recursive member uses the result set above as the input value, and returns this result set:

[Copy](javascript:if%20(window.epx.codeSnippet)window.epx.codeSnippet.copyCode('CodeSnippetContainerCode_b4b3c8a1-70c8-44f7-b542-57812a1b2bfd');)

ManagerID EmployeeID Title                         Level

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

16        23         Marketing Specialist          3

274       275        Sales Representative          3

274       276        Sales Representative          3

285       286        Sales Representative          3

1. The final result set returned by the running query is the union of all result sets generated by the anchor and recursive members.

Here is the complete result set returned by the example:

[Copy](javascript:if%20(window.epx.codeSnippet)window.epx.codeSnippet.copyCode('CodeSnippetContainerCode_a8e2cb90-c27a-43c1-9b5b-dfecc4587e01');)

ManagerID EmployeeID Title                         Level

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

NULL      1          Chief Executive Officer       0

1         273        Vice President of Sales       1

273       16         Marketing Manager             2

273       274        North American Sales Manager  2

273       285        Pacific Sales Manager         2

16        23         Marketing Specialist          3

274       275        Sales Representative          3

274       276        Sales Representative          3

285       286        Sales Representative          3

1. **Tell me about subqueries.**
2. **What would the query look like that retrieves the top paid employee from the emp and dept tables?**

select \* from employee where salary = (select max(salary) from employee)

BY DEPARTMENT

|  |  |
| --- | --- |
|  | with cte as (  select \*, rank() over (partition by dept order by salary desc) as [r]  from employees  )  select \* from cte where [r] = 1; |

1. **What would the query look like that retrieves all the departments and each employee in the departments from the dept and emp tables?**
2. **Write a query that returns the highest paid employee from the Dept and Emp tables**
3. **Write a query that returns the total payroll from each department from the Dept and Emp tables.**
4. **Table A has an unknown number of records. Return only the bottom 3 records from the result set using a subquery and COUNT.**
5. **On a scale of 1-10 how good are you with TSQL, BI, Data warehouse testing**
6. **Have you had experience Importing and Exporting files using SSIS with FTP**
7. **What is the difference between a BRD(business requirements doc) and TRD(technical requirements doc)**
8. **Define the dimensional relation model.**
9. **What is the source to target mapping document.**
10. **How would you use a Slowly Changing Demension in SSIS**

Dimension is a term in data management and data warehousing. It's the logical groupings of data such as geographical location, customer or product information. With Slowly Changing Dimensions (SCDs), data changes slowly rather than changing on a time-based, regular schedule. [~Wikipedia](http://en.wikipedia.org/wiki/Slowly_changing_dimension)

There are different types of slowly changing dimensions:

**SCD Type 0 (Fixed)** – This type is the least frequently used as this type does not accept changes and is fixed after first time insertion; it means once written, the value does not get overwritten.

**SCD Type 1 (Changing)** – In this type, if the data is getting changed it gets overwritten with the new value.

For example consider this example:

|  |  |  |
| --- | --- | --- |
| **SupplierCode** | **SupplierName** | **Address** |
| S0000001 | ABC Company | USA |
| S0000002 | XYZ Corporation | USA |

If the name of the supplier changes over time, as you can see in the change in SupplierName below, the record will be updated. This looks pretty simple to implement, though it does not have history to keep track on.

|  |  |  |
| --- | --- | --- |
| **SupplierCode** | **SupplierName** | **Address** |
| S0000001 | ABC Company Ltd. | USA |
| S0000002 | XYZ Corporation | USA |
|  |  |  |

**SCD Type 2 (Historical)** – In this type, if the data is changed it gets saved in a new record and the previous record with the previous value is marked as outdated.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **SupplierCode** | **SupplierName** | **Address** | **EffectiveDate** | **Expiration Date** |
| S0000001 | ABC Company | USA | 3/2/2013 | 3/2/2013 |
| S0000002 | XYZ Corporation | USA | 3/2/2013 |  |
| S0000001 | ABC Company Ltd. | USA | 3/3/2013 |  |

To maintain SCD type 2, different people take different approaches. For example, one approach is to add effective and expiration dates to indicate a time period during which the record was active. If the expiration date is NULL it indicates the current active record. Another approach is to add one column to indicate the current active record. Normally people use the first approach or a combination of both.

1. **How would you filter a string parameter for a 0 or 1 from the string**

I would use an If Else and use the Keyword LIKE

* + If Param1 LIKE(‘0’,’1’) then Do This ELSE Do This

|  |  |
| --- | --- |
|  | There are probably several different ways to do it, some uglier than others. Here's one:  (Note: dat = ‘18-84-7’)  select \*,  substring(dat,1,charindex('-',dat)-1) as Section,  substring(dat,charindex('-',dat)+1,charindex('-',dat)-1) as TownShip,  reverse(substring(reverse(dat),0,charindex('-',reverse(dat)))) as myRange  from myTable |

1. **What can a SP return verses a UDF User defined function.**
2. **What result will you get when you test for a NULL value?**
   * NULL returns FALSE
3. **When and how would you use MERGE in TSQL**
4. **Have you used power pivot?**
5. **How do you analyze and fix a Correlated query when it grows too large?**
   * Break it up into TEMP tables
6. **How would you process a comma deliminated CSV file using SSIS and convert, format and export it into a datebase table?**
   * ANSWER: Use a Flat file Source tool to extract the data and re-format the file within the Flat file Source tool.
7. **What is a covered index?**
   * A covered Index creates an index for a query on the columns you are specifically including in that query.
8. **What is the difference between a Clustered Index and a NON Clustered Index?**
   * **A Clustered Index is a logical hierarchal ordering of the data**
   * **A NON Clustered Index is a pointer to the data which is not ordered.**
9. **What is the difference between a #temp table and a CTE**
   * **#temp table**
   * **CTE**
     + **Unlike temporary table its life is limited to the current query.**
     + This is used to store result of a complex sub query for further use
     + This is also used to create a recursive query.

**Temp tables**  
  
Behave just like normal tables, but are created in the TempDB database. They persist until dropped, or until the connection that created them disappears. They are visible in the procedure that created them and any procedures that that proc calls.  
  
Just like normal tables, they can have primary keys, constraints and indexes, and column statistics are kept for the table.   
  
Temp tables, while they have space assigned to them in the tempDB database, will generally be accessed only from memory, unless the server is under memory pressure, or the amount of data in the table is large.  
  
**Table Variables**  
  
These tables behave very much like other variables in their scoping rules. They are created when they are declared and are dropped when they go out of scope. They cannot be explicitly dropped.  
  
Like with temp tables, table variables reside in TempDB. they have entries in the system tables in tempDB, just like temp tables, and they follow the same behaviour regarding whether they are in memory or on disk.  
  
Table variables can have a primary key, but indexes cannot be created on them, neither are statistics maintained on the columns. This makes table variables less optimal for large numbers of rows, as the optimiser has no way of knowing the number of rows in the table variable.  
  
**CTE**  
  
In my experience, CTEs are more like temporary views than anything else. When you look at the execution plan, you'll see that they are inlined into the query, not materialised and stored. I find, with the exception of recursion, they're more to make queries simpler to write than faster to run.

1. Temp Tables are physically **created in the Tempdb database**. These tables act as the normal table **and also can have constraints, index like normal tables.**
2. CTE is a named temporary result set which is used to manipulate the complex sub-queries data. This exists for the scope of statement. **This is created in memory rather than Tempdb database**. You cannot create any index on CTE.
3. Table Variable acts like a variable and exists for a particular batch of query execution. It gets dropped once it comes out of batch. This is also created in the Tempdb database but not the memory.
4. **What happens when you put any function into a where clause?**
   * **EXAMPLE:**
     + SELECT \* FROM Prod.Product
     + WHERE StartDate < GETDATE()

**ANSWER:** You get a table scan. To avoid a table scan you would assign the value of the function to a variable.

DECLARE @Date

SET @Date = GetDate()

SELECT \* FROM Prod.Product

WHERE StartDate < @Date

1. **What is and where would you use a TABLE VARIABLE**

Microsoft introduced table variables with SQL Server 2000 as an alternative to using temporary tables. In many cases a table variable can outperform a solution using a temporary table, although we will need to review the strengths and weaknesses of each in this article.

Table variables store a set of records, so naturally the declaration syntax looks very similar to a CREATE TABLE statement, as you can see in the following example:

|  |
| --- |
| DECLARE @ProductTotals TABLE  (    ProductID int,    Revenue money  ) |

While connected to the Northwind data-base, we could write the following SELECT statement to populate the table variable.

|  |
| --- |
| INSERT INTO @ProductTotals (ProductID, Revenue)    SELECT ProductID, SUM(UnitPrice \* Quantity)      FROM [Order Details]      GROUP BY ProductID |

You can use table variables in batches, stored procedures, and user-defined functions (UDFs). We can UPDATE records in our table variable as well as DELETE records.

|  |
| --- |
| UPDATE @ProductTotals    SET Revenue = Revenue \* 1.15  WHERE ProductID = 62      DELETE FROM @ProductTotals  WHERE ProductID = 60      SELECT TOP 5 \*  FROM @ProductTotals  ORDER BY Revenue DESC |

You might think table variables work just like temporary tables (CREATE TABLE #ProductTotals), but there are some differences.

### Scope

Unlike the majority of the other data types in SQL Server, you cannot use a table variable as an input or an output parameter. **In fact, a table variable is scoped to the stored procedure, batch, or user-defined function just like any local variable you create with a DECLARE statement**. The variable will no longer exist after the procedure exits - there will be no table to clean up with a DROP statement.

Although you cannot use a table variable as an input or output parameter, you can return a table variable from a user-defined function – we will see an example later in this article. However, because you can’t pass a table variable to another stored procedure as input – there still are scenarios where you’ll be required to use a temporary table when using calling stored procedures from inside other stored procedures and sharing table results.

The restricted scope of a table variable gives SQL Server some liberty to perform optimizations.

### Performance

**Because of the well-defined scope, a table variable will generally use fewer resources than a temporary table**. Transactions touching table variables only last for the duration of the update on the table variable, so there is less locking and logging overhead.

Using a temporary table inside of a stored procedure may result in additional re-compilations of the stored procedure. Table variables can often avoid this recompilation hit. For more information on why stored procedures may recompile, look at Microsoft knowledge base article 243586 ([INF: Troubleshooting Stored Procedure Recompilation](http://support.microsoft.com/kb/243586/EN-US/)).

### Other Features

Constraints are an excellent way to ensure the data in a table meets specific requirements, and you can use constraints with table variables. The following example ensures ProductID values in the table will be unique, and all prices are less then 10.0.

|  |
| --- |
| UPDATE @ProductTotals    SET Revenue = Revenue \* 1.15  WHERE ProductID = 62    DELETE FROM @ProductTotals  WHERE ProductID = 60    SELECT TOP 5 \*  FROM @ProductTotals  ORDER BY Revenue DESC |

You can also declare primary keys. identity columns, and default values.

|  |
| --- |
| UPDATE @ProductTotals    SET Revenue = Revenue \* 1.15  WHERE ProductID = 62    DELETE FROM @ProductTotals  WHERE ProductID = 60    SELECT TOP 5 \*  FROM @ProductTotals  ORDER BY Revenue DESC |

So far it seems that table variables can do anything temporary tables can do within the scope of a stored procedure, batch, or UDF), but there are some drawbacks.

### Restrictions

You cannot create a non-clustered index on a table variable, unless the index is a side effect of a PRIMARY KEY or UNIQUE constraint on the table (SQL Server enforces any UNIQUE or PRIMARY KEY constraints using an index).

Also, SQL Server does not maintain statistics on a table variable, and statistics are used heavily by the query optimizer to determine the best method to execute a query. Neither of these restrictions should be a problem, however, as table variables generally exist for a specific purpose and aren’t used for a wide range of ad-hoc queries.

The table definition of a table variable cannot change after the DECLARE statement. Any ALTER TABLE query attempting to alter a table variable will fail with a syntax error. Along the same lines, you cannot use a table variable with SELECT INTO or INSERT EXEC queries. f you are using a table variable in a join, you will need to alias the table in order to execute the query.

|  |
| --- |
| UPDATE @ProductTotals    SET Revenue = Revenue \* 1.15  WHERE ProductID = 62    DELETE FROM @ProductTotals  WHERE ProductID = 60    SELECT TOP 5 \*  FROM @ProductTotals  ORDER BY Revenue DESC |

You can use a table variable with dynamic SQL, but you must declare the table inside the dynamic SQL itself. The following query will fail with the error “Must declare the variable ['@MyTable'](mailto:'@MyTable').”

|  |
| --- |
| DECLARE @MyTable TABLE  (    ProductID int ,    Name varchar(10)  )      EXEC sp\_executesql N'SELECT \* FROM @MyTable' |

It’s also important to note how table variables do not participate in transaction rollbacks. Although this can be a performance benefit, it can also catch you off guard if you are not aware of the behavior. To demonstrate, the following query batch will return a count of 77 records even though the INSERT took place inside a transaction with ROLLBACK.

|  |
| --- |
| DECLARE @MyTable TABLE  (    ProductID int ,    Name varchar(10)  )      EXEC sp\_executesql N'SELECT \* FROM @MyTable' |

### Choosing Between Temporary Tables and Table Variables

Now you’ve come to a stored procedure that needs temporary resultset storage. Knowing what we have learned so far, how do you decide on using a table variable or a temporary table?

First, we know there are situations that which demand the use of a temporary table. This in-cludes calling nested stored procedures which use the resultset, certain scenarios using dy-namic SQL, and cases where you need transaction rollback support.

**Secondly, the size of the resultset will determine which solution to choose. If the table stores a resultset so large you require indexes to improve query performance, you’ll need to stick to a temporary table.** In some borderline cases try some performance benchmark testing to see which approach offers the best performance. **If the resultset is small, the table variable is always the optimum choice.**

### An Example: Split

Table variables are a superior alternative to using temporary tables in many situations. **The ability to use a table variable as the return value of a UDF is one of the best uses of table vari-ables.** In the following sample, we will address a common need: a function to parse a delimited string into pieces. In other words, given the string “1,5,9” – we will want to return a table with a record for each value: 1, 5, and 9.

The following user-defined function will walk through an incoming string and parse out the individual entries. The UDF insert the en-tries into a table variable and returns the table variable as a result. As an example, calling the UDF with the following SELECT statement:

|  |
| --- |
| SELECT \* FROM fn\_Split('foo,bar,widget', ',') |

will return the following result set.

|  |
| --- |
| position value  1        foo  2        bar  3        widget |

We could use the resultset in another stored procedure or batch as a table to select against or filter with. We will see why the split function can be useful in the next OdeToCode article. For now, here is the source code to fn\_Split.

|  |
| --- |
| if exists (select \* from dbo.sysobjects where id = ob-ject\_id(N'[dbo].[fn\_Split]') and xtype in (N'FN', N'IF', N'TF'))  drop function [dbo].[fn\_Split]  GO    SET QUOTED\_IDENTIFIER OFF  GO  SET ANSI\_NULLS OFF  GO    CREATE  FUNCTION fn\_Split(@text varchar(8000), @delimiter varchar(20) = ' ')  RETURNS @Strings TABLE  (    position int IDENTITY PRIMARY KEY,    value varchar(8000)  )  AS  BEGIN    DECLARE @index int  SET @index = -1    WHILE (LEN(@text) > 0)      BEGIN      SET @index = CHARINDEX(@delimiter , @text)      IF (@index = 0) AND (LEN(@text) > 0)        BEGIN          INSERT INTO @Strings VALUES (@text)            BREAK        END        IF (@index > 1)        BEGIN          INSERT INTO @Strings VALUES (LEFT(@text, @index - 1))          SET @text = RIGHT(@text, (LEN(@text) - @index))        END      ELSE        SET @text = RIGHT(@text, (LEN(@text) - @index))      END    RETURN    END  GO    SET QUOTED\_IDENTIFIER OFF  GO    SET ANSI\_NULLS ON  GO |

### Summary

**Next time you find yourself using a temporary table, think of table variables instead. Table variables can offer performance benefits and flexibility when compared to temporary tables, and you can let the server clean up afterwards.**

**44)**

**What is Row Version?**

Datetime is a datatype that holds a date or time value.

**45)**

**What is a Timestamp?**

Timestamp is a method for row versioning. In fact, in sql server 2008 this column type was renamed (i.e. timestamp is deprecated) to row version. It basically means that every time a row is changed, this value is increased. This is done with a database counter, i.e. two different rows that where updated in the same transaction have the same row version.

Datetime is a datatype that holds a date or time value.

**45)**

**What are the differences among batches, stored procedures, and triggers?**

**Answer  
A batch** is a group of one or more SQL statements. SQL Server compiles the statements of a batch into a single executable unit, called an execution plan. The statements in the execution plan are then executed one at a time.

**A stored procedure** is a group of SQL statements that is compiled one time and can then be executed many times.

**A trigger** is a special type of stored procedure that is not called directly. Trigger is fired each time row is affected by Insert, Update or Delete command.

**DBA QUESTIONS**

1. **What would I expect to be in place for a database not meaning tools?**
   * Tables
   * Primary keys
   * Foreign keys
   * Relationships
   * Normalized data
2. **How would you defrag a database?**
   * When data is inserted into, deleted from, or updated in a SQL Server table, the indexes defined on that table are automatically updated to reflect those changes. As the indexes are modified, the information stored in them becomes fragmented, resulting in the information being scattered across the data files. When this occurs, the logical ordering of the data no longer matches the physical ordering, which can lead to a deterioration of query performance.
   * To fix this problem, indexes must be periodically reorganized or rebuilt (defragmented) so the physical order of the leaf-level pages matches the logical order of the leaf nodes. This means that you should analyze your indexes periodically to determine whether they’ve become fragmented and the extent of that fragmentation. From there, you can either reorganize or rebuild the affected indexes, depending on the results of your analysis. In this article, I explain how to analyze SQL Server indexes and how reorganize and rebuild them.

**Analyzing Fragmentation**

To analyze SQL Server 2005 or 2008 indexes, you use the system function **sys.dm\_db\_index\_physical\_stats** to determine which indexes are fragmented and the extent of that fragmentation. You can use the function to analyze all the indexes in an instance of SQL Server 2005 or 2008, all indexes in a database, all indexes defined on a table, or a specific index. You can also analyze an index based on the partition number of the indexed object.

The **sys.dm\_db\_index\_physical\_stats** function takes the following parameters (in the order specified):

* **Database ID:** A smallint value that represents the ID number of a database. If null is specified, the function retrieves index-related data from all databases on a SQL Server instance. If you specify null, you must also specify null for the object ID, index ID, and partition number.
* **Object ID:** An int value that represents the ID number of a table or view. If null is specified, the function retrieves index-related data for all tables and views in a specific database or SQL Server instance. If you specify null, you must also specify null for the index ID and partition number.
* **Index ID:** An int value that represents the ID number of an index. If null is specified, the function retrieves index-related data for all indexes defined on the specified table or view. If you specify null, you must also specify null for the partition number. Also, if the object ID refers to a heap, use 0 as the index ID.
* **Partition number:** An int value that represents the partition number of an index or heap. If null is specified, the function retrieves index-related information for all partitions related to a specific object.
* **Mode:** The scan level used to obtain index-related information. Valid inputs include NULL, DEFAULT, or one of the following three modes:
  + **LIMITED:** Scans the smallest number of pages, which means this is the fastest mode. The LIMITED mode is equivalent to NULL and DEFAULT.
  + **SAMPLED:** Scans 1% of all pages. If an index contains fewer than 10,000 pages, then DETAILED mode is used.
  + **DETAILED:** Scans all index pages, which means this is the slowest mode, but most accurate.

You must specify all five parameters, even if their values are null.

The **sys.dm\_db\_index\_physical\_stats** function returns a number of values that provide details about the indexes you specify. The topic “sys.dm\_db\_index\_physical\_stats” in SQL Server Books Online provides details about each of these values. However, several values are worth noting when analyzing an index:

* **avg\_fragmentation\_in\_percent:** Percentage of the logical index that is fragmented.
* **fragment\_count:** Number of fragments in the leaf level.
* **avg\_fragment\_size\_in\_pages:** Average number of pages in a leaf-level fragment.
* **page\_count:** Number of index or data pages.

An index always has at least one fragment (**fragment\_count)**. The maximum number of fragments that an index can have is equal to the number of pages (**page\_count**). For example, an index that is made up of five pages can at the most have five fragments. The larger the fragment, the less disk I/O that is required. So a five-page index with one fragment requires less disk I/O than the index with five fragments. Ideally, the **avg\_fragmentation\_in\_percent** value should be as close to zero as possible, and the **avg\_fragment\_size\_in\_pages** should be as high as possible.

Based on your index analysis, you can determine what action to take. Microsoft recommends that you reorganize your index if the **avg\_fragmentation\_in\_percent** value is less than or equal to 30% and rebuild the index if the value is greater than 30%. (Reorganizing and rebuilding indexes are described in the following sections.)

**Note:** The topic “**sys.dm\_db\_index\_physical\_stats**” in SQL Server Books Online includes a script that automatically analyzes and then, based on that analysis, reorganizes or rebuilds your indexes according to the Microsoft guidelines. You can modify the script as necessary to meet your administrative needs.

Keep in mind that these recommendations are guidelines only. A fragmented index (especially a low percentage) is not always enough of a reason to reorganize or rebuild your index. If your queries do not regularly involve table scans as a result of singleton lookups, defragmenting the index might have no effect on performance. In addition, for smaller indexes with relatively few pages and small amounts of data, you might see little to no improvement when you defragment the index. **FILLFACTOR** settings can also affect the types of improvements you see.

That said, you should still analyze your indexes regularly, and the **sys.dm\_db\_index\_physical\_stats** function is the best tool to use. So let’s take a look at an example of how to use the function to retrieve index-related statistics. In the following SELECT statement, I retrieve index data from the **AdventureWorks** database:

SELECT object\_id AS ObjectID,

  index\_id AS IndexID,

  avg\_fragmentation\_in\_percent AS PercentFragment,

  fragment\_count AS TotalFrags,

  avg\_fragment\_size\_in\_pages AS PagesPerFrag,

  page\_count AS NumPages

FROM sys.dm\_db\_index\_physical\_stats(DB\_ID('AdventureWorks'),

  NULL, NULL, NULL , 'DETAILED')

WHERE avg\_fragmentation\_in\_percent > 0

ORDER BY ObjectID, IndexID

1. **What do you know about linked servers?**
2. **Tell me about query optimization and performance.**
3. **What is your database maintanence plan?**
4. **Tell me about the isolation levels**
5. **Tell me about your database backup plan.**
6. **Tell me about your database restoration plan.**
7. **What are some database performance tecniques?**
8. **What are the differences between an Unique constraint and a Primary Key?**

The difference between a UNIQUE constraint and a Primary Key is that per table you may only have one Primary Key but you may define more than one UNIQUE constraints. Primary Key constraints are not nullable. UNIQUE constraints may be nullable.

When you create a UNIQUE constraint, the database automatically creates a UNIQUE index. For MS SQL Server databases, a PRIMARY KEY will generate a unique CLUSTERED INDEX. A UNIQUE constraint will generate a unique NON-CLUSTERED INDEX.

**Primary Key:**  
Primary Key enforces uniqueness of the column on which they are defined. Primary Key creates a clustered index on the column. Primary Key does not allow Nulls.

*Create table with Primary Key:*  
CREATE TABLE Authors (  
AuthorID INT NOT NULL PRIMARY KEY,  
Name VARCHAR(100) NOT NULL  
)  
GO  
 *Alter table with Primary Key:*  
ALTER TABLE Authors  
ADD CONSTRAINT pk\_authors PRIMARY KEY (AuthorID)  
GO  
  
**Unique Key:**  
Unique Key enforces uniqueness of the column on which they are defined. Unique Key creates a non-clustered index on the column. Unique Key allows only one NULL Value.

*Alter table to add unique constraint to column:*  
ALTER TABLE Authors ADD CONSTRAINT IX\_Authors\_Name UNIQUE(Name)  
GO

**Primary key:**

1. Primary key is nothing but it uniquely identifies each row in a table.
2. Primary key does not allow duplicate values, nor NULL.
3. Primary key by default is a clustered index.
4. A table can have only one primary key.

**Unique Key:**

1. Unique key is nothing but it uniquely identifies each row in a table.
2. Unique key does not allow duplicate values, but it allows (at most one) NULL.
3. Unique key by default is a non-clustered index.
4. **What is the difference between a Star vs Snowflake schema**
   * Star Schema : Star Schema is a relational database schema for representing multimensional [datahttp://images.intellitxt.com/ast/adTypes/icon1.png](http://www.geekinterview.com/question_details/38599). It is the simplest form of data warehouse schema that contains one or more dimensions and fact tables. It is called a star schema because the entity-relationship diagram between dimensions and fact tables resembles a star where one fact table is connected to multiple dimensions. The center of the star schema consists of a large fact table and it points towards the dimension tables. The advantage of star schema are slicing down, performance increase and easy understanding of data.
   * Snowflake Schema : A snowflake schema is a term that describes a star schema structure normalized through the use of outrigger [tableshttp://images.intellitxt.com/ast/adTypes/icon1.png](http://www.geekinterview.com/question_details/38599). i.e dimension table hierachies are broken into simpler tables.
   * In a star schema every dimension will have a primary key.
   * In a star schema, a dimension table will not have any parent table
   * Whereas in a snow flake schema, a dimension table will have one or more parent tables.
   * Hierarchies for the dimensions are stored in the dimensional table itself in star schema.
   * Whereas hierachies are broken into separate tables in snow flake schema. These hierachies helps to drill down the data from topmost hierachies to the lowermost hierarchies.
5. **What is data modeling?**

A data model is a graphical view of data created for analysis and design purposes. Data modeling includes designing data warehouse databases in detail, it follows principles and patterns established in [Architecture for Data Warehousing and Business Intelligence](http://infogoal.com/datawarehousing/architecture.htm).

If you need to understand this subject from the beginning check the article, [Data Modeling Basics](http://infogoal.com/datawarehousing/data_modeling_basics.htm) to learn key terms and concepts.

|  |  |  |
| --- | --- | --- |
|  |  | Data warehouse modeling includes:   * Top Down / Requirements Driven Approach * Fact Tables and Dimension Tables * Multidimensional Model/Star Schema * Support Roll Up, Drill Down, and Pivot Analysis * Time Phased / Temporal Data * Operational Logical and Physical Data Models * Normalization and Denormalization * Model Granularity : Level of Detail |

Areas that require specialized patterns are:

* Data Mart / Frontroom - uses dimension modeling - the ROLAP star schema or the MOLAP cube
* Data warehouse / Backroom - uses normalized ERD
* Staging / Landing Area - looks like source system

### Data Modeling Tools

Data modeling involves visualizing data through use of graphical tools, so you will want to obtain a data modeling software package or use graphical capabilities in existing software. See the Data Management Center [Data Modeling Directory](http://infogoal.com/dmc/dmcdmd.htm) for a list of data modeling tools and other resources.

We have used open source software to develop examples for this article so that readers will able to learn on their own without licensing fees. Data models have been developed using DBDesigner4 from [FabForce.Net](http://infogoal.com/datawarehousing/fabforce.net). The target database is MySQL from [Sun Software](http://www.sun.com/).

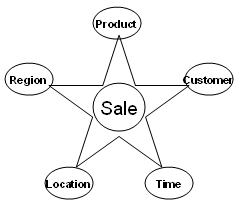
### Data Modeling for Business Intelligence

It is best to organize data to best meet the needs of its users.  Business intelligence commonly performs analytic operations on data such as:

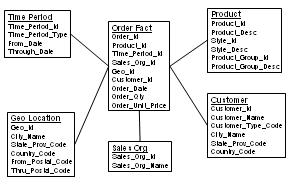
* Query by multiple criteria
* "Slice and dice"
* Drill Down
* Roll Up

The "Dimensional Data Model" otherwise known as the "Star Schema" was developed by Ralph Kimball in the 1980s to support these business needs.  This approach has stood the test of time and is the recommended way to organize data for business query and analysis.

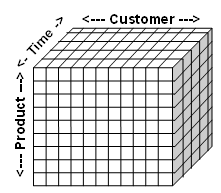
The two major table types of the Star Schema are the Fact and the Dimension.  The Fact contains quantitative measurements while the Dimension contains classification information.  Each Fact is surrounded by the Dimensions that provide context to it, given the appearance of a star.



The Order Fact with dimensions is a classic example.  In this case the Order Fact measurers order quantity and currency amount.  Dimensions of Calendar Date, Product, Customer, Geo Location and Sales Organization put the Order Fact into context.



This star schema supports looking orders like a cube, enabling slicing and dicing by customer, time and product.



### Surrogate Keys Improve Data Mart Efficiency and Performance

Surrogate keys, typical stored as integers, improve efficiency and increase performance.  Joins between facts and dimensions are faster with integers.  Indexes on integers are compact and provide rapid access.

1. **What is 1st Normal form**
   * Normalization is used for mainly two purposes
     + Eliminating redundant (useless) data
     + Ensuring data dependencies make sense and data is logically stored.

## 1st Normal Form Definition

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

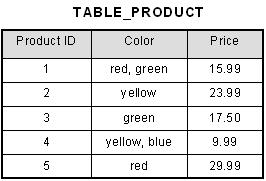
* Contains only atomic values
* There are no repeating groups

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

A repeating group means that a table contains two or more columns that are closely related. For example, a table that records data on a book and its author(s) with the following columns: [Book ID], [Author 1], [Author 2], [Author 3] is not in 1NF because [Author 1], [Author 2], and [Author 3] are all repeating the same attribute.

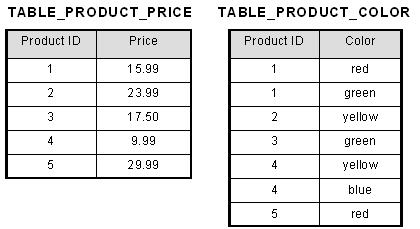
## 1st Normal Form Example

How do we bring an unnormalized table into first normal form? Consider the following example:



This table is not in first normal form because the [Color] column can contain multiple values. For example, the first row includes values "red" and "green."

To bring this table to first normal form, we split the table into two tables and now we have the resulting tables:



Now first normal form is satisfied, as the columns on each table all hold just one value.

1. **What is 2nd Normal form**

## 2nd Normal Form Definition

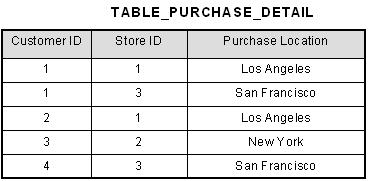
A database is in second normal form if it satisfies the following conditions:

* It is in first normal form
* All non-key attributes are fully functional dependent on the primary key

In a table, if attribute B is functionally dependent on A, but is not functionally dependent on a proper subset of A, then B is considered fully functional dependent on A. Hence, in a 2NF table, all non-key attributes cannot be dependent on a subset of the primary key. Note that if the primary key is not a composite key, all non-key attributes are always fully functional dependent on the primary key. **A table that is in 1st normal form and contains only a single key as the primary key is automatically in 2nd normal form.**

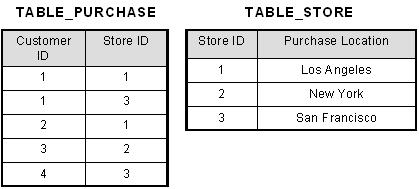
## 2nd Normal Form Example

Consider the following example:



This table has a composite primary key [Customer ID, Store ID]. The non-key attribute is [Purchase Location]. In this case, [Purchase Location] only depends on [Store ID], which is only part of the primary key. Therefore, this table does not satisfy second normal form.

To bring this table to second normal form, we break the table into two tables, and now we have the following:



What we have done is to remove the partial functional dependency that we initially had. Now, in the table [TABLE\_STORE], the column [Purchase Location] is fully dependent on the primary key of that table, which is [Store ID].

1. **What is 3rd Normal form**

## 3rd Normal Form Definition

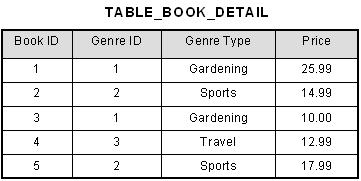
A database is in third normal form if it satisfies the following conditions:

* It is in second normal form
* There is no transitive functional dependency

By transitive functional dependency, we mean we have the following relationships in the table: A is functionally dependent on B, and B is functionally dependent on C. In this case, C is transitively dependent on A via B.

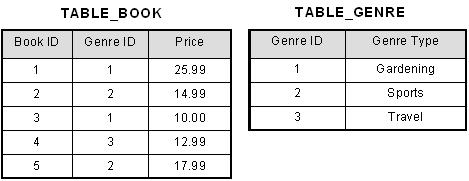
## 3rd Normal Form Example

Consider the following example:



In the table able, [Book ID] determines [Genre ID], and [Genre ID] determines [Genre Type]. Therefore, [Book ID] determines [Genre Type] via [Genre ID] and we have transitive functional dependency, and this structure does not satisfy third normal form.

To bring this table to third normal form, we split the table into two as follows:



Now all non-key attributes are fully functional dependent only on the primary key. In [TABLE\_BOOK], both [Genre ID] and [Price] are only dependent on [Book ID]. In [TABLE\_GENRE], [Genre Type] is only dependent on [Genre ID].

**SSRS QUESTIONS**

1. **What is the difference between a Matrix and a Tablix?**
   * Matrix allows you to group by rows and columns
   * Tablix allows you to combine the Table, Matrix and List together
2. **What version of SSRS has a bug when putting two Data Regions in one report.**
3. **What is the difference between a report parameter and a query parameter.**

Query parameters are written against a data source.They can limit data retrieved from the data source. These are generally written in your SQL statement or SP written in the dataset.

Report parameters are the filters created in your report. These are applied on the data fetched from the data source. They are used to filter the report data.

1. **How would you pass multi valued parameters to a stored procedure.**
2. **What are cascading parameters**
   * One parameter is dependent on another parameter.
3. **What is the .rdl file**

**SSIS QUESTIONS**

**SSAS QUESTIONS**