SQL SERVER Indexes

MSBISkills.com

Index – Important Points

1. An index is a structure stored on the disk. This structure is called B+ Trees.
2. An index can only be created on a table or view.
3. They speeds retrieval of rows from the table or view.
4. An index contains keys built from one or more columns in the table or view.
5. These helps SQL Server to find the row or rows associated with the key values quickly and efficiently.
6. B+ Trees are used to store indexes in SQL SERVER.
7. B+ Tree structure always contains at least 2 levels. Root and leaf level. Number of intermediate levels can vary.
8. Every index will take some space in your database.
9. Index will be modified when DML (Insert, update and delete) operation(s) are performed. This means reordering B+ Tree occurs every time the index changes (Updates, Inserts, and Deletes).
10. Index size can be of maximum of 900 bytes.

UPDATE from EXPERT - Kind of. It's really that the KEY of an index can be 900 bytes OR 16 columns – whichever comes first. But, even that's not always true. If a table has an XML column then the PK can have a maximum of 15 columns AND the PK *must* be the CL key.

11. If the table is too small then don't create indexes on it. In this case we will be wasting our resources because table scan will be faster than index scan.

    UPDATE from EXPERT - Every table should have indexes – even small tables. Indexes can be used to enforce uniqueness and allow for point queries (which do not require index scans). Having said that, I suppose the only case would be a table that has only one page. But, you don't really care about performance for that table anyway (as far as access). What you might care about there is locking. Indexes can be beneficial to allow SQL Server to lock only the necessary row(s).

12. You can use ALT + F1 to check out what all indexes are created on a table.

13. Indexes are a lot of "trial and error" thing depending on database design, SQL queries, and database size.

Types of Indexes

1. Clustered Index
2. NonClustered Index

Clustered Index

0. Clustered index sort and store the data rows based on their key values. These are the columns included in the index definition.
1. We can have only one clustered index per table. We cannot have more than one clustered index per table.
2. Because data rows can be sorted in only one order.
3. At leaf level you will get entire data. Means even if you create clustered index on a single column, at leaf level you will get all the columns means actual data.

4. Clustered Index will be modified when DML (Insert, update and delete) operation(s) are performed. This means reordering B+ Tree occurs every time the index changes (Updates, Inserts, and Deletes). Here physical ordering of the data changes after each DML operation.

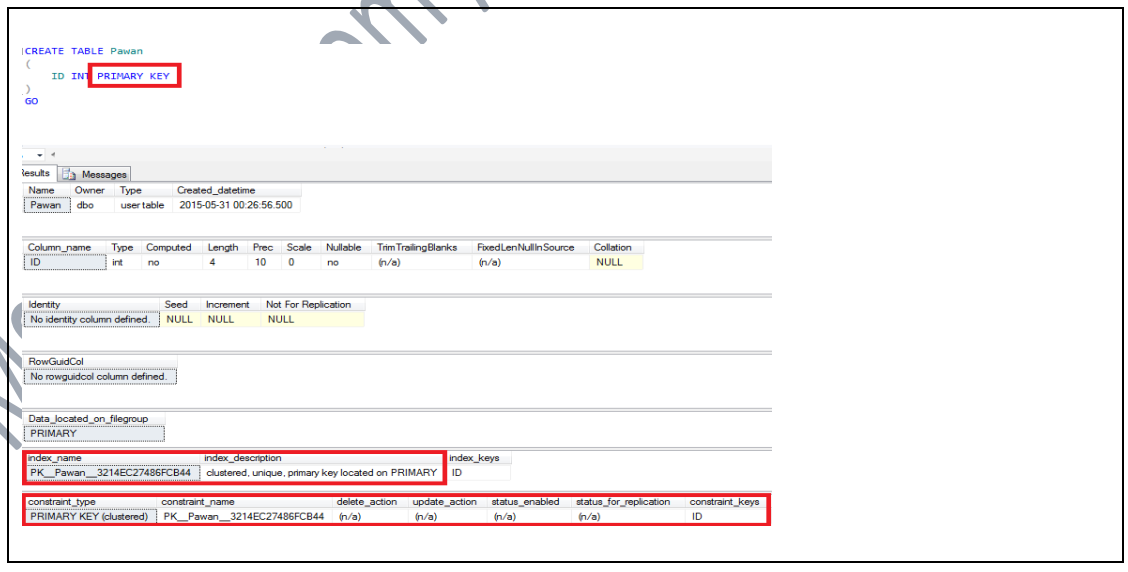
5. If a table has Clustered Index it is called Clustered Table.

6. If a table does not contain Clustered Index, then it is stored in Heap. Heap is an unordered structure.

7. Clustered index can be unique or non-unique.

8. If you create primary key, SQL Internally creates unique clustered index and primary key constraint. Primary key is basically a constraint used to enforce uniqueness in a table. The primary key columns cannot hold NULL values. Please check out the example below.

   UPDATE from EXPERT - It's really that the PK defaults to being enforced by a unique clustered index. None of the columns that make up the primary key can allow nulls (or, be nullable). However, if you have a unique constraint then ALL of the columns that make up the unique constraint can allow nulls but no more than one row can be NULL for all columns. With a PK NONE of the columns can allow NULL values.



9. If you just create a unique clustered index, one null value will be allowed in the table.

10. Note – Do not use GUID as Clustered Index. It will create fragmentation issues.

11. Don't create Clustered Index blindly. It should be created based on the business need. Try to make Clustered key's size small and if possible integer type. Other simple rule is column should be somewhat ordered and frequently queried column. For e.g. EmplD in Employee table.

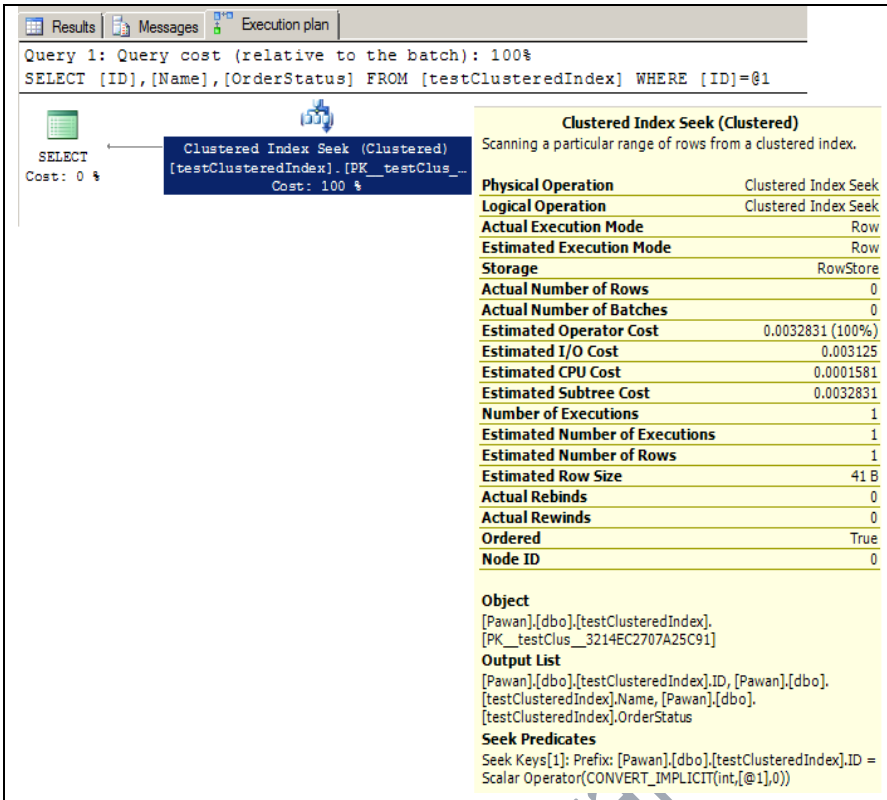
    Also please note that these are just simple tips, there might be scenarios where you to choose different data type or multi column clustered key.

    So all in all I can say choose wisely.

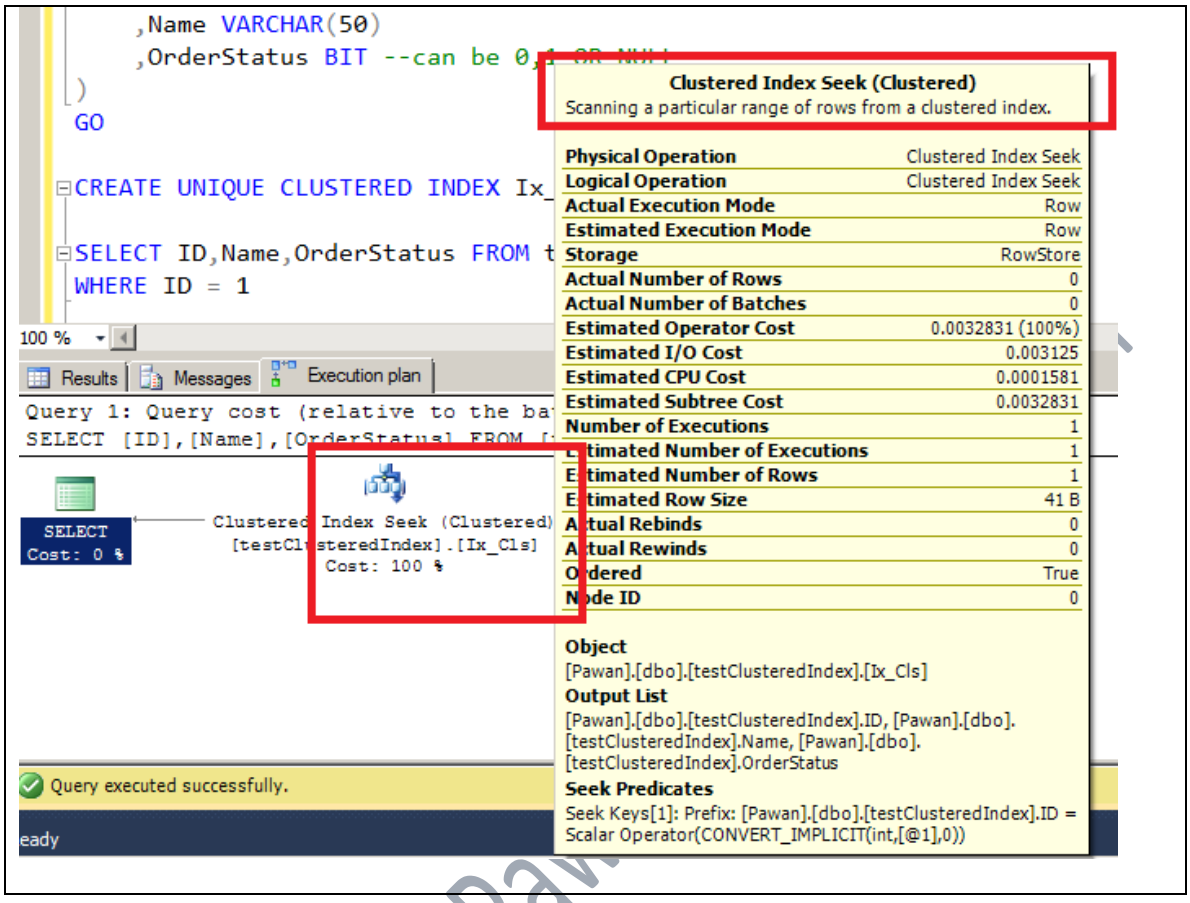
12. We can create Clustered Index with multiple columns also. These types of indexes are called Composite Index.

13. The example of clustered index is given below.

```
--OPTION 1

CREATE TABLE testClusteredIndex
(
       ID INT PRIMARY KEY
      ,Name VARCHAR(50)
      ,OrderStatus BIT --can be 0,1 OR NULL
)
GO

SELECT ID,Name,OrderStatus FROM testClusteredIndex
WHERE ID = 1

Output

--------------
```




```
--OPTION 2

CREATE TABLE testClusteredIndex
(
       ID INT
      ,Name VARCHAR(50)
      ,OrderStatus BIT --can be 0,1 OR NULL
)
GO

CREATE UNIQUE CLUSTERED INDEX Ix_Cls ON testClusteredIndex(ID)

SELECT ID,Name,OrderStatus FROM testClusteredIndex
WHERE ID = 1

Output

--------------
```



14. We can change fill factor while creating Clustered Index. See example below.

```
CREATE UNIQUE CLUSTERED INDEX Ix_Inx ON Pawan(ID)
WITH (FILLFACTOR=80)
```

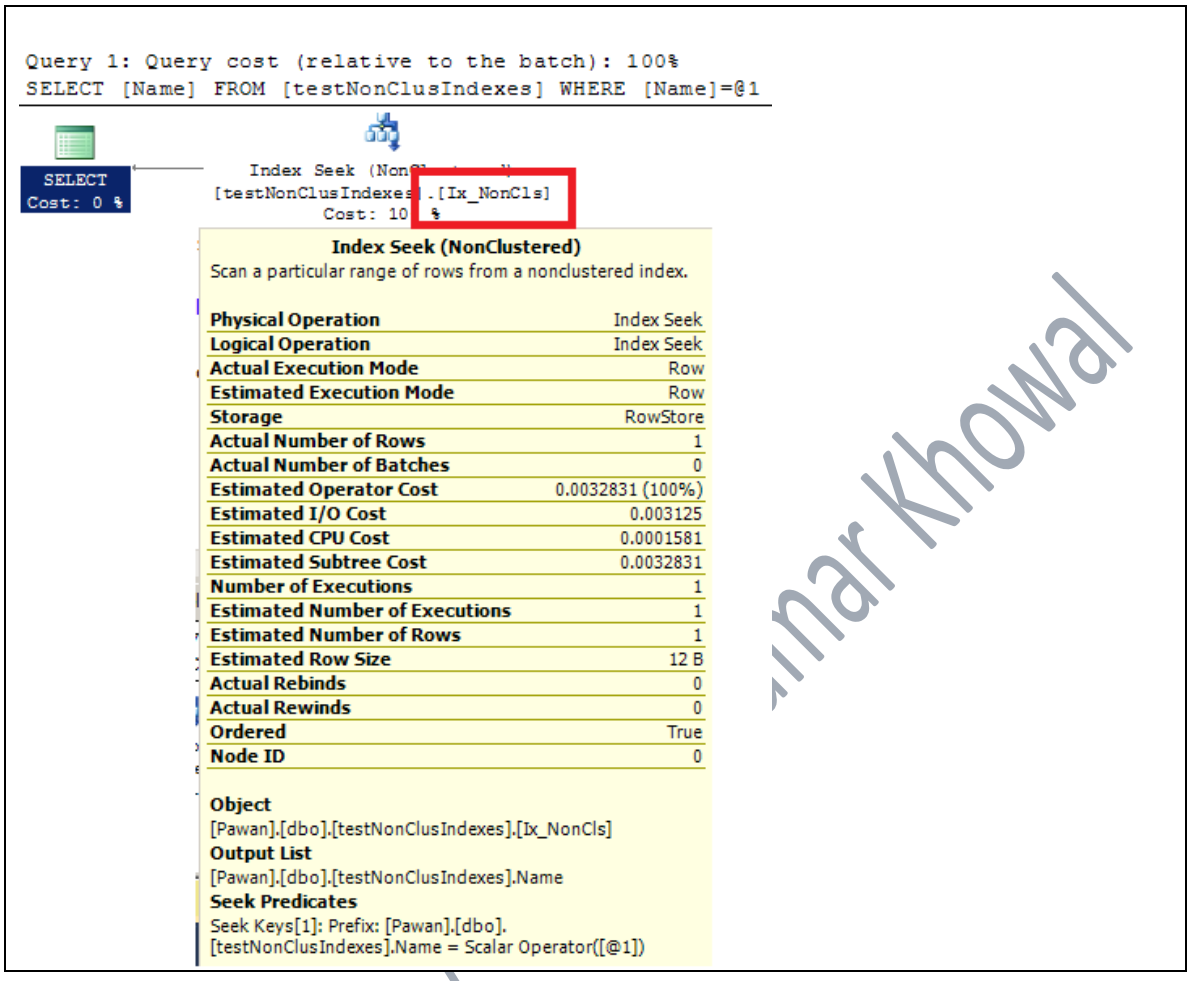
NonClustered Index

1. A NonClustered index contains index key values and each key value entry has a row locator.

2. The structure of the row locator depends on whether the data pages are stored in a heap or a clustered table.

3. For a clustered table, the row locator is the clustered index key.
4. For a heap, a row locator is a RID pointer to the row.
5. NonClustered indexes can be unique or non-unique.
6. Indexes are internally updated after each DML operation (Insert, Update & Delete).
7. Data is not physically sorted order here.
8. We can have maximum 999 non clustered index per table.
9. A NonClustered index does not affect the ordering and storing of the data.
10. We can create duplicate NonClustered Indexes obviously with different name but we should not because indexes will be updated after every DML operation and we have pay for their maintenance also. Extra Indexes will lead to CPU and disk overhead, so craft indexes carefully and test them thoroughly.
11. The example of NonClustered index is given below.

```
CREATE TABLE testNonClusIndexes
(
       ID INT
      ,Name VARCHAR(50)
      ,OrderStatus BIT --can be 0,1 OR NULL
)
GO

INSERT INTO testNonClusIndexes VALUES(1,'Pawan',1),(2,'Isha',0)

CREATE NONCLUSTERED INDEX Ix_NonCls ON testNonClusIndexes(Name)

SELECT Name FROM testNonClusIndexes
WHERE Name = 'Pawan'

Output
-----------
```



12. We can change fill factor while creating NonClustered Index. See example below.

```
CREATE UNIQUE NONCLUSTERED INDEX Ix_NonInx_WithFillFactor ON
Pawan(ID) WITH (FILLFACTOR = 90)
```

13. Even If you create a NonClustered primary key, SQL does not allow a single NULL value. E.g. below

```
CREATE TABLE Pawan
(
      ID INT PRIMARY KEY NONCLUSTERED
)
GO
```

```
INSERT INTO Pawan(ID) VALUES(NULL)
```

Output

```
Msg 515, Level 16, State 2, Line 1
Cannot insert the value NULL into column 'ID', table
'AdventureWorks2012.dbo.Pawan'; column does not allow nulls.
INSERT fails.
The statement has been terminated.
```

14. Use NonClustered indexes on foreign Keys that are used in joins.

Clustered Index Vs NonClustered Index

Sr.No	Clustered Index	NonClustered Index
1	Here data is physically sorted based on the Key columns	Here data is not physically sorted based on the key columns
2	At leaf level you will get entire data means data for all columns. Even if you create clustered index on a single column all the columns will be available at leaf level.	Here at leaf level you will get Key value and row locator. For a clustered table, the row locator is the clustered index key. For a heap, a row locator is a RID pointer to the row.
3	We can have only 1 clustered index per table	We can have 999 NonClustered index per table.

Other Index Types

1. NonClustered Index with Included columns
2. Covering Index

NonClustered Index with Included columns

1. A NonClustered index can be extended by including nonkey columns in addition to the index key columns. The nonkey columns are stored at the leaf level of the index b+ Tree.

2. The general syntax of a Non Clustered Index with Included column is given below

```
CREATE INDEX <Index_Name> ON <table> (KeyColumns) INCLUDE
(NonKeyColumns)
```

3. **KeyColumns** – These columns are used for row restriction and processing E.g they were used in WHERE, JOIN, ORDER BY, GROUP BY etc.

4. **NonKeyColumns** – These columns are used in SELECT and Aggregation. For e.g. AVG(col) after selection/restriction.

5. So always choose KeyColumns and NonKeyColumns based on the query requirements only.

6. Please note that you cannot add a column in Key column section and a NonKey column section. It is either a key column or a non-key, included column.

7. A nonkey column will only be stored at the leaf level, however. There are some benefits of using non-key columns.

Columns can be accessed without extra lookups (RID lookup / Clustered Key lookup). It will reduce IO operation and improve performance of queries.

Included columns do not count against the 900 byte index key limit enforced by SQL Server.

8. Data types not allowed in key columns are allowed in nonkey columns. All data types but text, ntext, and image are allowed.

9. Example of NonClustered Index with Included column is given below-

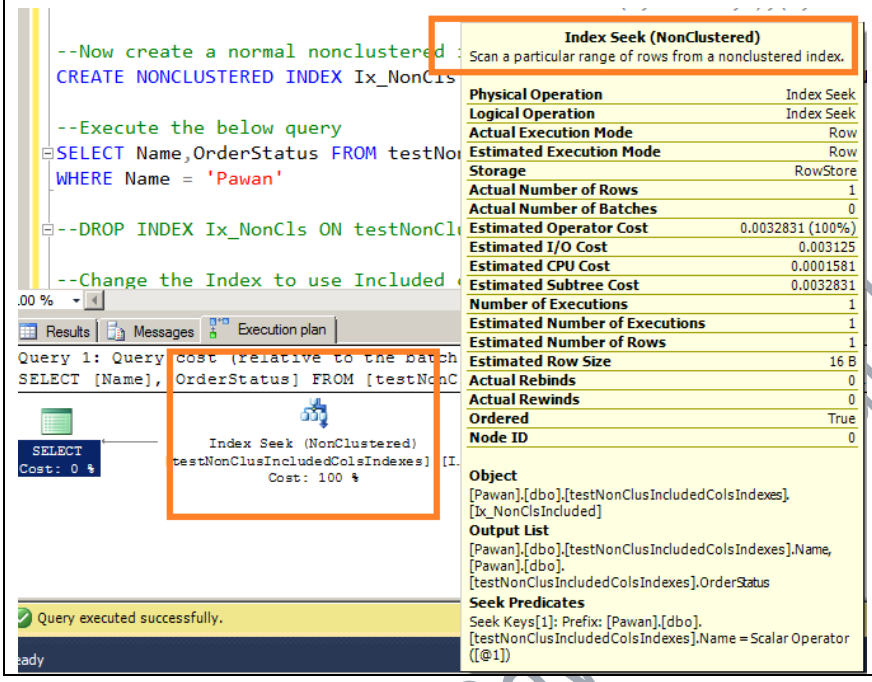
```
--Create table
CREATE TABLE testNonClusIncludedColsIndexes
(
       ID INT
      ,Name VARCHAR(50)
      ,OrderStatus BIT --can be 0,1 OR NULL
)
GO

--Insert some rows
INSERT INTO testNonClusIncludedColsIndexes
VALUES(1,'Pawan',1),(2,'Isha',0)

--Now create a normal nonclustered index
CREATE NONCLUSTERED INDEX Ix_NonCls ON
testNonClusIncludedColsIndexes(Name)

--Execute the below query
SELECT Name,OrderStatus FROM testNonClusIncludedColsIndexes
WHERE Name = 'Pawan'
```

--Now here you can see that even you have NonClusteredIndex still table scan is used. Now let's drop the above index and create NonClusteredIndex with Included column.

```
DROP INDEX Ix_NonCls ON testNonClusIncludedColsIndexes

CREATE NONCLUSTERED INDEX Ix_NonClsIncluded ON
testNonClusIncludedColsIndexes(Name) INCLUDE(OrderStatus)
```

Execute the below query and check the execution plan.

```
--Now execute and check the execution plan
SELECT Name,OrderStatus FROM testNonClusIncludedColsIndexes
WHERE Name = 'Pawan'
```



10. Included columns do not count against the 900 byte index key limit enforced by SQL Server. Please check out the above example where we have a column in Include column with size greater than 900 byte.

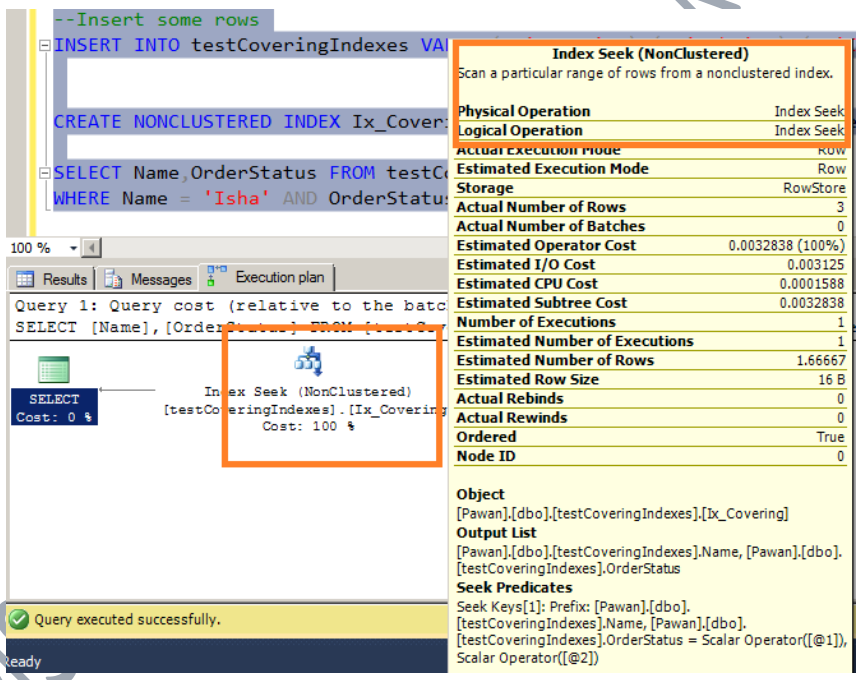
Covering Index

1. In covering index all columns returned in a query are in the available in the index, so no additional reads are required to get the data.
2. A covering index will reduce the IO operations and improve performance of queries.
3. Please find the example of covering index below.

```
CREATE TABLE testCoveringIndexes
(
       ID INT
      ,Name VARCHAR(50)
```

```
      ,OrderStatus BIT -- can be 0,1 OR NULL
)
GO

--Insert some rows
INSERT INTO testCoveringIndexes
VALUES(1,'Pawan',1),(2,'Isha',0),(2,'Isha',0),(2,'Nisha',1),(2,'I
sha',0)


CREATE NONCLUSTERED INDEX Ix_Covering ON
testCoveringIndexes(Name,OrderStatus)

SELECT Name,OrderStatus FROM testCoveringIndexes
WHERE Name = 'Isha' AND OrderStatus = 0
```

Output



4. Now let's check whether can create index size greater than 900 bytes.
   NOTE – yes we can but a warning will come.

```
CREATE TABLE Pawan
(
       ID INT
```

```
      ,Name VARCHAR(100)
      ,Addre VARCHAR(1000)
)
GO

CREATE UNIQUE NONCLUSTERED INDEX Ix_CoveringInx ON
Pawan(Name,Addre)
```

Output

```
---------
Warning! The maximum key length is 900 bytes. The index
'Ix_CoveringInx' has maximum length of 1100 bytes. For some
combination of large values, the insert/update operation will
fail.
```

Filtered Index

1. If you add a where clause to a NonClustered index it is called Filtered Index.

2. When you create a NonClustered index, you can add a WHERE clause to reduce the number of rows that are stored at the leaf level.

3. If we have fewer rows in an index then less I/O will be used and it will improve query performance.

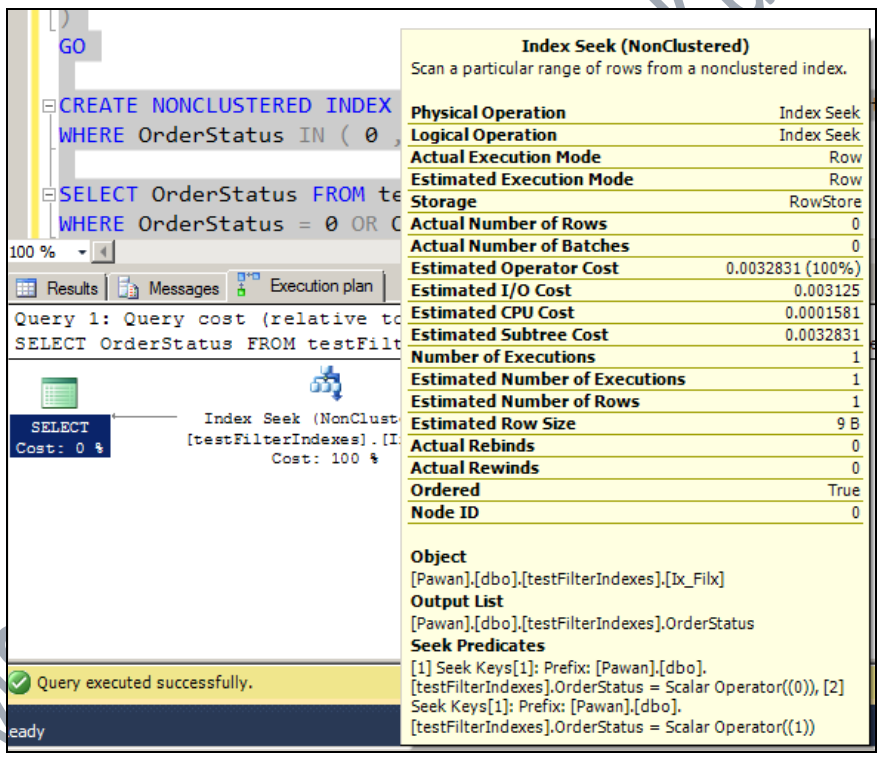
4. Also the size of the index will be smaller since we have fewer rows in the index. Hence less space will be used by the Index.

5. Filtered indexes are excellent for performance if you have a value that is used in a where clause very frequently and that value is only a small amount of the total values for that table.

6. The example of Filtered index is given below.

```
CREATE TABLE testFilterIndexes
(
       ID INT
      ,Name VARCHAR(50)
      ,OrderStatus BIT --can be 0,1 OR NULL
)
GO

CREATE NONCLUSTERED INDEX Ix_Filx ON
testFilterIndexes(OrderStatus)
WHERE OrderStatus IN ( 0 , 1 )

SELECT OrderStatus FROM testFilterIndexes
WHERE OrderStatus = 0 OR OrderStatus = 1
```

Output



7. You cannot use OR in filtered indexes. Please check out the example below.

```
CREATE NONCLUSTERED INDEX Ix_Filx1 ON
testFilterIndexes(OrderStatus)
WHERE OrderStatus =  0 OR  OrderStatus =  1


Output
--------

Msg 156, Level 15, State 1, Line 2
Incorrect syntax near the keyword 'OR'.
```

8. Please check out some sample Filtered Indexes below. We can use AND, IN, NULL, NOT NULL.

```
CREATE NONCLUSTERED INDEX Ix_Filx1 ON
testFilterIndexes(OrderStatus)
WHERE OrderStatus =  0 AND OrderStatus =  1


CREATE NONCLUSTERED INDEX Ix_Filx2 ON
testFilterIndexes(OrderStatus)
WHERE OrderStatus IN ( 0 , 1 )


CREATE NONCLUSTERED INDEX Ix_Filx3 ON
testFilterIndexes(OrderStatus)
WHERE OrderStatus IS NOT NULL


CREATE NONCLUSTERED INDEX Ix_Filx4 ON
testFilterIndexes(OrderStatus)
WHERE OrderStatus IS NULL
```


9. You cannot use BETWEEN, NOT IN, or a CASE statement with Filtered Indexes.
10. The query optimizer won't use filtered indexes if you're using local variables or parameterized SQL. Use the way we have used our dynamic parameterized queries given below.

```
SELECT OrderStatus FROM testFilterIndexes
WHERE OrderStatus = 0


DECLARE @SQL NVARCHAR(MAX), @OrderStatus INT

SET @OrderStatus = 0

SET @SQL = N' SELECT OrderStatus FROM testFilterIndexes WHERE
OrderStatus =' + CAST(@OrderStatus AS VARCHAR(10))

EXECUTE sp_executesql @SQL
```

Final Comment-

Indexes are very easy to add to your database to improve performance. However, too much of an indexes can be bad as we have to pay their maintenance cost. When designing a database, or troubleshooting poor performing query, consider all your indexes carefully and test them thoroughly.