<http://venkatsqlinterview.blogspot.com/2011/05/sql-server-interview-questions-by-topic.htm>

[Basic SQL Server Interview Questions](http://venkatsqlinterview.blogspot.com/2011/05/basic-sql-server-interview-questions.html)

**Explain DML, DDL, DCL and TCL statements with examples?**  
**DML**: DML stands for Data Manipulation Language. DML is used to retrieve, store, modify, delete, insert and update data in database.  
**Examples**of DML statements: SELECT, UPDATE, INSERT, DELETE statements.  
  
  
**DDL**: DDL stands for Data Definition Language. DDL is used to create and modify the structure of database objects.  
  
**Examples**: CREATE, ALTER, DROP statements.  
  
  
**DCL**: DCL stands for Data Control Language. DCL is used to create roles, grant and revoke permissions, establish referential integrity etc.  
**Examples**: GRANT, REVOKE statements  
  
  
**TCL**: TCL stands for Transactional Control Language. TCL is used to manage transactions within a database.  
**Examples**: COMMIT, ROLLBACK statements

**What is the difference between Drop, Delete and Truncate statements in SQL Server?**  
**Drop, Delete and Truncate** - All operations can be rolled back.  
All the statements (**Delete, Truncate and Drop**) are logged operations, but the amount of information that is logged varies. **Delete**statement logs an entry in the transaction log for each deleted row, where as **Truncate Table**logs only the Page deallocations. **Hence, truncate is a little faster than Delete**.   
  
You can have a where clause in Delete statement where as Truncate statement cannot have a where clause. Truncate will delete all the rows in a Table, but the structure of the table remains. Drop would delete all the rows including the structure of the Table.  
  
  
Please refer to the screen shot below for the differences summary snapshot between Drop, Delete and Truncate statements in SQL Server.

|  |
| --- |
| http://1.bp.blogspot.com/-D6vcPgujroo/TffVCroxJPI/AAAAAAAAAHY/Y1jlyzR5klU/s400/Drop-Delete-Truncate.png |

**What is Cascading referential integrity constraint?**  
**Cascading referential integrity constraints** allow you to define the actions Microsoft SQL Server should take when a user attempts to delete or update a key to which an existing foreign keys point.  
 **You can instruct SQL Server to do the following:**  
  
**1. No Action:** This is the default behavior. No Action specifies that if an attempt is made to delete or update a row with a key referenced by foreign keys in existing rows in other tables, an error is raised and the DELETE or UPDATE is rolled back.  
  
**2. Cascade:** Specifies that if an attempt is made to delete or update a row with a key referenced by foreign keys in existing rows in other tables, all rows containing those foreign keys are also deleted or updated.  
  
**3. Set NULL:** Specifies that if an attempt is made to delete or update a row with a key referenced by foreign keys in existing rows in other tables, all rows containing those foreign keys are set to NULL.    
  
**4. Set Default:** Specifies that if an attempt is made to delete or update a row with a key referenced by foreign keys in existing rows in other tables, all rows containing those foreign keys are set to default values.

**Difference between primary key and unique key in SQL Server?**  
1. A table can have only one primary key. On the other hand a table can have more than one unique key.  
2. Primary key column does not accept any null values, where as a unique key column accept one null value.

[SQL Server Interview Questions on Temporary Tables](http://venkatsqlinterview.blogspot.com/2011/05/sql-server-interview-questions-on_2213.html)

**What are the 2 types of Temporary Tables in SQL Server?**  
**1.** Local Temporary Tables  
**2.** Global Temporary Tables  
  
**What is the difference between Local and Global Temporary Tables?**  
**Local Temporary Tables:**  
**1.** Prefixed with a single pound sign (#).   
**2.** Local temporary tables are visible to that session of SQL Server which has created it.   
**3.** Local temporary tables are automatically dropped, when the session that created the temporary tables is closed.  
  
**Global Temporary Tables:**  
**1.** Prefixed with two pound signs (##).   
**2.** Global temporary tables are visible to all the SQL server sessions.   
**3.** Global temporary tables are also automatically dropped, when the session that created the temporary tables is closed.  
  
  
**Can you create foreign key constraints on temporary tables?**  
No  
  
  
**Do you have to manually delete temporary tables?**  
No, temporary tables are automatically dropped, when the session that created the temporary tables is closed. But if you maintain a persistent connection or if connection pooling is enabled, then it is better to explicitly drop the temporary tables you have created.  
However, It is generally considered a good coding practice to explicitly drop every temporary table you create.

**In which database, the temporary tables get created?**  
TEMPDB database.  
  
  
**How can I check for the existence of a temporary table?**

|  |
| --- |
| http://1.bp.blogspot.com/-qcC6Qrb2e88/TcUcWrXtkPI/AAAAAAAAAEw/shlygQ7T88Y/s1600/Temporary+Table.png |

[SQL Server Interview Questions on Indexes - Part 1](http://venkatsqlinterview.blogspot.com/2011/05/sql-server-interview-questions-on.html)

**What is the use of an Index in SQL Server?**  
Relational databases like SQL Server use indexes to find data quickly when a query is processed. Creating the proper index can drastically increase the performance of an application.

**What is a table scan?**  
or   
**What is the impact of table scan on performance?**  
When a SQL Server has no index to use for searching, the result is similar to the reader who looks at every page in a book to find a word. The SQL engine needs to visit every row in a table. In database terminology we call this behavior a table scan, or just scan. A full table scan of a very large table can adversely affect the performance. Creating proper indexes will allow the database to quickly narrow in on the rows to satisfy the query, and avoid scanning every row in the table.   
  
  
**What is the system stored procedure that can be used to list all the indexes that are created for a specific table?**  
**sp\_helpindex** is the system stored procedure that can be used to list all the indexes that are created for a specific table.   
  
For example, to list all the indexes on table **tblCustomers**, you can use the following command.  
**EXEC sp\_helpindex tblCustomers**

[Example:](https://msdn.microsoft.com/en-us/library/ms188771.aspx)  
USE AdventureWorks2012;

GO

EXEC sp\_helpindex N'Sales.Customer';

GO

**What is the purpose of query optimizer in SQL Server?**  
An important feature of SQL Server is a component known as the query optimizer. The query optimizer's job is to find the fastest and least resource intensive means of executing incoming queries. An important part of this job is selecting the best index or indexes to perform the task.  
  
  
**What is the first thing you will check for, if the query below is performing very slow?**  
SELECT \* FROM tblProducts ORDER BY UnitPrice ASC  
  
  
Check if there is an Index created on the UntiPrice column used in the ORDER BY clause. An index on the UnitPrice column can help the above query to find data very quickly. When we ask for a sorted data, the database will try to find an index and avoid sorting the results during execution of the query. We control sorting of a data by specifying a field, or fields, in an ORDER BY clause, with the sort order as ASC (ascending) or DESC (descending).   
  
  
With no index, the database will scan the **tblProducts** table and sort the rows to process the query. However, if there is an index, it can provide the database with a presorted list of prices. The database can simply scan the index from the first entry to the last entry and retrieve the rows in sorted order.   
  
  
The same index works equally well with the following query, simply by scanning the index in reverse.  
SELECT \* FROM tblProducts ORDER BY UnitPrice DESC

**What is the significance of an Index on the column used in the GROUP BY clause?**  
Creating an Index on the column, that is used in the **GROUP BY** clause, can greatly improve the performance. We use a **GROUP BY** clause to group records and aggregate values, for example, counting the number of products with the same UnitPrice. To process a query with a GROUP BY clause, the database will often sort the results on the columns included in the GROUP BY.   
  
  
The following query counts the number of products at each price by grouping together records with the same UnitPrice value.  
SELECT UnitPrice, Count(\*) FROM tblProducts GROUP BY UnitPrice   
  
  
The database can use the index (Index on **UNITPRICE** column) to retrieve the prices in order. Since matching prices appear in consecutive index entries, the database is able to count the number of products at each price quickly. Indexing a field used in a **GROUP BY** clause can often speed up a query.  
  
  
**What is the role of an Index in maintaining a Unique column in table?**  
Columns requiring unique values (such as primary key columns) must have a unique index applied. There are several methods available to create a unique index.   
**1.** Marking a column as a **primary key** will automatically create a unique index on the column.  
**2.** We can also create a unique index by checking the Create UNIQUE checkbox when creating the index graphically.   
**3.** We can also create a unique index using SQL with the following command:   
CREATE UNIQUE INDEX IDX\_ProductName On Products (ProductName)  
  
  
The above SQL command will not allow any duplicate values in the ProductName column, and an index is the best tool for the database to use to enforce this rule. Each time an application adds or modifies a row in the table, the database needs to search all existing records to ensure none of values in the new data duplicate existing values.

<https://msdn.microsoft.com/en-us/library/ms187019.aspx>

**PRIMARY KEY or UNIQUE constraint**

When you create a PRIMARY KEY constraint, a unique clustered index on the column or columns is automatically created if a clustered index on the table does not already exist and you do not specify a unique non-clustered index. The primary key column cannot allow NULL values.

When you create a UNIQUE constraint, a unique non-clustered index is created to enforce a UNIQUE constraint by default. You can specify a unique clustered index if a clustered index on the table does not already exist.

For more information, see [Unique Constraints and Check Constraints](https://msdn.microsoft.com/en-us/library/ms187550.aspx) and [Primary and Foreign Key Constraints](https://msdn.microsoft.com/en-us/library/ms179610.aspx).

[SQL Server Interview Questions on Indexes - Part 2](http://venkatsqlinterview.blogspot.com/2011/05/sql-server-interview-questions-on_07.html)

**What are the disadvantages of an Index?**  
There are 2 disadvantages of an Index  
**1.**Increased Disk Space  
**2.** Insert, Update and Delete statements could be slow. In short, all DML statements could be slow.  
  
  
**Disk Space:** Indexes are stored on the disk, and the amount of space required will depend on the size of the table, and the number and types of columns used in the index. Disk space is generally cheap enough to trade for application performance, particularly when a database serves a large number of users.   
  
  
**Insert, Update and Delete statements could be slow:** Another downside to using an index is the performance implication on data modification statements. Any time a query modifies the data in a table (INSERT, UPDATE, or DELETE), the database needs to update all of the indexes where data has changed. Indexing can help the database during data modification statements by allowing the database to quickly locate the records to modify, however, providing too many indexes to update can actually hurt the performance of data modifications. This leads to a delicate balancing act when tuning the database for performance.  
  
  
**What are the 2 types of Indexes in SQL Server?**  
**1.** Clustered Index   
**2.** Non Clustered Index  
  
  
**How many Clustered and Non Clustered Indexes can you have per table?**  
**Clustered Index** - Only **one** Clustered Index per table. A clustered index contains all of the data for a table in the index, sorted by the index key. Phone Book is an example for Clustered Index.  
**Non Clustered Index**- You can have multiple Non Clustered Indexes per table. Index at the back of a book is an example for Non Clustered Index.  
  
  
**Which Index is faster, Clustered or Non Clustered Index?**  
**Clustered Index**is slightly faster than Non Clustered Index. This is because, when a Non Clustered Index is used there is an extra look up from the Non Clustered Index to the table, to fetch the actual rows.

**When is it usually better to create a unique non-clustered index on the primary key column?**  
Sometimes it is better to use a unique non-clustered index on the primary key column, and place the clustered index on a column used by more queries. For example, if the majority of searches are for the price of a product instead of the primary key of a product, the clustered index could be more effective if used on the price field.  
  
  
**What is a Composite Index in SQL Server?**  
or   
**What is the advantage of using a Composite Index in SQL Server?**  
or   
**What is Covering Query?**  
**A composite index**is an index on two or more columns. Both clustered and non-clustered indexes can be composite indexes.

If all of the information for a query can be retrieved from an Index. A clustered index, if selected for use by the query optimizer, always covers a query, since it contains all of the data in a table.  
  
By creating a composite indexes, we can have covering queries.

[What is the difference between a Temporary Table and a Table Variable](http://venkatsqlinterview.blogspot.com/2011/05/what-is-difference-between-temporary.html)

**What is the difference between a Temporary Table and a Table Variable?**  
Or  
**When do you use Table Variable over a Temporary Table and vice versa?**  
  
  
**1.** Table variable is created in the memory where as a temporary table is created in the TempDB. But, if there is a memory pressure, the pages belonging to a table variable may be pushed out to tempdb.  
  
**2.** Table variables cannot be involved in transactions, logging or locking. This makes table variable faster than a temporary table.  
  
**3.** You can pass table variable as parameter to functions and stored procedures, where as you cannot do the same with temporary table.  
  
**4.** A temporary table can have indexes, whereas a table variable can only have a primary index. If speed is an issue Table variables can be faster, but if there are a lot of records, or there is a need to search the temporary table based on a clustered index, then a Temporary Table would be better. 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.

[What is the use of COALESCE in SQL Server](http://venkatsqlinterview.blogspot.com/2011/05/what-is-use-of-coalesce-in-sql-server.html)

Let us understand the use of **COALESCE** with the help of an example.  
  
In this example, the **Candidate** table is shown to include three columns with information about a Candidate:   
1. Candidate\_id  
2. PrimaryEmail  
3. SecondaryEmail  
  
  
**COALESCE** in the SELECT statement below, selects the PrimaryEmail if it is not null. If the PrimaryEmail is null then SecondaryEmail will be selected. If both PrimaryEmail and SecondaryEmail is present then only PrimaryEmail is selected. So, COALESCE returns the first nonnull column among the list of columns passed. If both PrimaryEmail and SecondaryEmail is NULL, COALESCE returns NULL.

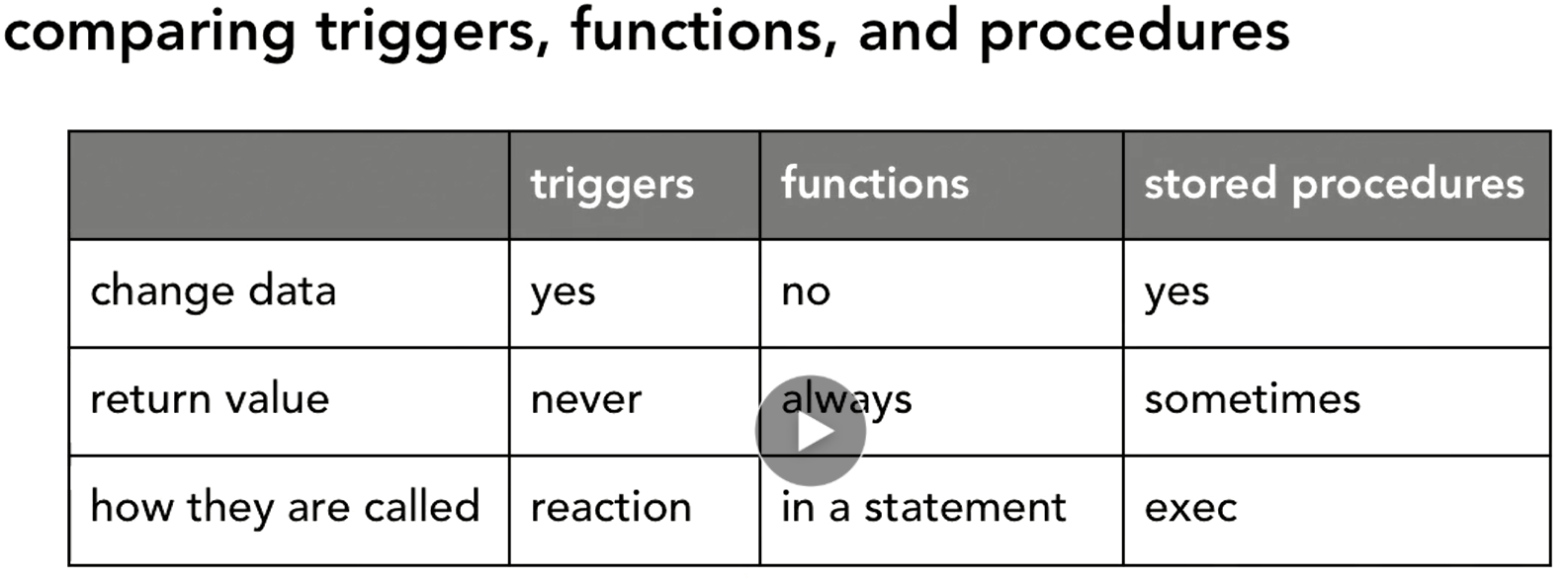
|  |
| --- |
| http://1.bp.blogspot.com/-6R49zlPNh6M/TchOH97_qhI/AAAAAAAAAE0/kDW8rujbq08/s1600/coalesce.png |

**COALESCE** can also be used in joins as shown in the example below. If the Candidate table has a non null value in the Email column, then the value is selected. If the Email column is null in the Candidate Table then, CompanyEmail from CandidateCompany Table is selected.

|  |
| --- |
| http://1.bp.blogspot.com/-OcOSIklGodw/TchPPz2q8-I/AAAAAAAAAE4/19Dc22lIH4o/s1600/coalesce2.jpg |

[SQL Server Interview Questions on triggers](http://venkatsqlinterview.blogspot.com/2011/05/sql-server-interview-questions-on_10.html)

**What is a Trigger in SQL Server?**  
A Trigger is a database object that is attached to a table. In many aspects it is similar to a stored procedure. As a matter of fact, triggers are often referred to as a "special kind of stored procedure." The main difference between a trigger and a stored procedure is that the former is attached to a table and is only fired when an INSERT, UPDATE or DELETE occurs.

  
  
**What are the two types of Triggers in SQL Server?**  
**1. After Triggers:**Fired after Insert, Update and Delete operations on a table.  
**2. Instead of Triggers:**Fired instead of Insert, Update and Delete operations on a table.  
  
  
**What are the special tables used by Triggers in SQL Server?**  
**Triggers**make use of two special tables called **inserted** and **deleted**. The **inserted**table contains the data referenced in an INSERT before it is actually committed to the database. The **deleted**table contains the data in the underlying table referenced in a DELETE before it is actually removed from the database. When an UPDATE is issued both tables are used. More specifically, the new data referenced in the UPDATE statement is contained in **inserted**table and the data that is being updated is contained in **deleted**table.  
  
  
**Give a real time example for triggers usage?**  
It is recommended to avoid triggers in a real time environment. There is one scenario I can think of why you may want to use triggers in a real time environment. Let us use an example to understand this.

I have 2 tables, **tblPerson**and **tblGender** as shown below. **GenderId**is the foriegn key in **tblPerson**table.

|  |
| --- |
| http://2.bp.blogspot.com/-Ugc6i7kn2hI/TcmlRO6JWJI/AAAAAAAAAE8/I5hgINVEowM/s1600/Tables.png |

Now create a view based on the above 2 tables as shown below.

|  |
| --- |
| http://4.bp.blogspot.com/-vig9OaUW32U/TcmmtknmgqI/AAAAAAAAAFA/xwV7kZdFlDI/s1600/ViewOnMultipleTables.png |

Select \* from **vWPersons** will give us the result as shown below.

|  |
| --- |
| http://4.bp.blogspot.com/-zCCwM2LHGo8/Tcmnm0FkcvI/AAAAAAAAAFE/t34g_SSxPz0/s1600/ViewResult.png |

Now update the view the following query. This will change the Gender Text to **Female** in **tblGender**table for Id = 1. This is not what we have expected.  
  
Update tblGender Set Gender='Female' where Id=1  
  
  
The base tables are updated incorrectly. So, Select \* from **vWPersons** will give us the result as shown below.

|  |
| --- |
| http://4.bp.blogspot.com/-qa8L2rHTIAk/Tcmow25J3qI/AAAAAAAAAFI/wbGeOtr13fw/s1600/IncorrectResult.png |

To update the base tables correctly, you can create an **INSTEAD OF** trigger on the view as shown below.

|  |
| --- |
| http://1.bp.blogspot.com/-yQLK8OYwKpo/TcmpbZoHRqI/AAAAAAAAAFM/Zy7GTDaCUQ0/s1600/InsteadOfTrigger.png |

Now run the query below which will update the underlying base tables correctly.  
Update vWPersons Set Gender='Female' Where Id=1  
  
  
Select \* from vWPersons, will show the correct result set as shown below. The **INSTEAD OF** trigger has correctly updated the GenderId in tblPerson table.

|  |
| --- |
| http://3.bp.blogspot.com/-arYuQjIVnW0/TcmqaLRJeFI/AAAAAAAAAFQ/vrBpSMJUOHU/s1600/CorrectResult.png |

So, **Instead of triggers**can be used to facilitate **updating Views that are based on multiple base tables**.

[Difference between User Defined Function and Stored Procedure](http://venkatsqlinterview.blogspot.com/2011/05/what-is-difference-between-user-defined.html)

**1. Stored Procedure**support differed name resolution whereas **functions**do not support differed name resolution.   
  
  
**2. User Defined Function**can be used in a select statement where as you cannot use a **stored procedure**in a select statement.   
  
  
**3. UDF**'s cannot return Image, Text where as a **Stored Procedure** can return any datatype.   
  
  
**4.** In general, **User Defined Functions**are used for computations whereas **Stored Procedures**are used for performing business logic.   
  
  
**5. UDF**should return a value whereas **Stored Procedure**need not.   
   
**6. User Defined Functions**accept lesser number of input parameters than **Stored Procedures. UDF**can have up to 1023 input parameters where as a **Stored Procedure**can have up to 21000 input parameters.

**7.** Temporary Tables cannot be used in a **UDF**whereas a **Stored Procedure**can use Temporary Tables.   
  
  
**8. UDF** cannot Execute Dynamic SQL where as a **Stored Procedure**can execute Dynamic SQL.   
  
  
**9. User Defined Function**does not support error handling whereas **Stored Procedure**support error handling. RAISEERROR or @@ERROR are not allowed in **UDF**s.

[SQL Server Interview Questions on Views - Part 1](http://venkatsqlinterview.blogspot.com/2011/05/sql-server-interview-questions-on-views.html)

**What is a View in SQL Server?**  
You can think of a **view** either as a **compiled sql query**or a **virtual**table. As a view represents a virtual table, it does not physically store any data. When you query a view, you actually retrieve the data from the underlying base tables.  
  
**What are the advantages of using views?**  
Or   
**When do you usually use views?**  
**1.** Views can be used to implement row level and column level security.  
  
**Example 1:** Consider the **tblEmployee** table below. I don't want some of the users to have access to the salary column, but they should still be able to access ID, NAME and DEPT columns. If I grant access to the table, the users will be able to see all the columns. So, to achieve this, I can create a view as shown in **Listing 1** below. Now, grant access to the view and not the table. So using views we can provide **column level security**.

|  |
| --- |
| **tblEmployee**  http://4.bp.blogspot.com/-M6kUTahKBuM/Tcwedojw4nI/AAAAAAAAAFU/bHEA1r5iRHo/s1600/ViewBaseTable.png |

|  |
| --- |
| **Listing 1** Create View      vWEmployee As Select               ID, Name, Dept From                 tblEmployee |

**Example 2:** Let us say, we have a few users who should be able to access only **IT employee** details and not any other dept. To do this, I can create a view as shown in **Listing 2** below. Now, grant access only to the view and not the table. So using views we can provide row level security as well.  
  
**Listing 2**  
Create View     vWITEmployees  
As  
Select              ID, Name, Dept  
From                tblEmployee  
Where              Dept = 'IT'  
  
**2.** Simplify the database schema to the users. You can create a view based on multiple tables which join columns from all these multiple tables so that they look like a single table.  
  
**3.** Views can be used to present **aggregated**and **summarized**data.  
  
**Example 1:** Consider the **tblEmployee** table above. I want to aggregate the data as shown in the image below. To do this I can create a view as shown in **Listing 3**. Now, you can simply issue a select query against the view rather than writing a complex query every time you want to retrieve the aggregated data.

|  |
| --- |
| http://4.bp.blogspot.com/-3gjWYSDkjh4/TcwieMlRYLI/AAAAAAAAAFY/aUkR3qMY-bQ/s1600/GroupedData.png  **Listing 3** Select        Dept, Count(\*) As Total From          tblEmployee Group By   Dept |

[SQL Server Interview Questions on Views - Part 2](http://venkatsqlinterview.blogspot.com/2011/05/sql-server-interview-questions-on-views_16.html)

**Can you create a view based on other views?**  
**Yes**, you can create a view **based on other views**. Usually we create views based on tables, but it also possible to create views based on views.  
  
**Can you update views?**  
**Yes**, views can be updated. However, updating a view that is based on multiple tables, may not update the underlying tables **correctly**. To correctly update a view that is based on multiple tables you can make use **INSTEAD OF triggers**in SQL Server. [Click here for a real time example, that we have already discussed in SQL Server Interview Questions on triggers article.](http://venkatsqlinterview.blogspot.com/2011/05/sql-server-interview-questions-on_10.html)

**What are indexed views?**  
Or  
**What are materialized views?**  
  
A view is a virtual table, it does not contain any physical data. A view is nothing more than compiled SQL query. Every time, we issue a select query against a view, we actually get the data from the underlying base tables and not from the view, as the view itself does not contain any data.  
  
When you create an index on a view, the data gets physically stored in the view. So, when we issue a select query against an indexed view, the data is retrieved from the index without having to go to the underlying table, which will make the select statement to work slightly faster. However, the disadvantage is, INSERT, UPDATE and DELETE operations will become a little slow, because every time you insert or delete a row from the underlying table, the view index needs to be updated. In short, DML operations will have negative impact on performance.  
  
Oracle refers to indexed views as materialized views.  
  
Only the views created with schema binding, can have an Index. Simply adding WITH SCHEMABINDING to the end of the CREATE VIEW statement will accomplish this. However, the effect is that any changes to the underlying tables which will impact the view are not allowed. Since the indexed view is stored physically, any schema changes would impact the schema of the stored results set. Therefore, SQL Server requires that schema binding be used to prevent the view's schema (and therefore the underlying tables) from changing.  
  
The first index for a view must be a UNIQUE CLUSTERED INDEX, after which, it's possible to create non-clustered indexes against the view.  
  
Indexed Views are heavily used in data warehouses and reporting databases that are not highly transactional.  
  
**What are the limitations of a View?**  
**1.** You cannot pass parameters to a view.  
  
**2.** Rules and Defaults cannot be associated with views.  
  
**3.** The ORDER BY clause is invalid in views unless TOP or FOR XML is also specified.  
  
**4.** Views cannot be based on temporary tables.

[Basic SQL Server Interview Questions on Joins](http://venkatsqlinterview.blogspot.com/2011/05/basic-sql-server-interview-questions-on.html)

**What are the different types of joins available in sql server?**  
There are 3 different types of joins available in sql server, and they are  
**1.** Cross Join   
**2.** Inner Join or Join   
**3.** Outer Join  
  
**Outer Join is again divided into 3 types as shown below.**  
**1.** Left Outer Join or Left Join   
**2.** Right Outer Join or Right Join   
**3.** Full Outer Join or Full Join   
  
You might have heard about self join, but self join is not a different type of join. A self join means joining a table with itself. We can have an inner self join or outer self join. Read this sql server interview question, to understand self join in a greater detail.

**What is cross join. Explain with an example?**  
Let us understand Cross Join with an example. Create 2 tables **Company** and **Candidate**. Use the script below to create these tables and populate them. CompanyId column in Candidate Table is a foreign key referencing CompanyId in Company Table.  
  
**CREATE TABLE Company**  
**(**  
**CompanyId TinyInt Identity Primary Key,**  
**CompanyName Nvarchar(50) NULL**  
**)**  
**GO**  
  
**INSERT Company VALUES('DELL')**  
**INSERT Company VALUES('HP')**  
**INSERT Company VALUES('IBM')**  
**INSERT Company VALUES('Microsoft')**  
**GO**  
  
**CREATE TABLE Candidate**  
**(**  
**CandidateId tinyint identity primary key,**  
**FullName nvarchar(50) NULL,**  
**CompanyId tinyint REFERENCES Company(CompanyId)**  
**)**  
**GO**  
  
**INSERT Candidate VALUES('Ron',1)**  
**INSERT Candidate VALUES('Pete',2)**  
**INSERT Candidate VALUES('Steve',3)**  
**INSERT Candidate VALUES('Steve',NULL)**  
**INSERT Candidate VALUES('Ravi',1)**  
**INSERT Candidate VALUES('Raj',3)**  
**INSERT Candidate VALUES('Kiran',NULL)**  
**GO**  
  
A cross join produces the Cartesian product of the tables involved in the join. The size of a Cartesian product result set is the number of rows in the first table multiplied by the number of rows in the second table. A query involving a CROSS JOIN for the **Candidate** and **Company**Table is shown below.  
  
**SELECT  Cand.CandidateId,Cand.FullName,Cand.CompanyId, Comp.CompanyId,Comp.CompanyName**  
**FROM Candidate Cand**  
**CROSS JOIN Company Comp**  
  
**If we run the above query, we produce the result set shown in the image below.**

|  |
| --- |
| http://4.bp.blogspot.com/-1v1yCC3atic/TdLLTiUBP8I/AAAAAAAAAFc/yyy3OdmD5Jk/s1600/CrossJoin.png |

**Key Points to remember about CROSS JOIN.**   
**1.** A cross join produces the Cartesian product of the tables involved in the join. This mean every row in the Left Table is joined to every row in the Right Table. **Candidate** is LEFT Table and **Company** is RIGHT Table. In our example we have 28 total number of rows in the result set. 7 rows in the Candidate table multiplied by 4 rows in the Company Table.   
  
**2.** In real time scenarios we rarely use CROSS JOIN. Most often we use either INNER JOIN or LEFT OUTER JOIN.   
  
**3.** CROSS JOIN does not have an ON clause with a Join Condition. All the other JOINS use ON clause with a Join Condition.   
  
**4.** Using an ON clause on a CROSS JOIN would generate a syntax error.

[Explain Inner Join with an example](http://venkatsqlinterview.blogspot.com/2011/05/inner-join-with-example.html)

**Inner Join and left join**are the most commonly used joins in real time projects. We will talk about left join in a later article. Now, let us understand **Inner join with an example.**  
  
Create 2 tables **Company**and **Candidate**. Use the script below to create these tables and populate them. CompanyId column in Candidate Table is a foreign key referencing CompanyId in Company Table.  
  
**CREATE TABLE Company**  
**(**  
**CompanyId TinyInt Identity Primary Key,**  
**CompanyName Nvarchar(50) NULL**  
**)**  
**GO**  
  
**INSERT Company VALUES('DELL')**  
**INSERT Company VALUES('HP')**  
**INSERT Company VALUES('IBM')**  
**INSERT Company VALUES('Microsoft')**  
**GO**  
  
**CREATE TABLE Candidate**  
**(**  
**CandidateId tinyint identity primary key,**  
**FullName nvarchar(50) NULL,**  
**CompanyId tinyint REFERENCES Company(CompanyId)**  
**)**  
**GO**  
  
**INSERT Candidate VALUES('Ron',1)**  
**INSERT Candidate VALUES('Pete',2)**  
**INSERT Candidate VALUES('Steve',3)**  
**INSERT Candidate VALUES('Steve',NULL)**  
**INSERT Candidate VALUES('Ravi',1)**  
**INSERT Candidate VALUES('Raj',3)**  
**INSERT Candidate VALUES('Kiran',NULL)**  
**GO**

If you want to select all the rows from the **LEFT table** (In our example **Candidate Table**) that have a non null foreign key value(CompanyId in**Candidate Table**is the foreign key) then we use **INNER JOIN**. A query involving an **INNER JOIN**for the **Candidate and Company**Table is shown below.   
  
**SELECT Cand.CandidateId, Cand.FullName, Cand.CompanyId, Comp.CompanyId, Comp.CompanyName**  
**FROM Candidate Cand**  
**INNER JOIN Company Comp**  
**ON Cand.CompanyId = Comp.CompanyId**  
  
If we run the above query the output will be as shown in the image below. If you look at the output, we only got 5 rows. We did not get the 2 rows which has NULL value in the CompanyId column. So an INNER JOIN would get all the rows from the LEFT Table that has non null foreign key value.

|  |
| --- |
| http://3.bp.blogspot.com/-QVR5dY6zHWI/TdQlpUaVEGI/AAAAAAAAAFg/WT0wMHOByWM/s1600/InnerJoin.png  **Inner Join Result** |

Instead of using **INNER JOIN**keyword we can just use **JOIN**keyword as shown below. **JOIN**or **INNER JOIN**means the same.  
  
**SELECT Cand.CandidateId, Cand.FullName, Cand.CompanyId, Comp.CompanyId, Comp.CompanyName**  
**FROM Candidate Cand**  
**JOIN Company Comp**  
**ON Cand.CompanyId = Comp.CompanyId**

[Explain Left Outer Join with an example](http://venkatsqlinterview.blogspot.com/2011/05/left-join-with-example.html)

**Inner Join and left join**are the most commonly used joins in real time projects. [Click here to read about Inner Join in SQL Server](http://venkatsqlinterview.blogspot.com/2011/05/inner-join-with-example.html). Now, let us understand **Left join with an example.**  
  
Create 2 tables **Company**and **Candidate**. Use the script below to create these tables and populate them. CompanyId column in Candidate Table is a foreign key referencing CompanyId in Company Table.  
  
**CREATE TABLE Company**  
**(**  
**CompanyId TinyInt Identity Primary Key,**  
**CompanyName Nvarchar(50) NULL**  
**)**  
**GO**  
  
**INSERT Company VALUES('DELL')**  
**INSERT Company VALUES('HP')**  
**INSERT Company VALUES('IBM')**  
**INSERT Company VALUES('Microsoft')**  
**GO**  
  
**CREATE TABLE Candidate**  
**(**  
**CandidateId tinyint identity primary key,**  
**FullName nvarchar(50) NULL,**  
**CompanyId tinyint REFERENCES Company(CompanyId)**  
**)**  
**GO**  
  
**INSERT Candidate VALUES('Ron',1)**  
**INSERT Candidate VALUES('Pete',2)**  
**INSERT Candidate VALUES('Steve',3)**  
**INSERT Candidate VALUES('Steve',NULL)**  
**INSERT Candidate VALUES('Ravi',1)**  
**INSERT Candidate VALUES('Raj',3)**  
**INSERT Candidate VALUES('Kiran',NULL)**  
**GO**

If you want to select all the rows from the **LEFT table**( In our example **Candidate** Table ) including the rows that have a null foreign key value ( CompanyId in **Candidate**Table is the foreign key ) then we use **LEFT OUTER JOIN**. A query involving a **LEFT OUTER JOIN**for the **Candidate and Company**Table is shown below.  
  
**SELECT Cand.CandidateId, Cand.FullName, Cand.CompanyId, Comp.CompanyId, Comp.CompanyName**  
**FROM Candidate Cand**  
**LEFT OUTER JOIN Company Comp**  
**ON Cand.CompanyId = Comp.CompanyId**  
  
If we run the above query the output will be as shown in below. If you look at the output, we now got all 7 rows (All the rows from the**Candidate** Table) including the row that has a null value for the **CompanyId** column in the **Candidate**Table. So, **LEFT OUTER JOIN**would get all the rows from the **LEFT Table**including the rows that has null foreign key value.

|  |
| --- |
| <http://2.bp.blogspot.com/-VeMdJ7nvrZ4/TdbA57vfNkI/AAAAAAAAAFk/wQC8W0iTaSg/s1600/LeftJoin.png>  **Left Join Result** |

Instead of using **LEFT OUTER JOIN**keyword we can just use **LEFT JOIN**keyword as shown below. **LEFT OUTER JOIN or LEFT JOIN** means the same.  
  
**SELECT Cand.CandidateId, Cand.FullName, Cand.CompanyId, Comp.CompanyId, Comp.CompanyName**  
**FROM Candidate Cand**  
**LEFT JOIN Company Comp**  
**ON Cand.CompanyId = Comp.CompanyId**

[Explain Right Outer Join with an example](http://venkatsqlinterview.blogspot.com/2011/05/right-outer-join-with-example.html)

**Inner Join and left join**are the most commonly used joins in real time projects.   
  
[Click here to read about Inner Join in SQL Server](http://venkatsqlinterview.blogspot.com/2011/05/inner-join-with-example.html).   
[Click here to read about Left Outer Join in SQL Server.](http://venkatsqlinterview.blogspot.com/2011/05/left-join-with-example.html)  
  
Now, let us understand **Right Outer join with an example.**  
  
Create 2 tables **Company**and **Candidate**. Use the script below to create these tables and populate them. CompanyId column in Candidate Table is a foreign key referencing CompanyId in Company Table.  
  
**CREATE TABLE Company**  
**(**  
**CompanyId TinyInt Identity Primary Key,**  
**CompanyName Nvarchar(50) NULL**  
**)**  
**GO**  
  
**INSERT Company VALUES('DELL')**  
**INSERT Company VALUES('HP')**  
**INSERT Company VALUES('IBM')**  
**INSERT Company VALUES('Microsoft')**  
**GO**  
  
**CREATE TABLE Candidate**  
**(**  
**CandidateId tinyint identity primary key,**  
**FullName nvarchar(50) NULL,**  
**CompanyId tinyint REFERENCES Company(CompanyId)**  
**)**  
**GO**  
  
**INSERT Candidate VALUES('Ron',1)**  
**INSERT Candidate VALUES('Pete',2)**  
**INSERT Candidate VALUES('Steve',3)**  
**INSERT Candidate VALUES('Steve',NULL)**  
**INSERT Candidate VALUES('Ravi',1)**  
**INSERT Candidate VALUES('Raj',3)**  
**INSERT Candidate VALUES('Kiran',NULL)**  
**GO**

If you want to select all the rows from the **LEFT Table**( In our example **Candidate**Table) that have non null foreign key values plus all the rows from the **RIGHT table**( In our example **Company**Table) including the rows that are not referenced in the **LEFT Table**, then we use **RIGHT OUTER JOIN**. A query involving a **RIGHT OUTER JOIN**for the **Candidate**and **Company Table**is shown below.  
  
**SELECT Cand.CandidateId, Cand.FullName, Cand.CompanyId, Comp.CompanyId, Comp.CompanyName**  
**FROM Candidate Cand**  
**RIGHT OUTER JOIN Company Comp**  
**ON Cand.CompanyId = Comp.CompanyId**  
  
If we run the above query the output will be as shown in below. If you look at the output, we now got 6 rows. All the rows from the **Candidate**Table that has non null foreign key value plus all the rows from the **Company**Table including the row that is not referenced in the **Candidate** Table.

|  |
| --- |
| http://4.bp.blogspot.com/-pBKO-hJS6Is/TdbK0UbN7NI/AAAAAAAAAFo/brTyad-PLMw/s1600/RightJoin.png  **Right Outer Join Results** |

Instead of using **RIGHT OUTER JOIN**keyword we can just use **RIGHT JOIN**keyword as shown below. **RIGHT OUTER JOIN or RIGHT JOIN** means the same.  
  
**SELECT Cand.CandidateId, Cand.FullName, Cand.CompanyId, Comp.CompanyId, Comp.CompanyName**  
**FROM Candidate Cand**  
**RIGHT JOIN Company Comp**  
**ON Cand.CompanyId = Comp.CompanyId**

[Explain Full Outer Join with an example](http://venkatsqlinterview.blogspot.com/2011/05/full-outer-join-in-sql-server-with.html)

**Inner Join and left join**are the most commonly used joins in real time projects. It is very important that you understand the basics of joins before reading this article. Please read the articles below if you have not done so already.  
**1.** [Basics of Joins in SQL Server](http://venkatsqlinterview.blogspot.com/2011/05/basic-sql-server-interview-questions-on.html)  
**2.** [Inner Join](http://venkatsqlinterview.blogspot.com/2011/05/inner-join-with-example.html)  
**3.** [Left Join](http://venkatsqlinterview.blogspot.com/2011/05/left-join-with-example.html)  
**4.** [Right Join](http://venkatsqlinterview.blogspot.com/2011/05/right-outer-join-with-example.html)  
Now, let us understand **Full Outer join with an example.**  
  
Create 2 tables **Company**and **Candidate**. Use the script below to create these tables and populate them. CompanyId column in Candidate Table is a foreign key referencing CompanyId in Company Table.  
  
**CREATE TABLE Company**  
**(**  
**CompanyId TinyInt Identity Primary Key,**  
**CompanyName Nvarchar(50) NULL**  
**)**  
**GO**  
  
**INSERT Company VALUES('DELL')**  
**INSERT Company VALUES('HP')**  
**INSERT Company VALUES('IBM')**  
**INSERT Company VALUES('Microsoft')**  
**GO**  
  
**CREATE TABLE Candidate**  
**(**  
**CandidateId tinyint identity primary key,**  
**FullName nvarchar(50) NULL,**  
**CompanyId tinyint REFERENCES Company(CompanyId)**  
**)**  
**GO**  
  
**INSERT Candidate VALUES('Ron',1)**  
**INSERT Candidate VALUES('Pete',2)**  
**INSERT Candidate VALUES('Steve',3)**  
**INSERT Candidate VALUES('Steve',NULL)**  
**INSERT Candidate VALUES('Ravi',1)**  
**INSERT Candidate VALUES('Raj',3)**  
**INSERT Candidate VALUES('Kiran',NULL)**  
**GO**

If you want to select all the rows from the **LEFT Table**( In our example **Candidate**Table ) plus all the rows from the **RIGHT table**( In our example **Company**Table ) , then we use **FULL OUTER JOIN**. A query involving a **FULL OUTER JOIN**for the **Candidate and Company** Table is shown below.  
  
**SELECT Cand.CandidateId, Cand.FullName, Cand.CompanyId, Comp.CompanyId, Comp.CompanyName**  
**FROM Candidate Cand**  
**FULL OUTER JOIN Company Comp**  
**ON Cand.CompanyId = Comp.CompanyId**  
  
If we run the above query the output will be as shown in below. If you look at the output, we now got 8 rows. All the rows from the **Candidate** Table and all the rows from the **Company**Table.

|  |
| --- |
| http://4.bp.blogspot.com/-fva2g-6QXEM/TdbQkUDwk7I/AAAAAAAAAFs/agWHEAVk8tU/s1600/FullOuterJoin.png **Full Outer Join Result** |

Instead of using **FULL OUTER JOIN**keyword we can just use **FULL JOIN**keyword as shown below. **FULL OUTER JOIN or FULL JOIN** means the same.  
  
**SELECT Cand.CandidateId, Cand.FullName, Cand.CompanyId, Comp.CompanyId, Comp.CompanyName**  
**FROM Candidate Cand**  
**FULL JOIN Company Comp**  
**ON Cand.CompanyId = Comp.CompanyId**

[Explain Self Join with an example](http://venkatsqlinterview.blogspot.com/2011/05/self-join-with-example.html)

There are 3 different types of joins available in sql server, and they are  
**1.** Cross Join   
**2.** Inner Join or Join   
**3.** Outer Join  
  
**Outer Join**is again divided into 3 types as shown below.  
**1.** Left Outer Join or Left Join   
**2.** Right Outer Join or Right Join   
**3.** Full Outer Join or Full Join   
  
I strongly recomend to learn about the basics and types of joins, before reading this article. Read the articles below, before proceeding with self join.  
**1.** [Basics of Joins](http://venkatsqlinterview.blogspot.com/2011/05/basic-sql-server-interview-questions-on.html)  
**2.** [Inner Join](http://venkatsqlinterview.blogspot.com/2011/05/inner-join-with-example.html)  
**3.** [Left Outer Join](http://venkatsqlinterview.blogspot.com/2011/05/left-join-with-example.html)  
**4.** [Right Outer Join](http://venkatsqlinterview.blogspot.com/2011/05/right-outer-join-with-example.html)  
**5.** [Full Outer Join](http://venkatsqlinterview.blogspot.com/2011/05/full-outer-join-in-sql-server-with.html)

**Self join**is not a different type of join. **Self join**means joining a table with itself. We can have an **inner self join**or **outer self join**. Let us try to understand with an **example**.  
  
To set up the data for the **example**, use the script below to create **Employee Table**and populate it with some sample data. We will be using **Employee Table**to understand **Self Join**.  
  
**CREATE TABLE EMPLOYEE**  
**(**  
**[EMPLOYEEID] INT PRIMARY KEY,**  
**[NAME] NVARCHAR(50),**  
**[MANAGERID] INT**  
**)**  
**GO**  
  
**INSERT INTO EMPLOYEE VALUES(101,'Mary',102)**  
**INSERT INTO EMPLOYEE VALUES(102,'Ravi',NULL)**  
**INSERT INTO EMPLOYEE VALUES(103,'Raj',102)**  
**INSERT INTO EMPLOYEE VALUES(104,'Pete',103)**  
**INSERT INTO EMPLOYEE VALUES(105,'Prasad',103)**  
**INSERT INTO EMPLOYEE VALUES(106,'Ben',103)**  
**GO**  
  
We use **Self Join**, if we have a **table**that references itself. For **example**, In the **Employee Table**below MANAGERID column references EMPLOYEEID column. So the table is said to **referencing itself**. This is the right scenario where we can use **Self Join**. Now I want to write a query that will give me the list of all Employee Names and their respective Manager Names. In order to achieve this I can use Self Join. In the Table below,Raj is the manager for Pete,Prasad and Ben. Ravi is the manager for Raj and Mary. Ravi does not have a manager as he is the president of the Company.

|  |
| --- |
| http://2.bp.blogspot.com/-hE83aq6KeKY/Tdd-iy4fj5I/AAAAAAAAAFw/u9x2hjMciYw/s1600/EmployeeTable.png |

The query below is an example of **Self Join**. Both E1 and E2 refer to the same **Employee**Table. In this query we are joining the **Employee** Table with itself.   
  
**SELECT E1.[NAME],E2.[NAME] AS [MANAGER NAME]**  
**FROM EMPLOYEE E1**  
**INNER JOIN EMPLOYEE E2**  
**ON E2.EMPLOYEEID =E1.MANAGERID**  
  
If we run the above query we only get 5 rows out of the 6 rows as shown below.

|  |
| --- |
| **Inner Self Join** http://4.bp.blogspot.com/-um9BNVkV9Xs/TdeAOvs5wxI/AAAAAAAAAF0/QpJ1MrGiPOY/s1600/InnerSelfJoin.png |

This is because Ravi does not have a Manager. MANAGERID column for Ravi is NULL. If we want to get all the rows then we can use **LEFT OUTER JOIN**as shown below.  
  
**SELECT E1.[NAME],E2.[NAME] AS [MANAGER NAME]**  
**FROM EMPLOYEE E1**  
**LEFT OUTER JOIN EMPLOYEE E2**  
**ON E2.EMPLOYEEID =E1.MANAGERID**  
  
If we execute the above query we get all the rows, including the row that has a null value in the MANAGERID column. The results are shown below. The MANAGERNAME for 2nd record is NULL as Ravi does not have a Manager.

|  |
| --- |
| **Left Outer Self Join** http://4.bp.blogspot.com/--JIy9u-2qDk/TdeCX2vxuPI/AAAAAAAAAF4/I10IaVn8KsY/s1600/LeftOuterSelfJoin.png |

Let us now slightly modify the above query using **COALESCE**as shown below. Read [COALESCE function in SQL Server](http://venkatsqlinterview.blogspot.com/2011/05/what-is-use-of-coalesce-in-sql-server.html) to understand **COALESCE**in a greater detail.  
  
**SELECT E1.[NAME],COALESCE(E2.[NAME],'No Manager') AS [MANAGER NAME]**  
**FROM EMPLOYEE E1**  
**LEFT JOIN EMPLOYEE E2**  
**ON E2.EMPLOYEEID =E1.MANAGERID**  
  
If we execute the above query the output will be as shown in the image below. This is how **COALESCE** can be used.

|  |
| --- |
| **Left Outer Self Join with COALESCE** http://4.bp.blogspot.com/-8SrwcgzOQZE/TdeES3ohXgI/AAAAAAAAAF8/5TTCKtTiNcU/s1600/SlefJoinCoalesce.png |

[What is the difference between Index Scan and Index Seek](http://venkatsqlinterview.blogspot.com/2011/05/difference-between-index-scan-and-index.html)

**Index Scan:**  
**Index Scan**scans each and every record in the index. **Table Scan**is where the table is processed row by row from beginning to end. If the index is a clustered index then an **index scan** is really a **table scan**. Since a scan touches every row in the table whether or not it qualifies, the cost is proportional to the total number of rows in the table. **Hence, a scan is an efficient strategy only if the table is small.**   
  
**Index Seek:**  
Since a seek only touches rows that qualify and pages that contain these qualifying rows, the cost is proportional to the number of qualifying rows and pages rather than to the total number of rows in the table.   
  
**Example**  
I have an **employee**table as shown in the diagram below. EmployeeId is the primary key. We have a clustered index on the employeeId column.

|  |
| --- |
| http://3.bp.blogspot.com/-Zy43d-GW4W4/TdeSgwliL4I/AAAAAAAAAGA/jV85y8QIJYI/s1600/IndexScan.png |

**Query 1** : **Select \* from Employee where FirstName='Ben'**  
**Query 2** : **Select \* from Employee where EmployeeId=2**  
  
Query 1 will do an **Index scan**(Table Scan) to retrieve the record as there is no Index on the FirstName column.   
Query 2 will do an **Index seek**to retrieve the record as there is an Index on the EmployeeId column.

[What are the advantages of using stored procedures?](http://venkatsqlinterview.blogspot.com/2011/07/advantages-of-stored-procedures.html)

This is a very common sql server interview question. There are several advantages of using stored procedures over adhoc queries, as listed below.  
  
1. **Better Performance**: Stored Procedures are compiled and their execution plan is cached and used again, when the same SP is executed again. Although adhoc queries also create and reuse plan, the plan is reused only when the query is textual match and the datatypes are matching with the previous call. Any change in the datatype or you have an extra space in the query then, a new plan is created.  
  
2. **Better Security**: Applications making use of dynamically built adhoc sql queries are highly susceptible to sql injection attacks, whereas Stored Procedures can avoid SQL injection attacks completely.  
  
3. **Reduced Network Traffic**: Stored procedures can reduce network traffic to a very great extent when compared with adhoc sql queries. With stored procedures, you only need to send the name of the procedure between client and server. Imagine the amount of network bandwidth that can be saved especially if the stored procedure contains 1000 to 2000 lines of SQL.  
  
4. **Better Maintenance and Reusability**: Stored procedures can be used anywhere in the application. It is easier to maintain a stored procedure that is used on several pages as the modifications just need to be changed at one place where the stored procedure is defined. On the other hand, maintaining an adhoc sql query that's used on several pages is tedious and error prone, as we have to make modifications on each and every page.

[What are the different ways to replace NULL values in SQL Server?](http://venkatsqlinterview.blogspot.co.uk/2012/03/what-are-different-ways-to-replace-null.html)

This interview question is not that common. My friend faced this interview question, when he attended an interview in London. My friend said we can use **COALESCE()** in SQL Server. Then the interviewer said, that's very good answer, but do you know of any other way?

Apart from using **COALESCE()**, there are 2 other ways to replace NULL values in SQL Server. Let's understand this with an example.  
  
  
I have a Table tblEmployee, as shown in the diagram below. Some of the Employees does not have gender. All those employees who does not have Gender, must have a replacement value of **'No Gender'** in your query result. Let's explore all the 3 possible options we have.

|  |
| --- |
| http://2.bp.blogspot.com/-ozwb8tX6SMY/T3dyEe9TUZI/AAAAAAAAAKM/MGbDiR49D48/s1600/ReplaceNulls.png |

**Option 1** : Replace NULL values in SQL Server using **ISNULL()** function.  
  
Select Name, ISNULL(Gender,'No Gender') as Gender  
From tblEmployee  
  
  
  
**Option 2** : Replace NULL values in SQL Server using **CASE**.  
  
Select Name, Case  When Gender IS NULL Then 'No Gender' Else Gender End as Gender  
From tblEmployee  
  
  
  
**Option 3** : Replace NULL values in SQL Server using **COALESCE()** function.  
  
Select Name, Coalesce(Gender, 'No Gender') as Gender  
From tblEmployee

[SQL Server interview questions on string manipulation functions](http://venkatsqlinterview.blogspot.co.uk/2012/04/sql-server-interview-questions-on.html)

The following 2 SQL Server Interview questions were asked when I attended an interview for SQL Server Developer role.  
  
**Can you list a few useful string manipulation functions in SQL Server?**  
LEN(), SUBSTRING(), CHARINDEX(), LEFT(), RIGHT() etc.

**Then he asked me, Can you give me one example of where you have used these functions in your experience?**  
The following is one simple real time example, where we can use LEN(), CHARINDEX() and SUBSTRING() functions. Let us assume we have table as shown below.

|  |
| --- |
| http://3.bp.blogspot.com/-0jh7vLiE134/T38tJBXZQZI/AAAAAAAAAKY/IOP_rq26kW8/s1600/String+Manipulation.png |

I want you to write a query to find out total number of emails, by domain. The result of the query should be as shown below.

|  |
| --- |
| http://3.bp.blogspot.com/-A8B18ogbvno/T38tyZ6P_aI/AAAAAAAAAKg/bcApP3RbyVw/s1600/String+Manipulation+Functions.png |

We can use LEN(), CHARINDEX() and SUBSTRING() functions to produce the desired results. Please refer to the query below.

|  |
| --- |
| Select SUBSTRING(Email,CHARINDEX('@',Email)+1,(LEN(Email) - CHARINDEX('@',Email))) as EmailDomain, Count(\*) as Total From TableName  Group By SUBSTRING(Email,CHARINDEX('@',Email)+1,(LEN(Email) - CHARINDEX('@',Email))) Order by Count(\*) Desc |

[Write a Stored Procedure that takes column name as a parameter and returns the result sorted by the column that is passed](http://venkatsqlinterview.blogspot.co.uk/2012/04/write-stored-procedure-that-takes.html)

Let's understand the requirement better with an example. I have an Employee table as shown below.

|  |
| --- |
| http://1.bp.blogspot.com/-vdg0xk8pUGI/T4Ry84i52WI/AAAAAAAAAKo/NdFN1aLleXc/s1600/Employee+Table.png |

I want a stored procedure that returns employee data sorted by a column that the user is going to pass into the stored procedure as a parameter. There are 2 ways of doing this.  
  
**Option 1: Use Case Statement as shown below:**  
Create Proc **spGetEmployeesSorted**  
@SortCoumn nvarchar(10)  
as  
Begin  
  
Select [Id],[Name],[Gender],[Salary],[City]   
From   [Employee]  
Order by Case When @SortCoumn = 'Id' Then Id End,  
                Case When @SortCoumn = 'Name' Then Name End,  
                Case When @SortCoumn = 'Gender' Then Gender End,  
                Case When @SortCoumn = 'Salary' Then Salary End,  
                Case When @SortCoumn = 'City' Then City End  
  
End

**Option 2: Use Dynamic SQL as shown below:**  
Create Proc **spGetEmployeesSortedUsingDynamicSQL**  
@SortCoumn nvarchar(10)  
as  
Begin  
  
Declare @DynamicQuery nvarchar(100)  
Set @DynamicQuery = 'select [Id],[Name],[Gender],[Salary],[City] from [Employee] order by ' + @SortCoumn  
Execute(@DynamicQuery)  
  
  
End

[What is deferred name resolution in SQL Server?](http://venkatsqlinterview.blogspot.com/2012/05/what-is-deferred-name-resolution-in-sql.html)

Let me explain deferred name resolution with an example. Consider the stored procedure shown below.  
  
  
Create procedure spGetCustomers  
as  
Begin  
 Select \* from Customers1  
End

**Customers1**table does not exist. When you execute the above SQL code, the stored procedure **spGetCustomers**will be successfully created without errors. But when you try to **call or execute** the stored procedure using **Execute spGetCustomers**, you will get a run time error stating Invalid object name 'Customers1'.  
  
  
So, at the time of creating stored procedures, only the syntax of the sql code is checked. The objects used in the stored procedure are not checked for their existence. Only when we try to run the procedure, the existence of the objects is checked. So, the process of postponing, the checking of physical existence of the objects until runtime, is called as deffered name resolution in SQL server.  
  
  
**Functions** in sql server does not support **deferred name resolution**. If you try to create an inline table valued function as shown below, we get an error stating Invalid object name 'Customers1' at the time of creation of the function itself.  
  
  
Create function fnGetCustomers()  
returns table  
as  
return Select \* from Customers1  
  
  
So, this proves that, stored procedures support deferred name resolution, where as functions does not. Infact, this is one of the major difference between functions and stored procedures in sql server. For all the [**differences between stored procedures and functions, please read this artilce.**](http://venkatsqlinterview.blogspot.com/2011/05/what-is-difference-between-user-defined.html)