**1) ACID Properties in SQL Server**

A - Atomicity

C - Consistency

I - Isolation

D - Durability

**Atomicity**

All the operations in the transaction must be complete, successfully and be committed, if any one of the operation fails then all the transactions must be rolled back in their previous state.

**Consistency**

The transaction must be consistent in a state, this means that the transaction must correctly change the state of the system for a particular operation.

**Isolation**

It means that one operation within the transaction cannot see the result of another operation within the transaction.

**Durability**

It means that anything committed to the managed resources must survive or failure and it cannot be done the damages to the resources in case of operation fails.

**2) Normalization**

[**https://www.morpheusdata.com/blog/2015-02-17-pros-cons-db-normalization**](https://www.morpheusdata.com/blog/2015-02-17-pros-cons-db-normalization)

To normalize or not to normalize? Find out when normalization of a database is helpful and when it is not.

TL;DR: When using a relational database, normalization can help keep the data free of errors and can also help ensure that the size of the database doesn't grow large with duplicated data. At the same time, some types of operations can be slower in a normalized environment. So, when should you normalize and when is it better to proceed without normalization?

**What is Normalization?**

*Database normalization is the process of organizing data within a database in the most efficient manner possible*. For example, you likely do not want a username stored in several different tables within your database when you could store it in a single location and point to that user via an ID instead.

By keeping the unchanging user ID in the various tables that need the user, you can always point it back to the appropriate table to get the current username, which is stored in only a single location. Any updates to the username occur only in that place, making the data more reliable.

**What Is Good about Database Normalization?**

A normalized database is advantageous when operations will be write-intensive or when [ACID compliance](http://en.wikipedia.org/wiki/ACID) is required. Some advantages include:

1. Updates run quickly due to no data being duplicated in multiple locations.
2. Inserts run quickly since there is only a single insertion point for a piece of data and no duplication is required.
3. Tables are typically smaller that the tables found in non-normalized databases. This usually allows the tables to fit into the buffer, thus offering faster performance.
4. Data integrity and consistency is an absolute must if the database must be ACID compliant. A normalized database helps immensely with such an undertaking.

**What Are the Drawbacks of Database Normalization?**

A normalized database is not as advantageous under conditions where an application is read-intensive. Here are some of the disadvantages of normalization:

1. Since data is not duplicated, table joins are required. This makes queries more complicated, and thus read times are slower.
2. Since joins are required, indexing does not work as efficiently. Again, this makes read times slower because the joins don't typically work well with indexing.

**What if the Application is Read-Intensive and Write-Intensive?**

In some cases, it isn't as clear that one strategy should be used over the other. Obviously, some applications really need both normalized and non-normalized data to work as efficiently as possible.

In such cases, companies will often use more than one database: a relational data such as MySQL for ACID compliant and write-intensive operations and a NoSQL database such as MongoDB for read-intensive operations on data where duplication is not as big of an issue.

**<http://agiledata.org/essays/dataNormalization.html>**

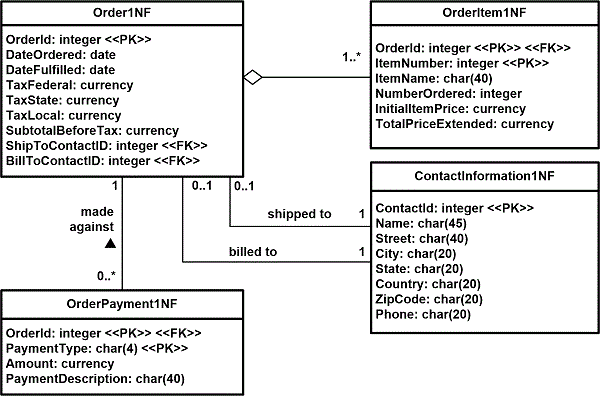
**Table 1. Data Normalization Rules.**

|  |  |
| --- | --- |
| **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). |

**First Normal Form (1NF) (Single Valued Dependency):** Let’s consider an example. **An entity type is in first normal form (1NF) when it contains no repeating groups of data.** For example, in Figure 1 you see that there are several repeating attributes in the data Order0NF table – the ordered item information repeats nine times and the contact information is repeated twice, once for shipping information and once for billing information. Although this initial version of orders could work, what happens when an order has more than nine order items? Do you create additional order records for them? What about the vast majority of orders that only have one or two items? Do we really want to waste all that storage space in the database for the empty fields? Likely not. Furthermore, do you want to write the code required to process the nine copies of item information, even if it is only to marshal it back and forth between the appropriate number of objects. Once again, likely not.

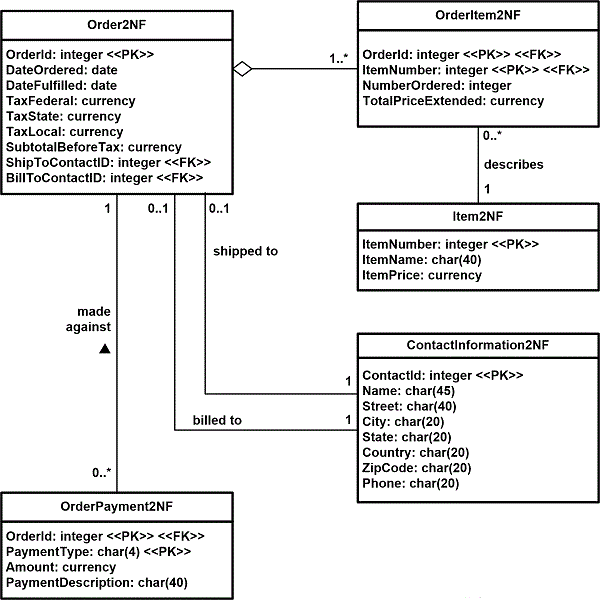


Above figure presents a reworked data schema where the order schema is put in first normal form. The introduction of the OrderItem1NF table enables us to have as many, or as few, order items associated with an order, increasing the flexibility of our schema while reducing storage requirements for small orders (the majority of our business). The ContactInformation1NF table offers a similar benefit, when an order is shipped and billed to the same person (once again the majority of cases) we could use the same contact information record in the database to reduce data redundancy. OrderPayment1NF was introduced to enable customers to make several payments against an order – Order0NF could accept up to two payments, the type being something like “MC" and the description “MasterCard Payment", although with the new approach far more than two payments could be supported, potentially one per payment type. Multiple payments are accepted only when the total of an order is large enough that a customer must pay via more than one approach, perhaps paying some by check and some by credit card.

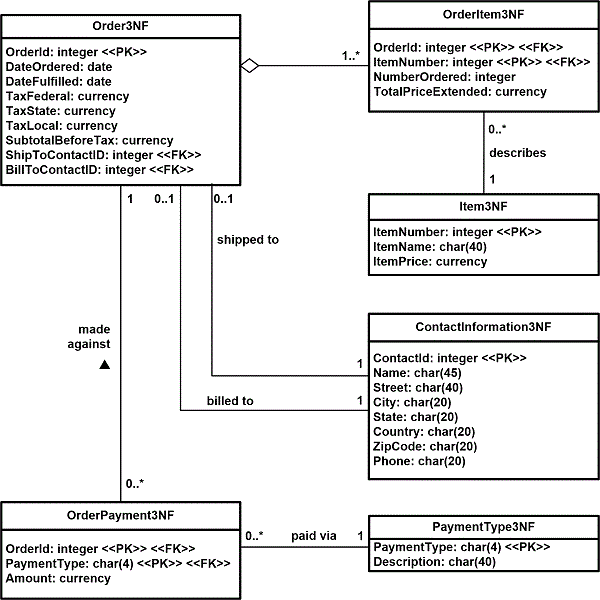


An important thing to notice is the application of primary and foreign keys in the new solution. Order1NF has kept OrderID, the original key of Order0NF, as its primary key. To maintain the relationship back to Order1NF, the OrderItem1NF table includes the OrderID column within its schema, which is why it has the stereotype of FK. When a new table is introduced into a schema, in this case OrderItem1NF, as the result of first normalization efforts it is common to use the primary key of the original table (Order0NF) as part of the primary key of the new table. Because OrderID is not unique for order items, you can have several order items on an order, the column ItemNumber (which is unique to a type of item) was used to form a composite primary key for the OrderItem1NF table. A different approach to keys was taken with the ContactInformation1NF table. The column ContactID, a surrogate key that has no business meaning, was made the primary key.

**Second Normal Form (2NF) (Partial Dependency):** Although the solution presented in Figure 2 is improved over that of Figure 1, it can be normalized further. Figure 3 presents the data schema of Figure 2 in second normal form (2NF). **An entity type is in second normal form (2NF) when it is in 1NF and when every non-key attribute (any attribute that is not part of the primary key), is fully dependent on the primary key.** This was definitely not the case with the OrderItem1NF table, therefore we need to introduce the new table Item2NF. The problem with OrderItem1NF is that item information, such as the name and price of an item, do not depend upon an order for that item. For example, if Hal Jordan orders three widgets and Oliver Queen orders five widgets, the facts that the item is called a “widget" and that the unit price is $19.95 is constant. This information depends on the concept of an item, not the concept of an order for an item, and therefore should not be stored in the order items table – therefore the Item2NF table was introduced. OrderItem2NF retained the TotalPriceExtended column, a calculated value that is the number of items ordered multiplied by the price of the item. The value of the SubtotalBeforeTax column within the Order2NF table is the total of the values of the total price extended for each of its order items.



**Third Normal Form (3NF) (Transitive Dependency): An entity type is in third normal form (3NF) when it is in 2NF and when all of its attributes are directly dependent on the primary key**. A better way to word this rule might be that the attributes of an entity type must depend on all portions of the primary key. In this case there is a problem with the OrderPayment2NF table, the payment type description (such as “Mastercard" or “Check") depends only on the payment type, not on the combination of the order id and the payment type. To resolve this problem the PaymentType3NF table was introduced in Figure 4, containing a description of the payment type as well as a unique identifier for each payment type.



**3) Denormalization: A denormalized data model is not the same as a data model that has not been normalized, and denormalization should only take place after a satisfactory level of normalization has taken place** and that any required constraints and/or rules have been created to deal with the inherent anomalies in the design.

Denormalization is a strategy used on a previously-normalized database to increase performance. In computing, denormalization is the process of trying to improve the read performance of a database, at the expense of losing some write performance, by adding redundant copies of data or by grouping data. It is often motivated by performance or scalability in relational database software needing to carry out very large numbers of read operations. Denormalization should not be confused with Unnormalized form. Databases/tables must first be normalized to efficiently denormalize them.

**4) What is the difference between Clustered and Non-Clustered Indexes in SQL Server**

Indexes are used to speed-up query process in SQL Server, resulting in high performance. They are similar to textbook indexes. In textbooks, if you need to go to a particular chapter, you go to the index, find the page number of the chapter and go directly to that page. Without indexes, the process of finding your desired chapter would have been very slow.

The same applies to indexes in databases. Without indexes, a DBMS has to go through all the records in the table in order to retrieve the desired results. This process is called table-scanning and is extremely slow. On the other hand, if you create indexes, the database goes to that index first and then retrieves the corresponding table records directly.

**There are two types of Indexes in SQL Server:**

1. Clustered Index
2. Non-Clustered Index

**Clustered Index**

**A clustered index defines the order in which data is physically stored in a table**. Table data can be sorted in only way, therefore, **there can be only one clustered index per table**. In SQL Server, the **primary key constraint automatically creates a clustered index on that particular column**.

Let’s take a look. First, create a “student” table inside “schooldb” by executing the following script:

CREATE TABLE student

(

id INT PRIMARY KEY,

name VARCHAR(50) NOT NULL,

gender VARCHAR(50) NOT NULL,

DOB datetime NOT NULL,

total\_score INT NOT NULL,

city VARCHAR(50) NOT NULL

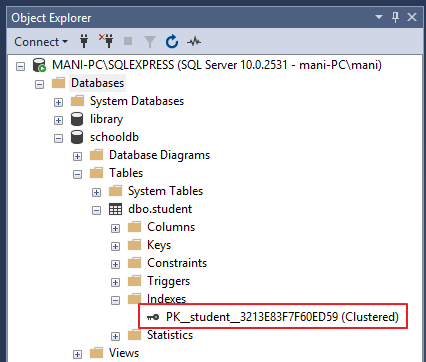
)

Notice here in the “student” table we have set primary key constraint on the “id” column. This automatically creates a clustered index on the “id” column. To see all the indexes on a particular table execute “sp\_helpindex” stored procedure. This stored procedure accepts the name of the table as a parameter and retrieves all the indexes of the table. The following query retrieves the indexes created on student table.

EXECUTE sp\_helpindex student

The above query will return this result:

|  |  |  |
| --- | --- | --- |
| index\_name | index\_description | index\_keys |
| PK\_\_student\_\_3213E83F7F60ED59 | clustered, unique, primary key located on PRIMARY | id |

In the output you can see the only one index. This is the index that was automatically created because of the primary key constraint on the “id” column.

This clustered index stores the record in the student table in the ascending order of the “id”. Therefore, if the inserted record has the id of 5, the record will be inserted in the 5th row of the table instead of the first row. Similarly, if the fourth record has an id of 3, it will be inserted in the third row instead of the fourth row. This is because the **clustered index has to maintain the physical order of the stored records according to the indexed column** i.e. id. To see this ordering in action, execute the following script:

INSERT INTO student VALUES

(6, 'Kate', 'Female', '03-JAN-1985', 500, 'Liverpool'),

(2, 'Jon', 'Male', '02-FEB-1974', 545, 'Manchester'),

(9, 'Wise', 'Male', '11-NOV-1987', 499, 'Manchester'),

(3, 'Sara', 'Female', '07-MAR-1988', 600, 'Leeds'),

(1, 'Jolly', 'Female', '12-JUN-1989', 500, 'London'),

(4, 'Laura', 'Female', '22-DEC-1981', 400, 'Liverpool'),

(7, 'Joseph', 'Male', '09-APR-1982', 643, 'London'),

(5, 'Alan', 'Male', '29-JUL-1993', 500, 'London'),

(8, 'Mice', 'Male', '16-AUG-1974', 543, 'Liverpool'),

(10, 'Elis', 'Female', '28-OCT-1990', 400, 'Leeds');

The above script inserts ten records in the student table. Notice here the records are inserted in random order of the values in the “id” column. But because of the default clustered index on the id column, the records are physically stored in the ascending order of the values in the “id” column. Execute the following SELECT statement to retrieve the records from the student table.

SELECT \* FROM student

The records will be retrieved in the following order:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| id | name | gender | DOB | total\_score | city |
| 1 | Jolly | Female | 1989-06-12 00:00:00.000 | 500 | London |
| 2 | Jon | Male | 1974-02-02 00:00:00.000 | 545 | Manchester |
| 3 | Sara | Female | 1988-03-07 00:00:00.000 | 600 | Leeds |
| 4 | Laura | Female | 1981-12-22 00:00:00.000 | 400 | Liverpool |
| 5 | Alan | Male | 1993-07-29 00:00:00.000 | 500 | London |
| 6 | Kate | Female | 1985-01-03 00:00:00.000 | 500 | Liverpool |
| 7 | Joseph | Male | 1982-04-09 00:00:00.000 | 643 | London |
| 8 | Mice | Male | 1974-08-16 00:00:00.000 | 543 | Liverpool |
| 9 | Wise | Male | 1987-11-11 00:00:00.000 | 499 | Manchester |
| 10 | Elis | Female | 1990-10-28 00:00:00.000 | 400 | Leeds |

**Creating Custom Clustered Index:** You can create your own custom index as well the default clustered index. **To create a new clustered index on a table you first have to delete the previous index.**

Now, to create a new clustered Index, execute the following script:

**CREATE CLUSTERED INDEX IX\_tblStudent\_Gender\_Score**

**ON student ( gender ASC, total\_score DESC )**

The process of creating clustered index is similar to a normal index with one exception. With clustered index, you have to use the keyword “CLUSTERED” before “INDEX”.

The above script creates a clustered index named “IX\_tblStudent\_Gender\_Score” on the student table. This index is created on the “gender” and “total\_score” columns. **An index that is created on more than one column is called “composite index”.**

The above index first sorts all the records in the ascending order of the gender. If gender is same for two or more records, the records are sorted in the descending order of the values in their “total\_score” column. You can create a clustered index on a single column as well. Now if you select all the records from the student table, they will be retrieved in the following order:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| id | name | gender | DOB | total\_score | city |
| 3 | Sara | Female | 1988-03-07 00:00:00.000 | 600 | Leeds |
| 1 | Jolly | Female | 1989-06-12 00:00:00.000 | 500 | London |
| 6 | Kate | Female | 1985-01-03 00:00:00.000 | 500 | Liverpool |
| 4 | Laura | Female | 1981-12-22 00:00:00.000 | 400 | Liverpool |
| 10 | Elis | Female | 1990-10-28 00:00:00.000 | 400 | Leeds |
| 7 | Joseph | Male | 1982-04-09 00:00:00.000 | 643 | London |
| 2 | Jon | Male | 1974-02-02 00:00:00.000 | 545 | Manchester |
| 8 | Mice | Male | 1974-08-16 00:00:00.000 | 543 | Liverpool |
| 5 | Alan | Male | 1993-07-29 00:00:00.000 | 500 | London |
| 9 | Wise | Male | 1987-11-11 00:00:00.000 | 499 | Manchester |

**Non-Clustered Indexes:** A non-clustered index doesn’t sort the physical data inside the table. In fact, **a non-clustered index is stored at one place and table data is stored in another place**. This is similar to a textbook where the book content is located in one place and the index is located in another. This allows for more than one non-clustered index per table.

It is important to mention here that inside the table the data will be sorted by a clustered index. However, inside the non-clustered index data is stored in the specified order. The index contains column values on which the index is created and the address of the record that the column value belongs to.

When a query is issued against a column on which the index is created, the database will first go to the index and look for the address of the corresponding row in the table. It will then go to that row address and fetch other column values. It is due to this additional step that **non-clustered indexes are slower than clustered indexes.**

Creating a Non-Clustered Index

The syntax for creating a non-clustered index is similar to that of clustered index. However, in case of non-clustered index keyword “NONCLUSTERED” is used instead of “CLUSTERED”. Take a look at the following script.

**CREATE NONCLUSTERED INDEX IX\_tblStudent\_Name**

**ON student(name ASC)**

The above script creates a non-clustered index on the “name” column of the student table. The index sorts by name in ascending order. As we said earlier, the table data and index will be stored in different places. The table records will be sorted by a clustered index if there is one. The index will be sorted according to its definition and will be stored separately from the table.

**Student Table Data:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| id | name | gender | DOB | total\_score | City |
| 1 | Jolly | Female | 1989-06-12 00:00:00.000 | 500 | London |
| 2 | Jon | Male | 1974-02-02 00:00:00.000 | 545 | Manchester |
| 3 | Sara | Female | 1988-03-07 00:00:00.000 | 600 | Leeds |
| 4 | Laura | Female | 1981-12-22 00:00:00.000 | 400 | Liverpool |
| 5 | Alan | Male | 1993-07-29 00:00:00.000 | 500 | London |
| 6 | Kate | Female | 1985-01-03 00:00:00.000 | 500 | Liverpool |
| 7 | Joseph | Male | 1982-04-09 00:00:00.000 | 643 | London |
| 8 | Mice | Male | 1974-08-16 00:00:00.000 | 543 | Liverpool |
| 9 | Wise | Male | 1987-11-11 00:00:00.000 | 499 | Manchester |
| 10 | Elis | Female | 1990-10-28 00:00:00.000 | 400 | Leeds |

**IX\_tblStudent\_Name Index Data**

**name Row Address**

Alan Row Address

Elis Row Address

Jolly Row Address

Jon Row Address

Joseph Row Address

Kate Row Address

Laura Row Address

Mice Row Address

Sara Row Address

Wise Row Address

Notice, here in the index every row has a column that stores the address of the row to which the name belongs. So if a query is issued to retrieve the gender and DOB of the student named “Jon”, the database will first search the name “Jon” inside the index. It will then read the row address of “Jon” and will go directly to that row in the “student” table to fetch gender and DOB of Jon.

**Conclusion**

From the discussion we find following differences between clustered and non-clustered indexes.

1. There can be only one clustered index per table. However, you can create multiple non-clustered indexes on a single table.
2. Clustered indexes only sort tables. Therefore, they do not consume extra storage. Non-clustered indexes are stored in a separate place from the actual table claiming more storage space.
3. Clustered indexes are faster than non-clustered indexes since they don’t involve any extra lookup step.

**When to Use Clustered or Non-Clustered Indexes**

Now that you know the differences between a clustered and a non-clustered index, let’s see the different scenarios for using each of them.

**1. Number of Indexes**

This is pretty obvious. If you need to create multiple indexes on your database, go for non-clustered index since there can be only one clustered index.

**2. SELECT Operations**

**If you want to select only the index value that is used to create and index, non-clustered indexes are faster**. For example, if you have created an index on the “name” column and you want to select only the name, non-clustered indexes will quickly return the name.

However, if you want to select other column values such as age, gender using the name index, the SELECT operation will be slower since first the name will be searched from the index and then the reference to the actual table record will be used to search the age and gender.

On the other hand, with clustered indexes since all the records are already sorted, the SELECT operation is faster if the data is being selected from columns other than the column with clustered index.

**3. INSERT/UPDATE Operations**

The INSERT and UPDATE operations are faster with non-clustered indexes since the actual records are not required to be sorted when an INSERT or UPDATE operation is performed. Rather only the non-clustered index needs updating.

**4. Disk Space**

Since, non-clustered indexes are stored at a separate location than the original table, non-clustered indexes consume additional disk space. If disk space is a problem, use a clustered index.

**5) Difference between UNION and UNION ALL**

UNION removes duplicate records (where all columns in the results are the same), UNION ALL does not.

There is a performance hit when using UNION instead of UNION ALL, since the database server must do additional work to remove the duplicate rows, but usually you do not want the duplicates (especially when developing reports).

**UNION Example:**

SELECT 'foo' AS bar UNION SELECT 'foo' AS bar

Result:

+-----+

| **bar** |

+-----+

| foo |

+-----+

1 row in set (0.00 sec)

**UNION ALL example:**

SELECT 'foo' AS bar UNION ALL SELECT 'foo' AS bar

**Result:**

+-----+

| **bar** |

+-----+

| foo |

| foo |

+-----+

2 rows in set (0.00 sec)

**6) What is Cross Join in SQL?**

1. The SQL CROSS JOIN produces a result set which is the number of rows in the first table multiplied by the number of rows in the second table if no WHERE clause is used along with CROSS JOIN. This kind of result is called as Cartesian Product.
2. *If WHERE clause is used with CROSS JOIN, it functions like an INNER JOIN.*
3. An alternative way of achieving the same result is to use column names separated by commas after SELECT and mentioning the table names involved, after a FROM clause.

**7) Natural Join (Not supported in SQL Server 2012 and above)**

A NATURAL JOIN is a JOIN operation that creates an implicit join clause for you based on the **common columns in the two tables being joined**. Common columns are **columns that have the same name, types, and lengths in both tables**. A NATURAL JOIN can be an INNER join, a LEFT OUTER join, or a RIGHT OUTER join.

One significant difference between INNER JOIN and NATURAL JOIN is the number of columns returned.

Consider:

**TableA**  **TableB**

Column1 Column2 Column1 Column3

1 2 1 3

The INNER JOIN of TableA and TableB on Column1 will return

a.column1 a.column2 b.column1 b.column3

1 2 1 3

SELECT \* FROM TableA INNER JOIN TableB USING (Column1)

SELECT \* FROM TableA INNER JOIN TableB ON TableA.Column1 = TableB.Column1

The NATURAL JOIN of TableA and TableB on Column1 will return:

column1 column2 column3

1 2 3

SELECT \* FROM TableA NATURAL JOIN TableB

The repeated column is avoided.

**8) Join and Inner Join both are same**

Both these joins will give me the same results:

SELECT C.Name FROM Customers C **JOIN** Orders O ON O.CustomerId = C.CustomerId

SELECT C.Name FROM Customers C **INNER** JOIN Orders O ON O.CustomerId = C.CustomerId

**Some Examples:**

**select \* from Customers**

**CustomerId Name**

1 Shree

2 Kalpana

3 Basavaraj

**select \* from Orders**

**OrderId CustomerId**

100 1

200 4

300 3

**9) Full Join**

The FULL OUTER JOIN/FULL JOIN keyword returns all records when there is a match in left (table1) or right (table2) table records.

select \* from Customers **Full JOIN** Orders on Customers.customerid=Orders.customerid

**CustomerId Name OrderId CustomerId**

1 Shree 100 1

2 Kalpana NULL NULL

3 Basavaraj 300 3

NULL NULL 200 4

select \* from Customers **Left JOIN** Orders on Customers.customerid=Orders.customerid

**CustomerId Name OrderId CustomerId**

1 Shree 100 1

2 Kalpana NULL NULL

3 Basavaraj 300 3

select \* from Customers **Right JOIN** Orders on Customers.customerid=Orders.customerid

**CustomerId Name OrderId CustomerId**

1 Shree 100 1

NULL NULL 200 4

3 Basavaraj 300 3

select \* from Customers **Cross JOIN** Orders

**CustomerId Name OrderId CustomerId**

1 Shree 100 1

2 Kalpana 100 1

3 Basavaraj 100 1

1 Shree 200 4

2 Kalpana 200 4

3 Basavaraj 200 4

1 Shree 300 3

2 Kalpana 300 3

3 Basavaraj 300 3

select \* from Customers **Cross JOIN** Orders **where** Customers.customerid=Orders.customerid

***Note :*** *(Works as Inner Join when use where clause in Cross Join)*

**CustomerId Name OrderId CustomerId**

1 Shree 100 1

3 Basavaraj 300 3

select \* from Customers, Orders **(Works as Cross Join)**

**CustomerId Name OrderId CustomerId**

1 Shree 100 1

2 Kalpana 100 1

3 Basavaraj 100 1

1 Shree 200 4

2 Kalpana 200 4

3 Basavaraj 200 4

1 Shree 300 3

2 Kalpana 300 3

3 Basavaraj 300 3

select \* from Customers,Orders where Customers.customerid=Orders.customerid

***Note :*** *(Works as Inner Join)*

**CustomerId Name OrderId CustomerId**

1 Shree 100 1

3 Basavaraj 300 3

**Self Join Example: Employee Manager Query**

create table #temp

(

empid int,

mgrid int,

name varchar(50)

)

insert into #temp

values (1,3,'Rahul'), (2,4,'Ajay'), (3,2,'Pankaj'), (4,1,'Sunil')

select a.Empid, a.name as EmpName, b.empid as MgrId, b.name as MgrName

from #temp a left join #temp b

on a.mgrid = b.empid

**Empid EmpName MgrId MgrName**

1 Rahul 3 Pankaj

2 Ajay 4 Sunil

3 Pankaj 2 Ajay

4 Sunil 1 Rahul

**Query to insert the record from #temp to #temp1 which are not in #temp1**

select top 2 \* into #temp1 from #temp

select \* from #temp

**empid mgrid name**

1 3 Rahul

2 4 Ajay

3 2 Pankaj

4 1 Sunil

**select \* from #temp1**

empid mgrid name

1 3 Rahul

2 4 Ajay

insert into #temp1 select \* from (select \* from #temp where empid not in(select empid from #temp1)) tt

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

1. The function must return a value but in Stored Procedure it is optional. Even a procedure can return zero or n values.
2. Functions can have only input parameters for it whereas Procedures can have input or output parameters.
3. Functions can be called from Procedure whereas Procedures cannot be called from a Function.
4. The procedure allows SELECT as well as DML(INSERT/UPDATE/DELETE) statement in it whereas Function allows only SELECT statement in it.
5. Procedures cannot be utilized in a SELECT statement whereas Function can be embedded in a SELECT statement.
6. Stored Procedures cannot be used in the SQL statements anywhere in the WHERE/HAVING/SELECT section whereas Function can be.
7. Functions that return tables can be treated as another rowset. This can be used in JOINs with other tables.
8. Inline Function can be though of as views that take parameters and can be used in JOINs and other Rowset operations.
9. An exception can be handled by try-catch block in a Procedure whereas try-catch block cannot be used in a Function.
10. We can use Transactions in Procedure whereas we can't use Transactions in Function.

**The uses of views In SQL**

Views are used to implement the security mechanism in SQL Server. Views are generally used to restrict the user from viewing certain columns and rows. Views display only the data specified in the query, so it shows only the data that is returned by the query defined during the creation of the view. The rest of the data is totally abstract from the end user.

**To refresh a view**

Exec sp\_refreshview Employee\_View1

**SchemaBinding a VIEW**

Such a way if we change the data type of any column in a table then we should refresh the view. **If we want to prevent any type of change in a base table** then we can use the concept of SCHEMABINDING. It will lock the tables being referred to by the view and restrict all kinds of changes that may change the table schema (no Alter command).

We can't specify "Select \* from tablename" with the query. We need to specify all the column names for reference.

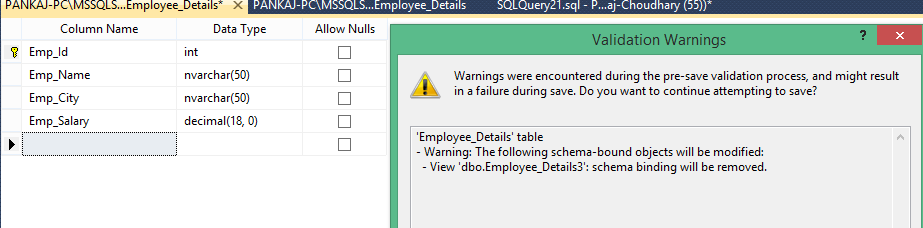
**Create View Employee\_Details3**

**with SCHEMABINDING**

**as**

**select Emp\_Id,Emp\_Name,Emp\_Salary,Emp\_City from DBO.Employee\_Details**

In the preceding example, we create a view using Schemabinding. Now we try to change the datatype of Emp\_Salary from int to Decimal in the Base Table.



**Encrypt a view in SQL Server**

The “WITH ENCRYPTION” option can encrypt any views. That means it will not be visible via SP\_HELPTEXT. **This option encrypts the definition of the view.** **Users will not be able to see the definition of the view after it is created. This is the main advantage of the view where we can make it secure.**

**Create View Employee\_Details4**

**with Encryption**

**as**

**select Emp\_Id,Emp\_Name,Emp\_Salary,Emp\_City from DBO.Employee\_Details**

**Why we should not use the cursor**

Because cursors take up memory and create locks.

When you open a cursor, you are basically loading those rows into memory and locking them, creating potential blocks. Then, as you loop through the cursor, you are making changes to other tables and still keeping all of the memory and locks of the cursor open.

All of which has the potential to cause performance issues for other users.

**10) Can records be deleted from a View in SQL Server?**

It depends.

There are two types of Views in SQL Server. One is a “simple” view that contains data from one table only, and the other is a “complex” view that contains data from multiple tables.

A delete operation can be performed on records in a simple view, but not in a complex view.

**11) Difference between Primary key and Unique key.**

**Primary Key**

1. Enforces uniqueness of the column in a table

2. Default clustered index

3. Does not allow nulls

**Unique Key**

1. Enforces the uniqueness of the column in a table.

2. Default non-clustered index.

3. Allows one null value

**12) Define the following keys Candidate key, Alternate key, Composite key.**

1 .Candidate key – Key which can uniquely identify a row in the table.

2. Alternate key – If the table has more than one candidate keys and when one becomes a primary key the rest becomes alternate keys.

3. Composite key – More than one key uniquely identifies a row in a table.

**13) What are defaults? Is there a column to which a default can’t be bound?**

1- It is a value that will be used by a column if no value is supplied to that column while inserting data.

2- It can’t be assigned for identity and timestamp values.

create table test

(

empid int,

empname varchar(50) default 'Raj'

)

**14) What is a transaction and what are ACID properties?**

A transaction is a logical unit of work in which, all the steps must be performed or none. ACID stands for Atomicity, Consistency, Isolation, and Durability. These are the properties of a transaction.

**15) What are cursors? Explain different types of cursors. What are the disadvantages of cursors? How can you avoid cursors?**

Cursors allow row-by-row processing of the result sets.

**Types of cursors:**

**Static –** Makes a temporary copy of the data and stores in tempdb and any modifications on the base table does not reflect in data returned by fetches made by the cursor.

**Dynamic –** Reflects all changes in the base table.

**Forward-only –** specifies that cursor can only fetch sequentially from first to last.

**Keyset-driven –** Keyset is the set of keys that uniquely identifies a row is built in a tempdb.

**Disadvantages of cursors:**

Each time you fetch a row from the cursor, it results in a network roundtrip, whereas a normal SELECT query makes only one roundtrip, however large the result set is. Cursors are also costly because they require more resources and temporary storage (results in more IO operations). Further, there are restrictions on the SELECT statements that can be used with some types of cursors.

Most of the times set-based operations can be used instead of cursors.

Here is an example:

If you have to give a flat hike to your employees using the following criteria:

1. Salary between 30000 and 40000 — 5000 hike
2. Salary between 40000 and 55000 — 7000 hike
3. Salary between 55000 and 65000 — 9000 hike

In this situation, many developers tend to use a cursor, determine each employee’s salary and update his salary according to the above formula. But the same can be achieved by multiple update statements or can be combined in a single UPDATE statement as shown below:

UPDATE tbl\_emp SET salary =

CASE

WHEN salary BETWEEN 30000 AND 40000 THEN salary + 5000

WHEN salary BETWEEN 40000 AND 55000 THEN salary + 7000

WHEN salary BETWEEN 55000 AND 65000 THEN salary + 10000

END

Another situation in which developers tend to use cursors: You need to call a stored procedure when a column in a particular row meets a certain conditions. You don’t have to use cursors for this. This can be achieved using WHILE loop, as long as there is a unique key to identify each row.

**16) What is the use of SCOPE\_IDENTITY() function?**

It returns the most recently created identity value for the tables in the current execution scope.

**17) What is the Referential Integrity?**

Referential integrity refers to the consistency that must be maintained between primary and foreign keys, i.e. every foreign key value must have a corresponding primary key value

**18) What is Common Table Expression?**

CTEs also is known as common table expressions are used to create a temporary table that will only exist for the duration of a query. It is a temporary named result set that you can reference within a SELECT, INSERT, UPDATE, or DELETE statement

**Note: It is must to refer the CTE with statement otherwise you will get the error**

* **Query to delete duplicate records**

WITH EmployeesCTE AS

(

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

FROM Employees

)

DELETE FROM EmployeesCTE WHERE RowNumber > 1

* **Query to find duplicate records**

WITH Test as

(

SELECT FirstName, Row\_Number() OVER(ORDER BY FirstName) RowNumber

from Employees

)

SELECT FirstName FROM Test

GROUP BY FirstName HAVING count(FirstName) > 1

**Drawback of CTE**: It is query bound.

**19) Difference between Inner Join and Outer Join**

An outer join, be it the left or right, it has to perform all the work of an inner join along with the additional work null- extending the results. An outer join is expected to return a greater number of records which further increases its total execution time just because of the larger result set.

Thus, an **outer join is slower than an inner join**.

Let us discuss an instance where the Left Join might be faster than the Inner Join. **If the tables involved in the join operation are too small, say they have less than 10 records and the tables do not possess sufficient indexes to cover the query, in that case, the Left Join is generally faster than Inner Join.**

**20) What are the steps you will take to improve the performance of a poor performing query?**

This is a very open-ended question and there could be a lot of reasons behind the poor performance of a query. But some general issues that you could talk about would be:

1. No indexes,
2. Table scans,
3. Missing or out of date statistics,
4. Blocking,
5. Excess recompilations of stored procedures,
6. Procedures and triggers without SET NOCOUNT ON,
7. Poorly written query with unnecessarily complicated joins,
8. Too much normalization,
9. Excess usage of cursors and temporary tables.

Some of the tools/ways that help you troubleshooting performance problems are:

* SET SHOWPLAN\_ALL ON,
* SET SHOWPLAN\_TEXT ON,
* SET STATISTICS IO ON,
* SQL Server Profiler,
* Windows NT /2000 Performance monitor,
* Graphical execution plan in Query Analyzer.

**21) What are statistics, under what circumstances they go out of date, how do you update them?**

**Statistics determine the selectivity of the indexes**. If an indexed column has unique values then the selectivity of that index is more, as opposed to an index with non-unique values. The query optimizer uses these indexes in determining whether to choose an index or not while executing a query.

Some situations under which you should update statistics:

1. If there is a significant change in the key values in the index.
2. If a large amount of data in an indexed column has been added, changed, or removed (that is if the distribution of key values has changed), or the table has been truncated using the TRUNCATE TABLE statement and then repopulated.
3. The database is upgraded from a previous version

**22) What is a deadlock and what is a live lock? How will you go about resolving deadlocks?**

Deadlock is a situation when two processes, each having a lock on one piece of data, attempt to acquire a lock on the other’s piece. Each process would wait indefinitely for the other to release the lock unless one of the user processes is terminated. SQL Server detects deadlocks and terminates one user’s process.

A livelock is one, where a request for an exclusive lock is repeatedly denied because a series of overlapping shared locks keeps interfering. SQL Server detects the situation after four denials and refuses further shared locks. A livelock also occurs when read transactions monopolize a table or page, forcing a write transaction to wait indefinitely.

**23) What is normalization? Explain different levels of normalization?**

It is the way to eliminate redundant data

1. Reduces null value
2. Enables efficient indexing
3. 1NF – Removes duplicated attributes, Attribute data should be atomic, and attribute should be same kind.
4. 2NF – Should be in 1NF and each non-key is fully dependent on the primary key.
5. 3NF – **When it is in 2NF and when all of its attributes are directly dependent on the primary key.** Should be in 2NF and all the non-key attributes which are not dependent on the primary key should be removed. All the attributes which are dependent on the other non-key attributes should also be removed. Normalization is done in OLTP.

**24) SET NOCOUNT ON and SET NOCOUNT OFF**

It will show the status of Transact-SQL statement. In real time, always use SET NOCOUNT ON.

**SET NOCOUNT ON -** It will show "Command(s) completed successfully".

**SET NOCOUNT OFF -** It will show "(No. Of row(s) affected)".

**25)**

sp\_settriggerorder.

**26) Difference between @@identity and scope\_identity()?**

- The **@@identity** function returns the last identity created in the same session.

- The **scope\_identity()** function returns the last identity created in the same session and the same scope.

A situation where the scope\_identity() and the @@identity functions differ, is if you have a trigger on the table. If you have a query that inserts a record, causing the trigger to insert another record somewhere, the scope\_identity() function will return the identity created by the query, while the @@identity function will return the identity created by the trigger.

So, normally you would use the scope\_identity() function.

**27) What is a conjunction table?**

A table that is composed of foreign keys that points to other tables.

**28) What is a relational attribute?**

An attribute that would not exist if it were not for the existence of a relation.

**29) What is a sparse column?**

It is a column that is optimized for holding null values.

**30) What is the transaction log?**

It keeps a record of all activities that occur during a transaction and is used to roll back changes.

**31) What are before images, after images, undo activities and redo activities in relation to transactions?**

Before images refers to the changes that are rolled back if a transaction is rolled back. After images are used to roll forward and enforce a transaction. Using the before images is called the undo activity. Using after images are called the redo activity.

**32) What are shared, exclusive and update locks?**

* A **shared lock** locks a row so that **it can only be read**.
* An **exclusive lock** locks a row so that **only one operation can be performed on it at a time**.
* An **update lock** basically has the ability to **convert a shared lock into an exclusive lock**.

**33) What is LOCK\_TIMEOUT used for?**

It is used for determining the amount of time that the system will wait for a lock to be released.

**34) Pivot in SQL Server**

PIVOT relational operator converts data from row level to column level

CREATE TABLE Employee

(

Name [nvarchar](max),

[Year] [int] ,

Sales [int]

)

INSERT INTO Employee

SELECT 'Pankaj',2010,72500 UNION ALL SELECT 'Rahul',2010,60500 UNION ALL

SELECT 'Sandeep',2010,52000 UNION ALL SELECT 'Pankaj',2011,45000 UNION ALL

SELECT 'Sandeep',2011,82500 UNION ALL SELECT 'Rahul',2011,35600 UNION ALL

SELECT 'Pankaj',2012,32500 UNION ALL SELECT 'Pankaj',2010,20500 UNION ALL

SELECT 'Rahul',2011,200500 UNION ALL SELECT 'Sandeep',2010,32000

**Syntax:**

SELECT <non-pivoted column>,

<list of pivoted column>

FROM

(<SELECT query to produces the data>)

AS <alias name>

PIVOT

(

<aggregation function>(<column name>)

FOR

[<column name that become column headers>]

IN ( [list of pivoted columns])

) AS <alias name for pivot table>

**e.g-1**

SELECT [Year],

Pankaj, Rahul, Sandeep

FROM (SELECT Name, [Year], Sales FROM Employee) Tab1

PIVOT

(

SUM(Sales) FOR Name IN (Pankaj, Rahul, Sandeep)

) AS Tab2

ORDER BY [Tab2].[Year]

|  |  |  |  |
| --- | --- | --- | --- |
| **Year** | **Pankaj** | **Rahul** | **Sandeep** |
| 2010 | 93000 | 60500 | 84000 |
| 2011 | 45000 | 236100 | 82500 |
| 2012 | 32500 | NULL | NULL |

**e.g-2**

SELECT Name,

2010, 2011, 2012

FROM (SELECT Name, [Year], Sales FROM Employee ) Tab1

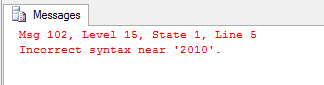
PIVOT

(

SUM(Sales) FOR [Year] IN (2010,2011,2012)

) AS Tab2

ORDER BY Tab2.Name



When we execute the above query, SQL Server throws an error because we can’t provide integer value as a column name directly. To remove this error use the brackets before each integer value as in the following code snippet:

SELECT [Name],

[2010], [2011], [2012]

FROM (SELECT Name, [Year], Sales FROM Employee) Tab1

PIVOT

(

SUM(Sales) FOR [Year] IN ([2010],[2011],[2012])

) AS Tab2

ORDER BY [Tab2].[Name]

**e.g.-3**

In previous example, we used 2010, 2011 and 2012 as pivot columns. But it is not fixed that these column will not change in future. What happens when we get data from year 2013? 2014? or even more?

To solve this problem, we need to use dynamic queries.

First, we retrieve all unique values from a pivot column and after that we will write a dynamic query to execute it with pivot query at run time.

/\*Declare Variable\*/

DECLARE @Pivot\_Column [nvarchar](max);

DECLARE @Query [nvarchar](max);

/\*Select Pivot Column\*/

SELECT @Pivot\_Column= COALESCE(@Pivot\_Column+',','')+ QUOTENAME(Year) FROM

(SELECT DISTINCT [Year] FROM Employee)Tab

/\*Create Dynamic Query\*/

SELECT @Query='SELECT Name, '+@Pivot\_Column+' FROM

(SELECT Name, [Year] , Sales FROM Employee )Tab1

PIVOT

(

SUM(Sales) FOR [Year] IN ('+@Pivot\_Column+')) AS Tab2

ORDER BY Tab2.Name'

/\*Execute Query\*/

EXEC sp\_executesql @Query

**35) UNPIVOT in SQL Server**

UNPIVOT relational operator is reverse process of PIVOT relational operator. UNPIVOT relational operator convert data from column level to row level.

UNPIVOT operation is a reverse process of PIVOT operation, but UNPIVOT is not the exact reverse of PIVOT. If PIVOT performs an aggregation and merges multiple rows into a single row in the output, then UNPIVOT can’t reproduce the original table-valued expression result because rows have been merged. So conclusion is that if PIVOT operation merges multiple row in a single row, then UNPIVOT operation can’t retrieve original table from the output of PIVOT operation. But if PIVOT operation doesn’t merge multiple row in a single row, then UNPIVOT operation can retrieve original table from the output of PIVOT operation.

SELECT Name,[Year] , Sales FROM

(

SELECT [Year],

Pankaj,Rahul,Sandeep

FROM (SELECT Name, [Year] , Sales FROM Employee )Tab1

PIVOT

(

SUM(Sales) FOR Name IN (Pankaj,Rahul,Sandeep)

) AS Tab2

)Tab

UNPIVOT

(

Sales FOR Name IN (Pankaj, Rahul, Sandeep)

) AS TAb2

**CAST AND CONVERT**

Both are used for conversion.

Main difference between them is **Convert** provides you additional style of formatting in case of date time.

For e.g. **select** convert(**varchar**(20),GETDATE(),108) // Here 108 is datetime format

Select convert(varchar,25) + CAST(15 as int) +CAST (10 as varchar) as Result //50

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

CONVERT(VARCHAR,GETDATE(),111) as YYYYMMDD

**Query to get the name of students who are involve in all subject**

select name from NameSubject

group by name

having count(distinct(subject)) = (select count(distinct(subject)) from Namesubject)

**Query to update gender from Male to Female and vice versa**

Update NameSubject set Gender = (case when Gender='F' then 'M' when Gender='M' then 'F' end)

where Gender in('F','M')

**Merge two tables in SQL Server**

MERGE table1 AS target

USING table2 AS source

ON source.id = target.id

WHEN MATCHED THEN

UPDATE SET str\_value = source.str\_value

WHEN NOT MATCHED BY TARGET THEN

INSERT (id, str\_value) VALUES (source.id, source.str\_value)

WHEN NOT MATCHED BY SOURCE THEN

DELETE

**COALESCE** : Return the first non-null value in a list:

SELECT COALESCE(NULL, NULL, NULL, 'W3Schools.com', NULL, 'Example.com');

W3Schools.com

**CAST** Convert a value to a given type datatype:

SELECT CAST(25.65 AS varchar);

**Sequence of commands execution**

**--------------------------------------------------**

**SELECT** yourcolumns  
**FROM** tablenames  
**JOIN** tablenames  
**WHERE** condition  
**GROUP BY** yourcolumns  
**HAVING** aggregatecolumn condition  
**ORDER BY** yourcolumns

**Difference between Temporary table and Table variable**

1. Temporary 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. Table Variable acts like a variable and exists for a particular batch of query execution. It gets dropped once it comes out of batch. It is created in the memory database but may be pushed out to tempdb.
3. Use Table variable, if you have less than 1000 rows otherwise go for Temporary tables.

For e.g.

**------Batch 1 Start------**

DECLARE @TStudent TABLE

(

RollNo INT IDENTITY(1,1),

StudentID INT,

Name varchar(max)

)

--Insert data to Table variable @TStudent

INSERT INTO @TStudent(StudentID,Name)

values(1,'raj')

--Select data from Table variable @TStudent

SELECT \* FROM @TStudent

**------Batch 1 End------**

**------Batch 2 Start------**

SELECT \* FROM @TStudent --gives error

**------Batch 2 End------**

**------Batch 1 Start------**

CREATE TABLE #StudentTemp

(

Name varchar(50),

Address varchar(150)

)

GO

INSERT INTO #StudentTemp VALUES ('Dipendra','Pune');

GO

SELECT \* FROM #StudentTemp

**------Batch 1 End------**

**------Batch 2 Start------**

SELECT \* FROM # StudentTemp --will work

**------Batch 2 End------**