Tuning SQL query performance

Test questions

en:

1. What functions does the query optimizer perform?

2. What is the purpose of the indexes?

3. Compare the **Estimated execution plan** with **Actual execution plan** .

ru:

1. Какие функции выполняет оптимизатор запросов?

2. Каково назначение индексов?

3.Сравните **предполагаемый план выполнения** с **действитель-ным планом выполнения**.

Contents

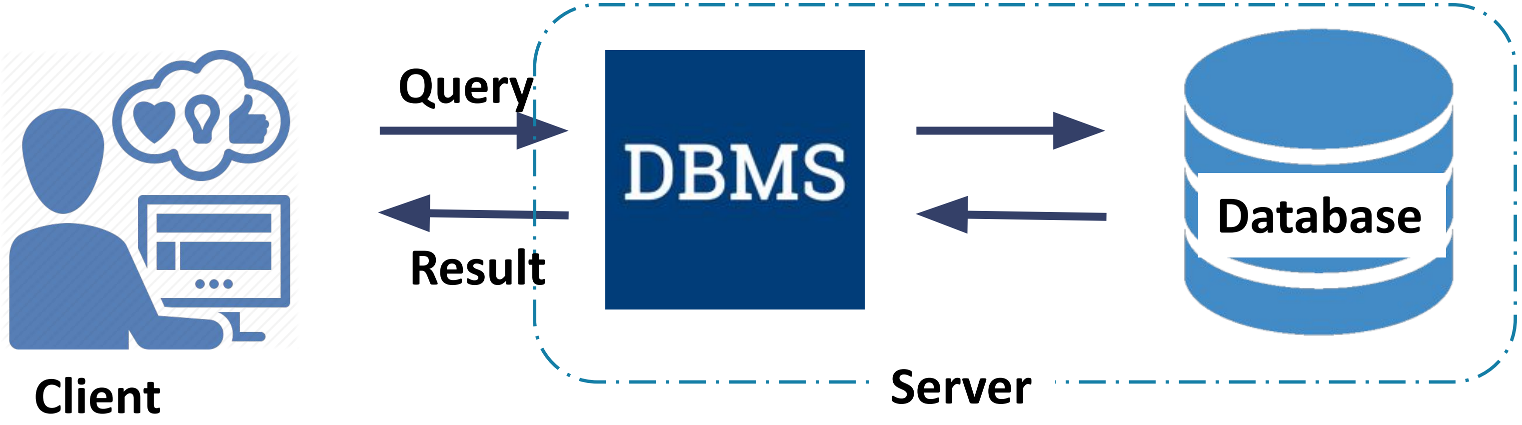
1. Query Processing

1.1. End User Interaction with DBMS

End users interact with the DBMS through the use of queries to generate information,

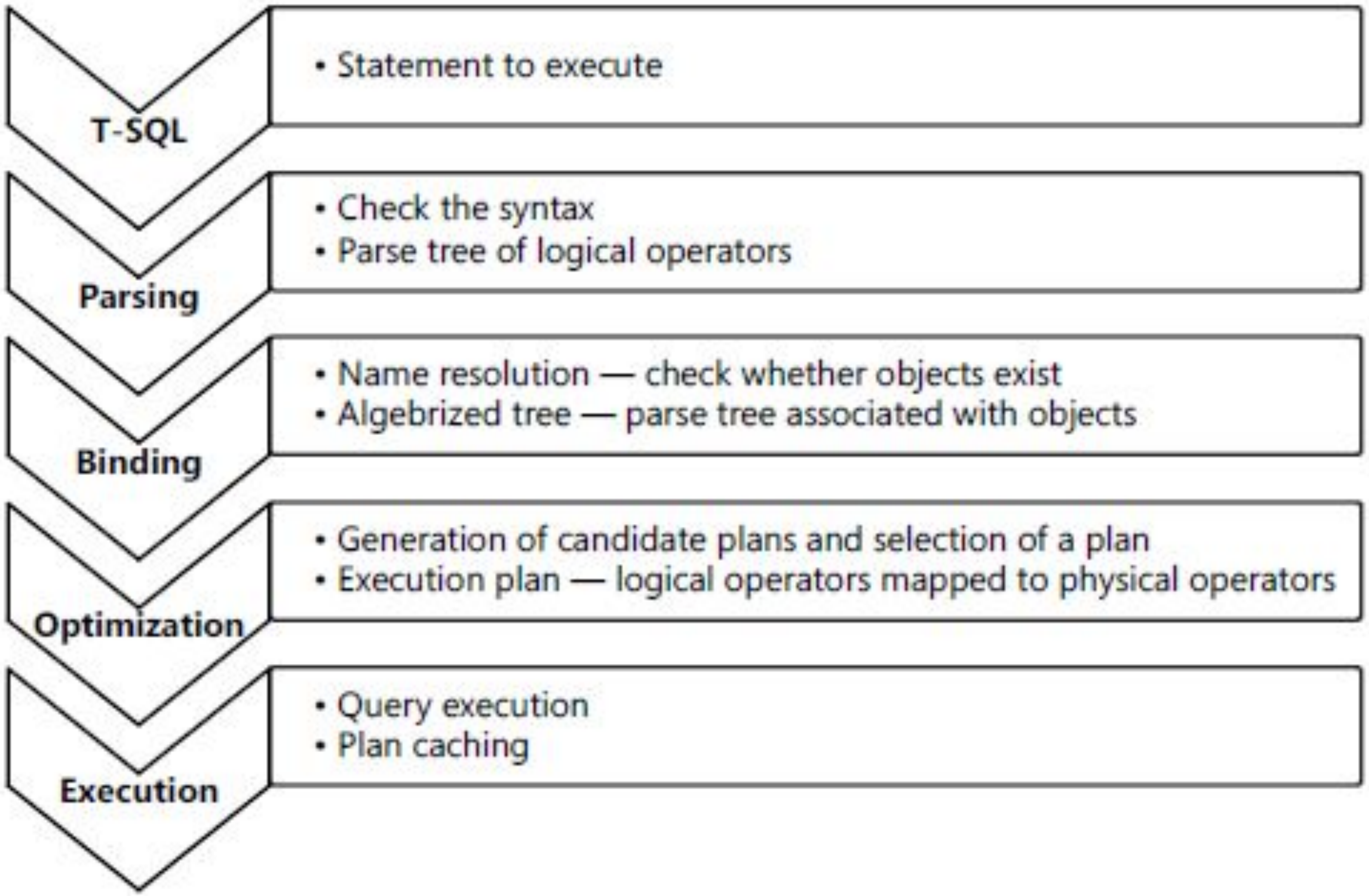
using the following sequence:
1. The end-user application generates a query.
2. The query is sent to the DBMS.
3. The DBMS executes the query.
4. The DBMS sends the resulting data set to the end-user application.

The goal of database performance is to execute queries as fast as possible



1.2. Query Processing

Query Processing include translations on high level Queries into low level expressions that can be used at physical level of file system, query optimization and actual execution of query to get the actual result.



1.3. Query Optimization

Importance: The goal of query optimization is to reduce the system resources required to fulfill a query, and ultimately provide the user with the correct result set faster.

1. It provides the user with faster results, which makes the application seem **faster to the user**.

2. It allows the system to service **more queries** in the same amount of time, because each request takes less time than unoptimized queries.

3. Query optimization ultimately reduces the amount of wear on the hardware (e.g. disk drives), and allows the server to run more efficiently (e.g. **lower power consumption**, **less memory usage**).

1.4. Query Optimizer

A single query can be executed through different algorithms or re-written in different forms and structures. Hence, the question of query optimization comes into the picture – Which of these forms or pathways is the most optimal? The query optimizer attempts to determine the most efficient way to execute a given query by considering the possible query plans.

The process of searching and evaluating various options (that is, **different candidate execution plans**) for fulfilling the query occurs at the optimization phase using the **Query Optimizer**.

It selects the best plan for the next phase. The **actual execution plan** is a single tree with physical operators.

1.5. Cost of Execution Plan

Query Optimizer is often a cost-based optimizer. It assigns a number called **cost** to each **possible plan**. A higher cost means a more complex plan, and a more complex plan means a slower query.

Query Optimizer calculates the cost of an operation by determining the algorithm used by a physical operator and by estimating the number of rows that have to be processed. The estimation of the number of rows is also called **cardinality estimation**. The cost expresses usage of physical resources such as the amount of disk I/O, CPU time, and memory needed for execution.

For calculating the cost, the Query Optimizer needs some information for the estimation of the **number of rows** processed by each physical operator. The Query Optimizer gets this information from optimizer **statistics**. DBMS maintains statistics about the total number of rows and distribution of the number of rows over key values of an index for each index.

After the Query Optimizer gets the cost for all operators in a plan, it can calculate the **cost of the whole plan**.

1.6. Query Optimization Issues

Since database structures are **complex**, in most cases, and especially for not-very-simple queries, the needed data for a query can be collected from a database by accessing it **in different ways**, through **different data-structures**, and in **different orders**.

Each different way typically requires **different processing time**. Processing times of the same query may have **large variance**, from a **fraction of a second** to **hours**, depending on the **way selected**.

The **purpose** of query optimization, which is an automated process, is to find the way to process a given query in **minimum time**. The large possible variance in time justifies performing query optimization, though finding the **exact optimal way** to execute a query, among all possibilities, is typically very **complex**, time consuming by itself, may be **too costly**, and often **practically impossible**.

Because the number of possible plans grows in a factorial way with query complexity, it is impossible to generate and check all possible plans for complex queries. The Query Optimizer balances between plan quality and time needed for the optimization. Therefore, the Query Optimizer cannot guarantee that the best possible plan is always selected.

Thus query optimization typically tries to approximate the optimum by comparing several common-sense alternatives to provide in a reasonable time a "**good enough**" plan which typically does not deviate much from the best possible result.

2. Database Indexes

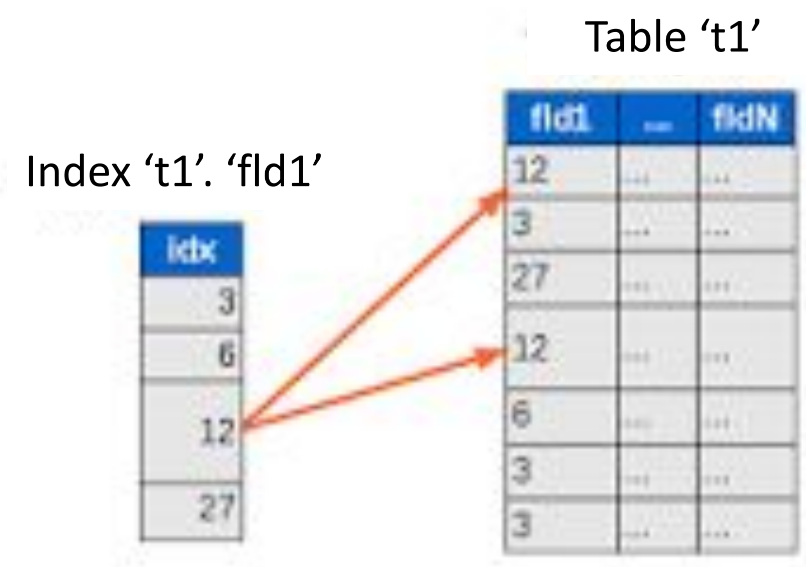
2.1. Database Index Concept

A **database index** is a **data** structure that improves the speed of **data** retrieval operations on a **database** table at the cost of additional writes and storage space to maintain the **index data** structure.

Tables in the database can have **a large number of rows** that are stored **in random order**, and it can take a lot of time to search them according to a specified criterion by sequentially viewing the table **row by row**.

The index is formed from the **values** of one or more **columns** of the table and **pointers** to the corresponding rows of the table and, thus, allows you to search for rows that meet the search criteria.

Acceleration of work using indexes is achieved primarily due to the fact that the index has a structure optimized for search - for example, a **balanced tree**.



2.2. Types of Indexes

Clustered indexes

Clustered indexes *sort* and ***store*** the data rows in the table or view based on their **key values**. These are the columns included in the index definition.

There can be only **one** clustered index per table, because the data rows themselves can be stored in only one order.

The only time the data rows in a table are stored in sorted order is when the table contains a clustered index.

When a table has a clustered index, the table is called a **clustered table**. If a table has no clustered index, its data rows are stored in an unordered structure called a **heap**.

2.2. Types of Indexes

Nonclustered indexes

Nonclustered index contains the nonclustered **index key** values and each key value entry has a **pointer** to the data row that contains the key value.

The pointer from an index row in a nonclustered index to a data row is called a **row locator**. The structure of the row locator depends on whether the data pages are stored in a heap or a clustered table. For a heap, a row locator is a pointer to the row. For a clustered table, the row locator is the clustered index key.

When you create a table with a **UNIQUE** constraint, Database Engine automatically creates a **nonclustered** index.

When you try to enforce a PRIMARY KEY constraint on an existing table and a clustered index already exists on that table, SQL Server enforces the primary key using an **nonclustered** index.

2.3. Create Indexes

Clustered indexes

When you create a table with a **Primary Key**, SQL Server automatically creates a corresponding clustered index based on columns included in the primary key.

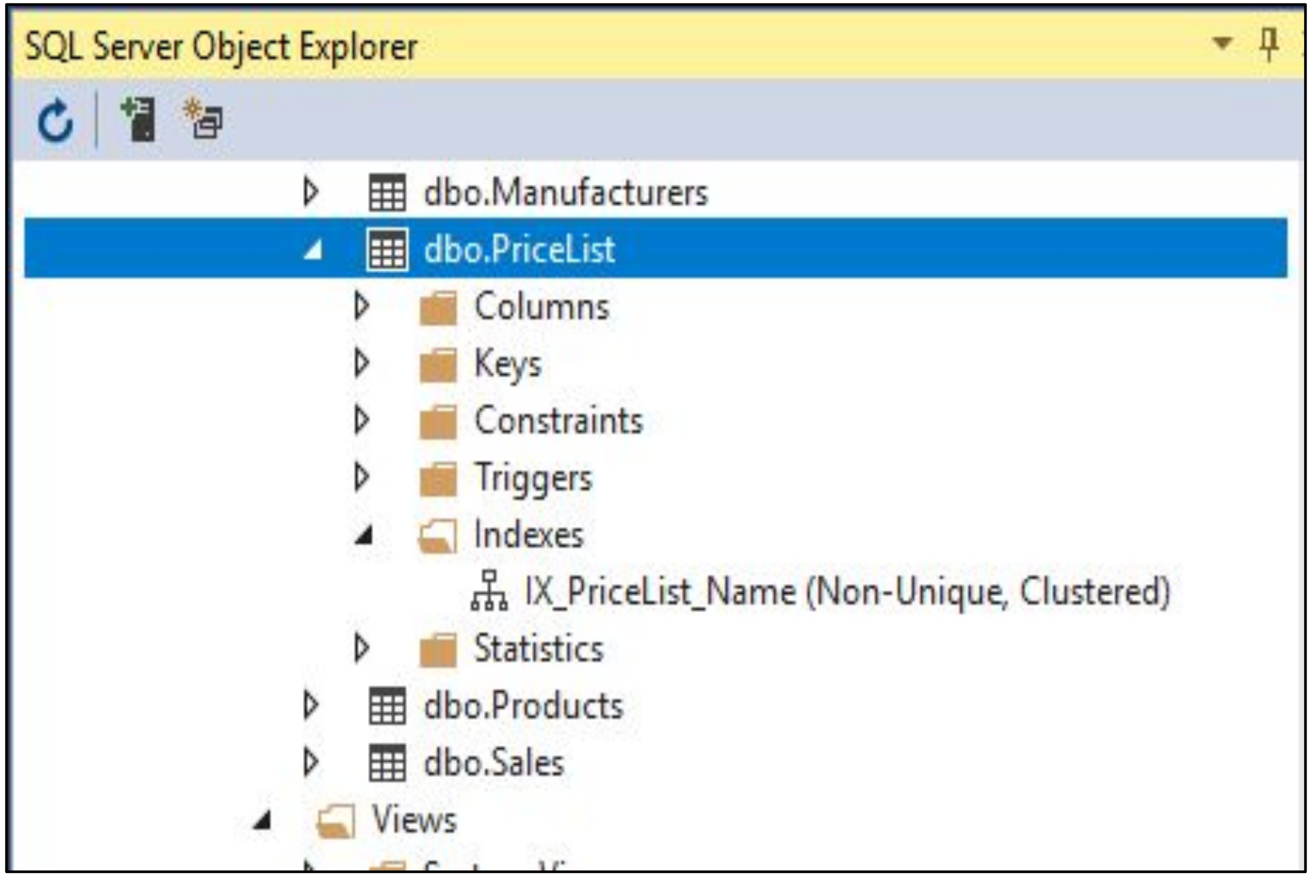
In case a table **does not have a primary key**, which is very rare, you can use the CREATE CLUSTERED INDEX statement to define a clustered index for the table.

```
CREATE [UNIQUE] INDEX index_name
ON table_name (column1, column2, ...)
```

Example. For the PriceList (Name, Price) table :

CREATE CLUSTERED INDEX IX_PriceList_Name
ON PriceList (Name);

In Visual Studio 2017:

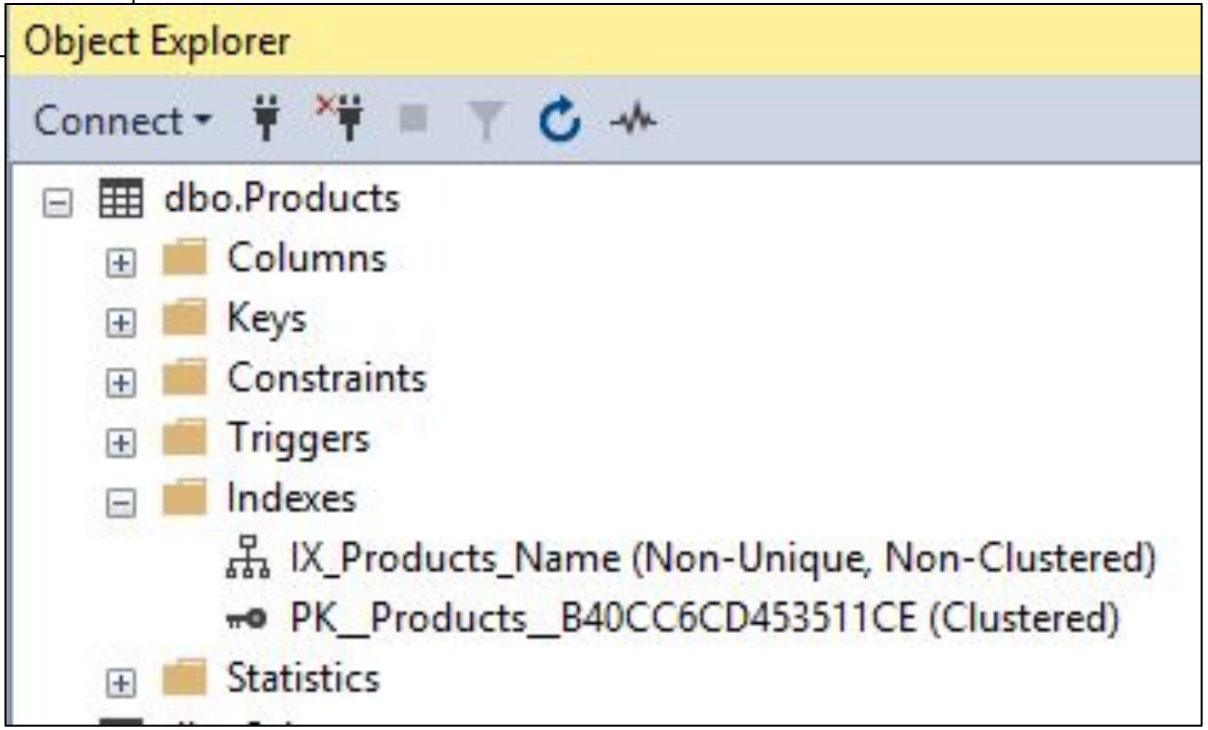


2.3. Create Indexes

Nonclustered indexes

```
CREATE [ UNIQUE ] [NONCLUSTERED ] INDEX index_name
ON <object> ( column_name [ ASC | DESC ] [ ,...n ] )
```

In SSMS 2017:



Example. For the Products table :

```
CREATE INDEX IX_Products_Name
ON Products (Name);
```

2.4. Drop Index

DROP INDEX *table_name.index_name*;

Example. For the Products table :

DROP INDEX Products.IX_Products_Name;

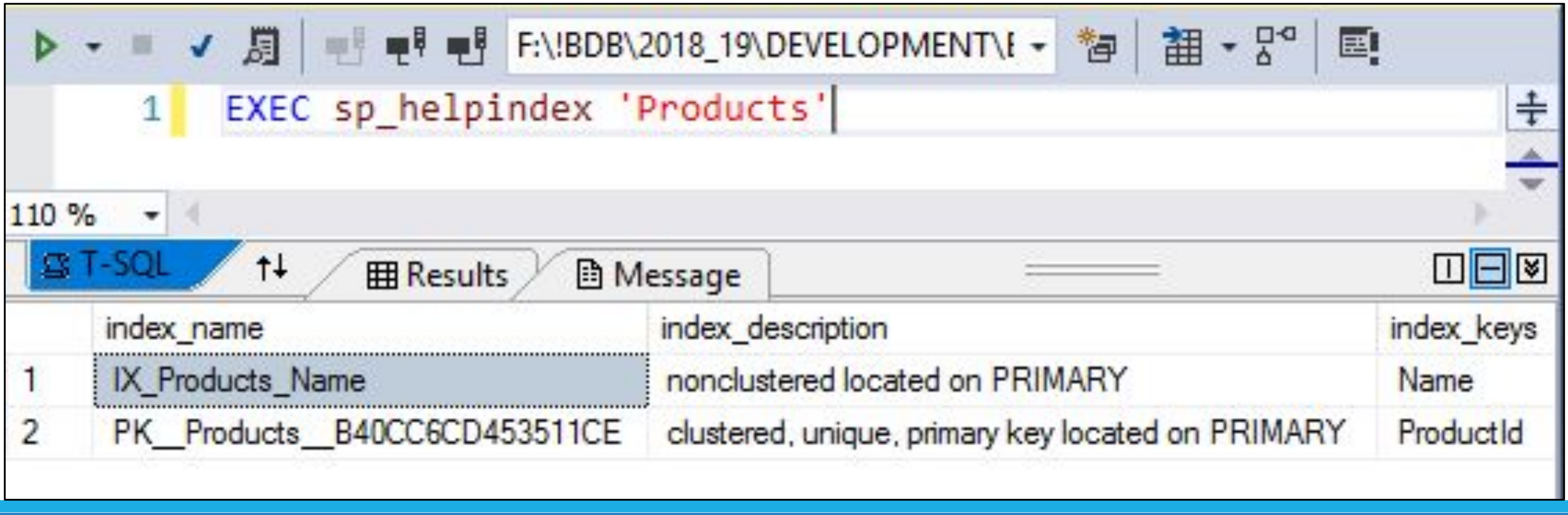
Note. Indexes that are created as the result of creating **PRIMARY KEY** or **UNIQUE** constraints cannot be dropped by using DROP INDEX. They are **dropped** using the **ALTER TABLE DROP CONSTRAINT** statement.

2.5. Looking for indexes

sp_helpindex is a system stored procedure which lists the information of **all the indexes on a table** or view. sp_helpindex returns the name of the index, description of the index and the name of the column on which the index was created.

```
EXEC sp_helpindex
'[[[SCHEMA-NAME.TABLE-NAME]]]'
```

Example.

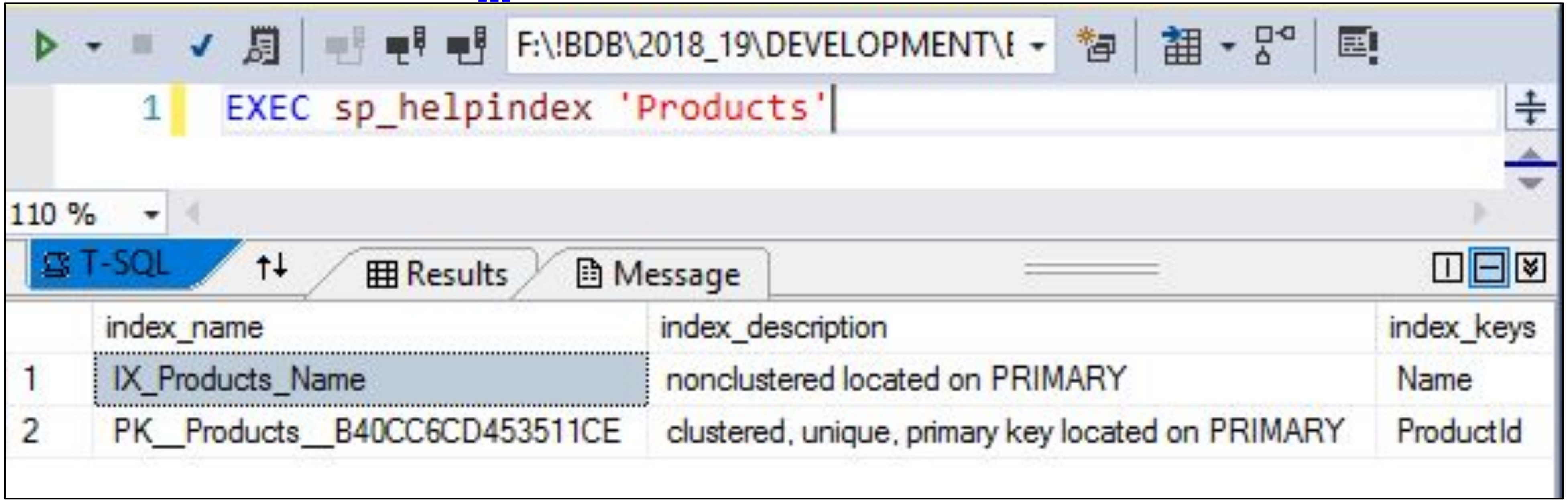


2.5. Looking for indexes

sp_helpindex is a system stored procedure which lists the information of **all the indexes on a table** or view. sp_helpindex returns the name of the index, description of the index and the name of the column on which the index was created.

```
EXEC sp_helpindex
'[[[SCHEMA-NAME.TABLE-NAME]]]'
```

Example.



3. Query Analysis Tools

3.1. STATISTICS IO

STATISTICS IO will tell you the cost of the query in terms of the actual number of **physical reads** from disk, **logical reads** from memory on query and **read-ahead reads** asnumber of pages placed into the cache for the query by SQL Servers 'Read-ahead' mechanism.

SET STATISTICS IO { ON | OFF }

Example.

```
DBCC DROPCLEANBUFFERS; -- Clear cache data
SET STATISTICS IO  ON
SELECT Sale_date, Name, Quantity
FROM Sales JOIN Products ON Sales.ProductId = Products.ProductId
SET STATISTICS IO OFF
```

Message:

```
Table 'Sales'. Scan count 1, logical reads 87, physical reads 1, read-ahead reads 85,
lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.
Table 'Products'. Scan count 1, logical reads 2, physical reads 1, read-ahead reads 0,
lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.
```

3.2. STATISTICS TIME

Displays the number of milliseconds required to parse, compile, and execute each statement.

```
SET STATISTICS TIME { ON | OFF }
```

Example.

```
DBCC DROPCLEANBUFFERS; -- Clear cache data
SET STATISTICS TIME  ON
SELECT Sale_date, Name, Quantity
FROM Sales JOIN Products ON Sales.ProductId = Products.ProductId
SET STATISTICS TIME OFF
```

Message:

```
SQL Server Execution Times:
  CPU time = 62 ms,  elapsed time = 490 ms.
```

3.3. Types of Execution Plans

Execution plans can tell you how a query **will** be executed, or how a query **was** executed.

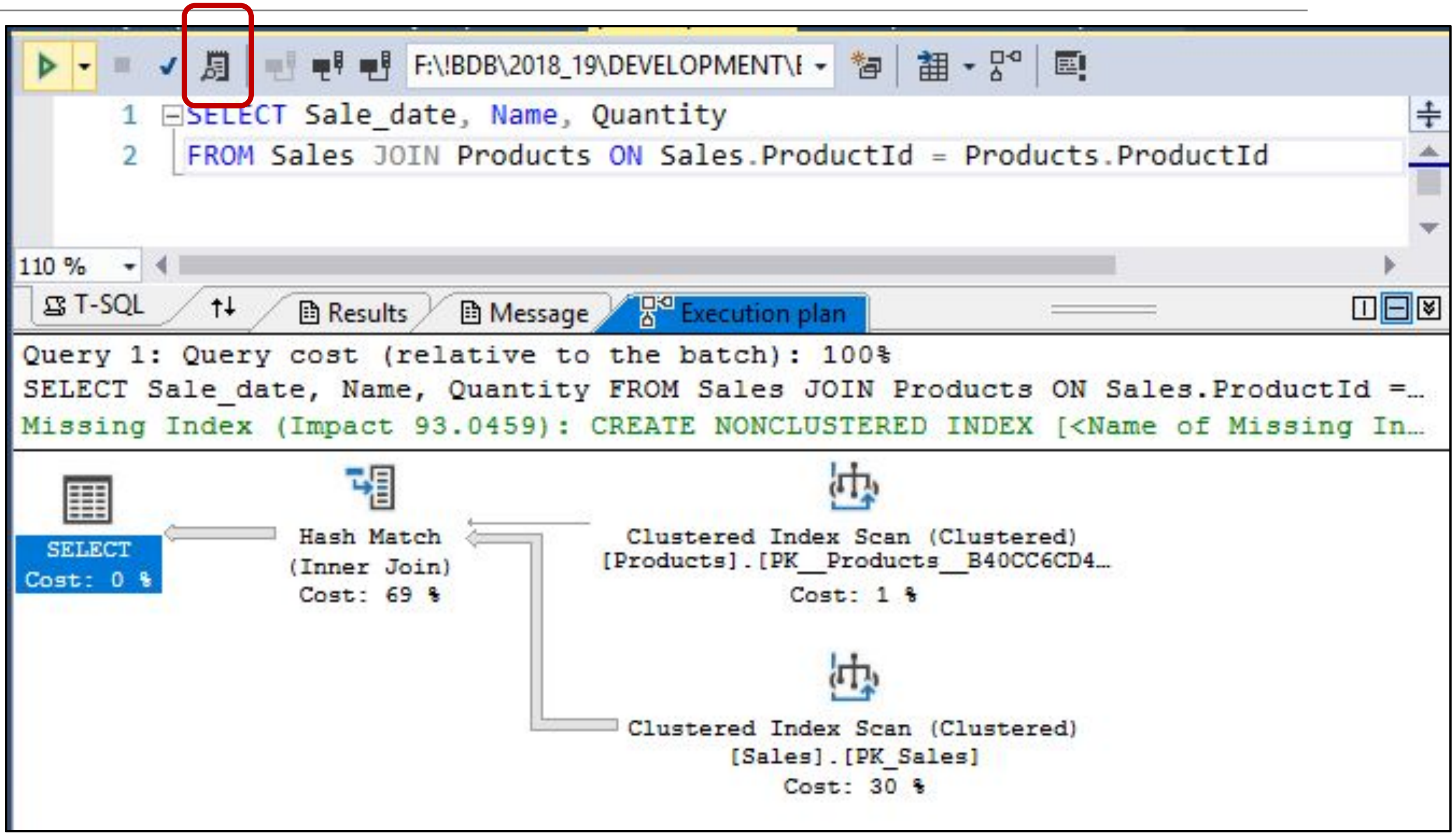
Estimated execution plan is the plan that represents the **output from the optimizer**. The operators, or steps, within the plan will be labelled as **logical**, because they're representative of the optimizer's view of the plan.

Actual execution plan is represents the **output from the actual query execution**. It shows what actually happened when the query executed.

The main cause of a **difference** between the **plans** is differences between the **statistics** and the **actual** data. This generally occurs over time as data is added and deleted. This causes the key values that define the index to change, or their distribution (how many of what type) to change. This means that, over time, the statistics become a less-and-less accurate reflection of the actual data.

3.4. Estimated Execution Plans

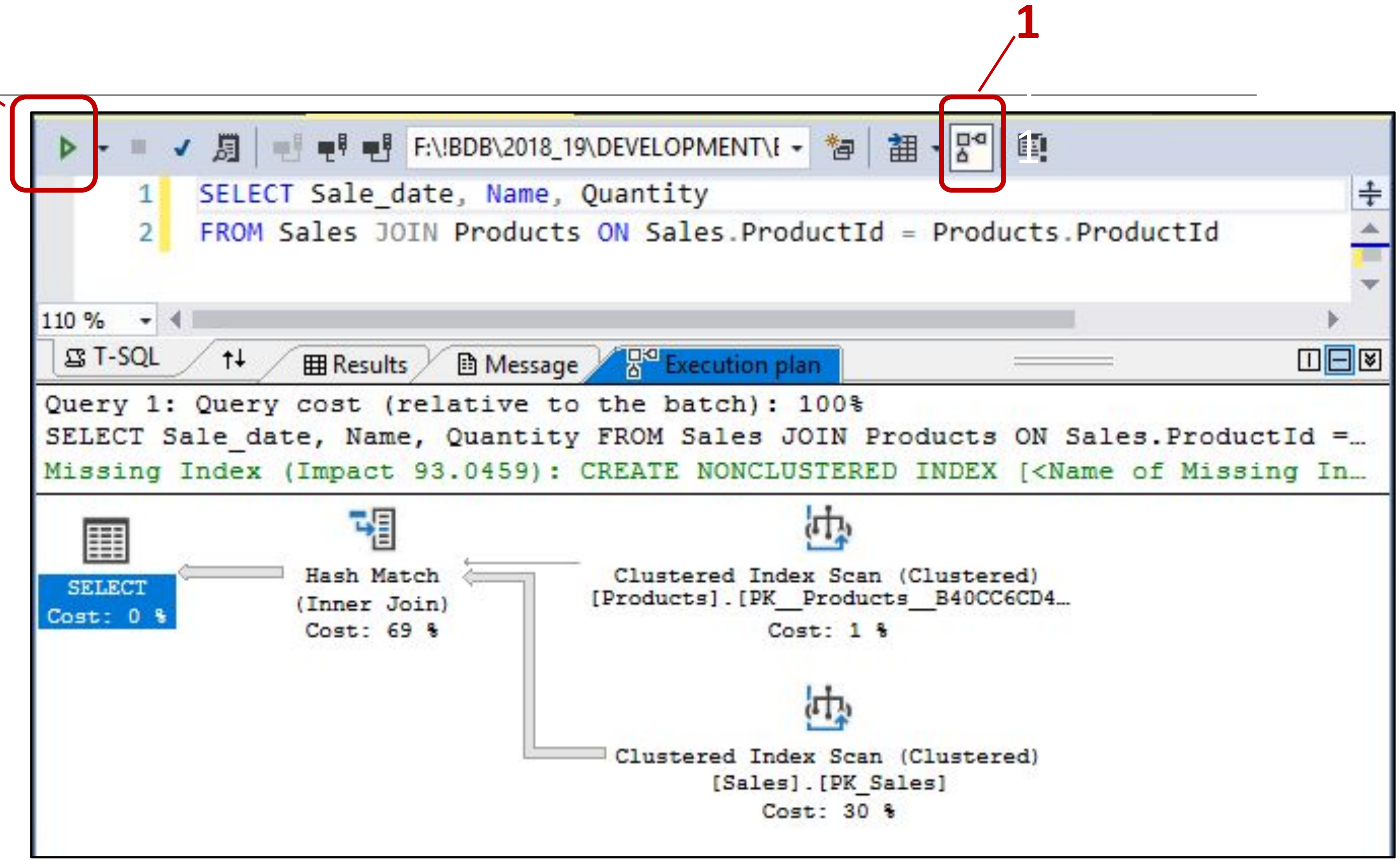
In the **Query Editor** window, click the **Display Estimated Execution Plan** icon on the tool bar.



3.5. Estimated Execution Plans

1. In the **Query Editor** window, click the **Include Actual Execution Plan** icon on the tool bar.

2. Click the **Execute** icon



3.6. Reading the Execution Plan

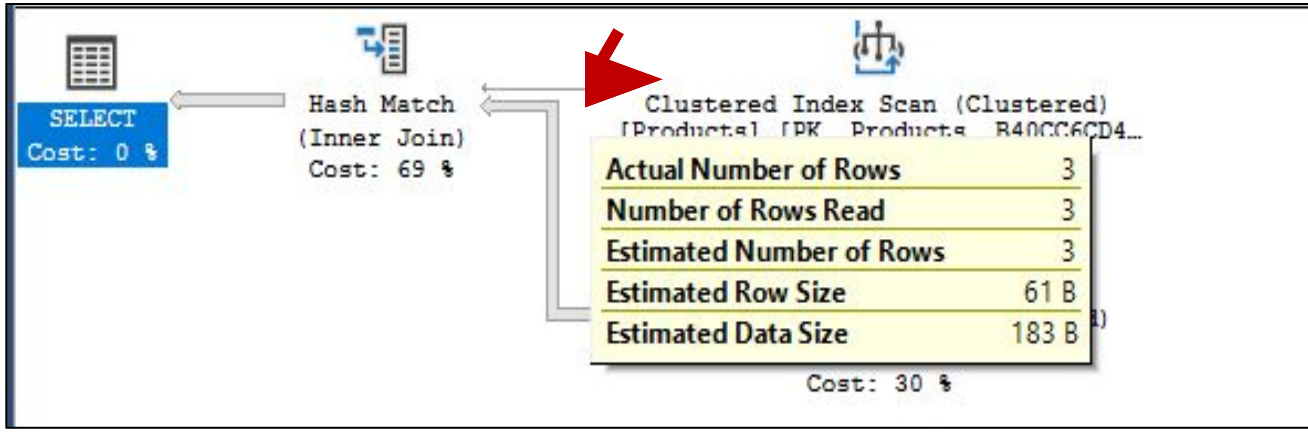
Usually, you read a graphical execution plan from **right** to **left** and **top** to **bottom**.

The **arrows** represent the **data** transmitted between the **operators** in the form of **icons**.

The **thickness** of the arrow reflects the **amount** of data being passed, thicker meaning more rows.

If you hover over these arrows, it will show the **number of rows** that it represents.

Below each icon is displayed a number as a percentage. It represents the **relative cost** to the query for that operator (the estimated execution time).



3.7. Types of Execution Plans

Execution plans can tell you how a query **will** be executed, or how a query **was** executed.

Estimated execution plan is the plan that represents the **output from the optimizer**. The operators, or steps, within the plan will be labelled as **logical**, because they're representative of the optimizer's view of the plan.

Actual execution plan is represents the **output from the actual query execution**. It shows what actually happened when the query executed.

The main cause of a **difference** between the **plans** is differences between the **statistics** and the **actual** data. This generally occurs over time as data is added and deleted. This causes the key values that define the index to change, or their distribution (how many of what type) to change. This means that, over time, the statistics become a less-and-less accurate reflection of the actual data.

3.8. Operator Descriptions

Image	Operator	Description
	Table Scan	Retrieves all rows from the specified table; can be a costly operation if the table has huge number of rows.
	Clustered Index Seek	Most optimized method to retrieve the data; engine uses index keys to look up required rows.
	Clustered Index Scan	Same as table scan; it occurs when the engine determines that it is not a time saver if the available index key is not enough to retrieve the data and almost all rows need to be returned.
	RID Lookup	It is a bookmark lookup and occurs on a heap table; uses row identifier to return the corresponding rows.
	Key Lookup	Key Lookup is a bookmark lookup on a table with a clustered index. It occurs when the engine has to use index key to retrieve the corresponding row.
	Nested Loops	Joins two set of data using scanning outer data set once for each row in the inner data set.
	Merge Join	Joins two tables when joining columns are already presorted.

https://docs.microsoft.com/en-us/sql/relational-databases/showplan-logical-and-physical-operators-reference?view=sql-server-ver15

4. Query tuning practice

4.1. Define business requirements before starting

- **Identify relevant stakeholders**. (All involved parties + DBA)
- **Focus on business outcomes**. Be sure the query has a definite and unique purpose.
- **Prepare a discussion for good requirements**. Define the function and scope of the report, specifying the intended audience. This will focus the query on tables with the right level of detail.
- **Develop good requirements by asking great questions**. Those questions typically follow the 5 W's – Who? What? Where? When? Why?
- **Write very specific requirements and confirm them with stakeholders.** The performance of the production database is too critical to have unclear or ambiguous requirements.

4.2. Avoid SELECT * in Your Queries

DBMS should **scan column names** and replace * with actual table columns.

Instead of:

`DBCC DROPCLEANBUFFERS;`
`SET STATISTICS TIME ON`
`SELECT * FROM Sales`
`SET STATISTICS TIME OFF`
	SQL Server Execution Times:
	CPU time = 32 ms, elapsed time = 619 ms.

use:

`DBCC DROPCLEANBUFFERS;`
`SET STATISTICS TIME ON`
`SELECT Sale_date, ManufacturerId,ProductId, Quantity`
`FROM Sales`
`SET STATISTICS TIME OFF`
	SQL Server Execution Times:
	CPU time = 16 ms, elapsed time = 515 ms.

4.2. Avoid SELECT * in Your Queries

DBMS should **scan column names** and replace * with actual table columns.

Instead of:

`DBCC DROPCLEANBUFFERS;`
`SET STATISTICS TIME ON`
`SELECT * FROM Sales`
`SET STATISTICS TIME OFF`
	SQL Server Execution Times:
	CPU time = 32 ms, elapsed time = 619 ms.

use:

`DBCC DROPCLEANBUFFERS;`
`SET STATISTICS TIME ON`
`SELECT Sale_date, ManufacturerId,ProductId, Quantity`
` FROM Sales`
`SET STATISTICS TIME OFF`
	SQL Server Execution Times:
	CPU time = 16 ms, elapsed time = 515 ms.

4.3. Avoid DISTINCT in SQL Queries

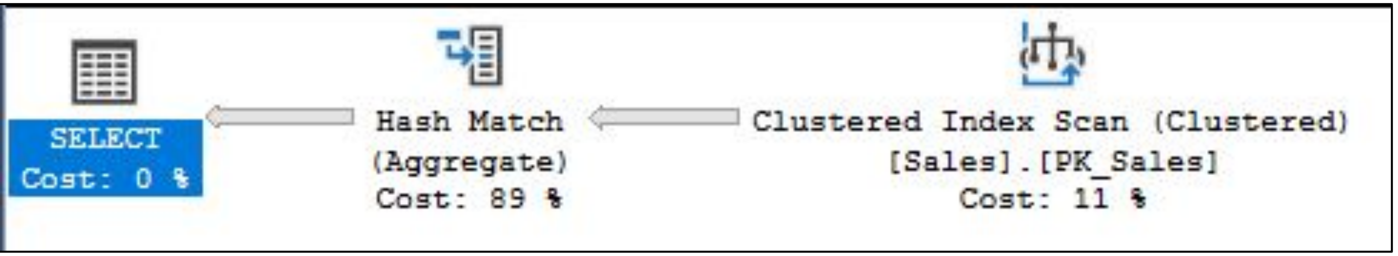
SELECT DISTINCT is a handy way to remove duplicates from a query.
SELECT DISTINCT works by **GROUP**ing all fields in the query to create distinct results.

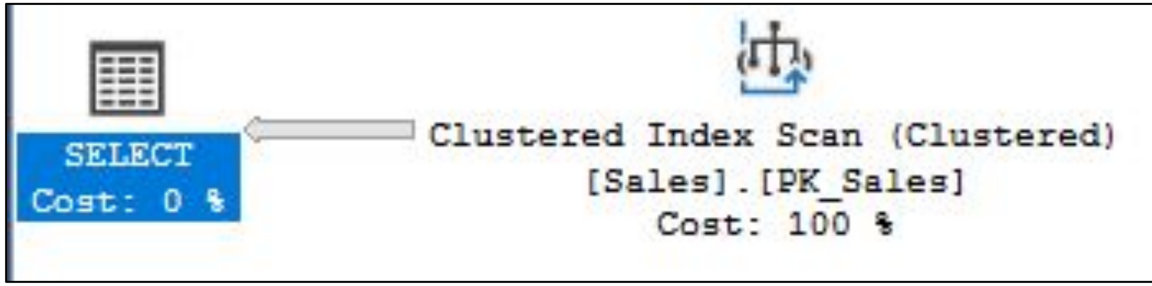
Instead of:

```
SELECT DISTINCT Sale_date, ManufacturerId, Quantity
FROM Sales
```

use:

```
SELECT Sale_date, ManufacturerId, ProductId, Quantity
FROM Sales
```





4.4. Create Joins with INNER JOIN Rather than WHERE

In some databases, this type of queries are inefficient as it first creates temp data with all possible options (most probably **CROSS JOIN**) and then it applies WHERE conditions.

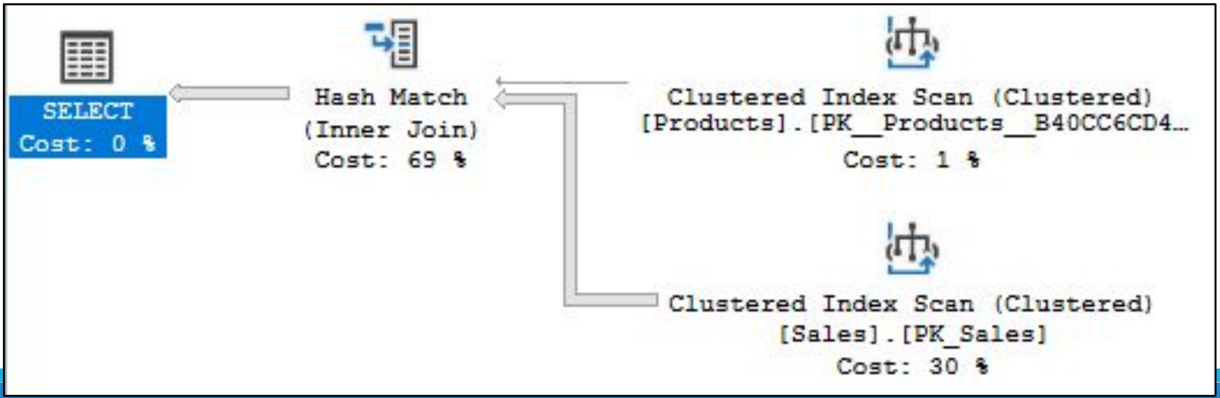
Instead of:

```
SELECT Sale_date, Name, Quantity
FROM Sales, Products
WHERE Sales.ProductId = Products.ProductId
```

use:

```
SELECT Sale_date, Name, Quantity
FROM Sales JOIN Products ON Sales.ProductId = Products.ProductId
```

In SQL Server, they are equivalent



4.5. Create Clustered and Non-Clustered Indexes

Practice to create **clustered and non-clustered index** since indexes helps in to **access data fastly**.

But be careful, more indexes on a table will slow the INSERT, UPDATE, DELETE operations.

Hence try to keep small no of indexes on a table.

Example. Optimize performance of the query

```
SELECT
   SalesId, ProductId, Quantity
FROM  Sales
WHERE  ProductId = 1;
```

Steps:

1. Check indexes on the Sales table
2. Simplified query without non-clustered indexes
3. Add non-clustered index on ProductId
4. Add new Quantity field in SELECT
5. Include columns

Example

1. Check indexes on the Sales table

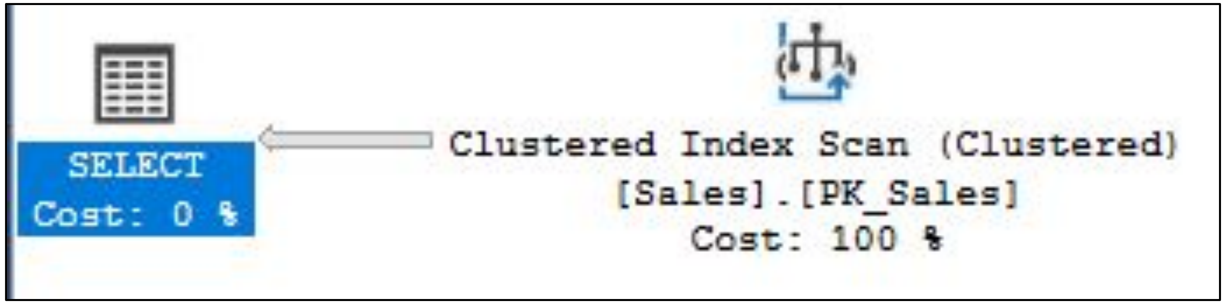
```
EXEC sp_helpindex 'Sales'
```

	index_name	index_description	index_keys
1	PK_Sales	clustered, unique, primary key located on PRIMARY	SaleId

Example

2. Simplified query without non-clustered indexes

```sql
SELECT SalesId, ProductId
FROM  Sales
WHERE  ProductId = 1;
```



Clustered Index Scan (Clustered)

Scanning a clustered index, entirely or only a range.

Physical Operation	Clustered Index Scan
Logical Operation	Clustered Index Scan
Estimated Execution Mode	Row
Storage	RowStore
Estimated I/O Cost	0.0653472
Estimated Operator Cost	0.0943825 (100%)
Estimated CPU Cost	0.0290353
Estimated Subtree Cost	0.0943825
Estimated Number of Executions	1
Estimated Number of Rows	8750
Estimated Number of Rows to be Read	26253
Estimated Row Size	15 B
Ordered	False
Node ID	0

Predicate

[F:\!BDB\2018_19\DEVELOPMENT\EN\WORK\11PERFORMANCE\PROJECTS\APPPETRENKO\APPPETRENKO\BREADPETRENKO.MDF].[dbo].[Sales].[ProductId]=CONVERT_IMPLICIT(int,[@1],0)

Object

[F:\!BDB\2018_19\DEVELOPMENT\EN\WORK\11PERFORMANCE\PROJECTS\APPPETRENKO\APPPETRENKO\BREADPETRENKO.MDF].[dbo].[Sales].[PK_Sales]

Output List

[F:\!BDB\2018_19\DEVELOPMENT\EN\WORK\11PERFORMANCE\PROJECTS\APPPETRENKO\APPPETRENKO\BREADPETRENKO.MDF].[dbo].[Sales].SaleId, [F:\!BDB\2018_19\DEVELOPMENT\EN\WORK\11PERFORMANCE\PROJECTS\APPPETRENKO\APPPETRENKO\BREADPETRENKO.MDF].[dbo].[Sales].ProductId

Example

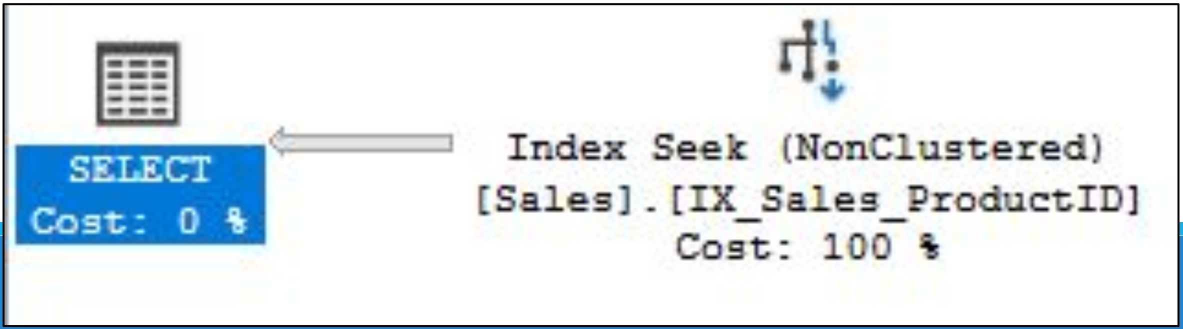
3. Add non-clustered index on ProductId

```sql
CREATE INDEX IX_Sales_ProductID
ON Sales(ProductID);
```

```sql
EXEC sp_helpindex 'Sales'
```

	index_name	index_description	index_keys
1	IX_Sales_ProductID	nonclustered located on PRIMARY	ProductId
2	PK_Sales	clustered, unique, primary key located on PRIMARY	SaleId

```sql
SELECT SaleId, ProductId
FROM Sales
WHERE ProductId = 1;
```



Index Seek (NonClustered)

Scan a particular range of rows from a nonclustered index.

Physical Operation	Index Seek
Logical Operation	Index Seek
Estimated Execution Mode	Row
Storage	RowStore
Estimated Operator Cost	0.023523 (100%)
Estimated I/O Cost	0.013741
Estimated Subtree Cost	0.023523
Estimated CPU Cost	0.009782
Estimated Number of Executions	1
Estimated Number of Rows	8750
Estimated Number of Rows to be Read	8750
Estimated Row Size	15 B
Ordered	True
Node ID	0

Object

[F:\!BDB\2018_19\DEVELOPMENT\EN\WORK\11PERFORMANCE\PROJECTS\APPPETRENKO\APPPETRENKO\BREADPETRENKO.MDF].[dbo].[Sales].[IX_Sales_ProductID]

Output List

[F:\!BDB\2018_19\DEVELOPMENT\EN\WORK\11PERFORMANCE\PROJECTS\APPPETRENKO\APPPETRENKO\BREADPETRENKO.MDF].[dbo].[Sales].SaleId, [F:\!BDB\2018_19\DEVELOPMENT\EN\WORK\11PERFORMANCE\PROJECTS\APPPETRENKO\APPPETRENKO\BREADPETRENKO.MDF].[dbo].[Sales].ProductId

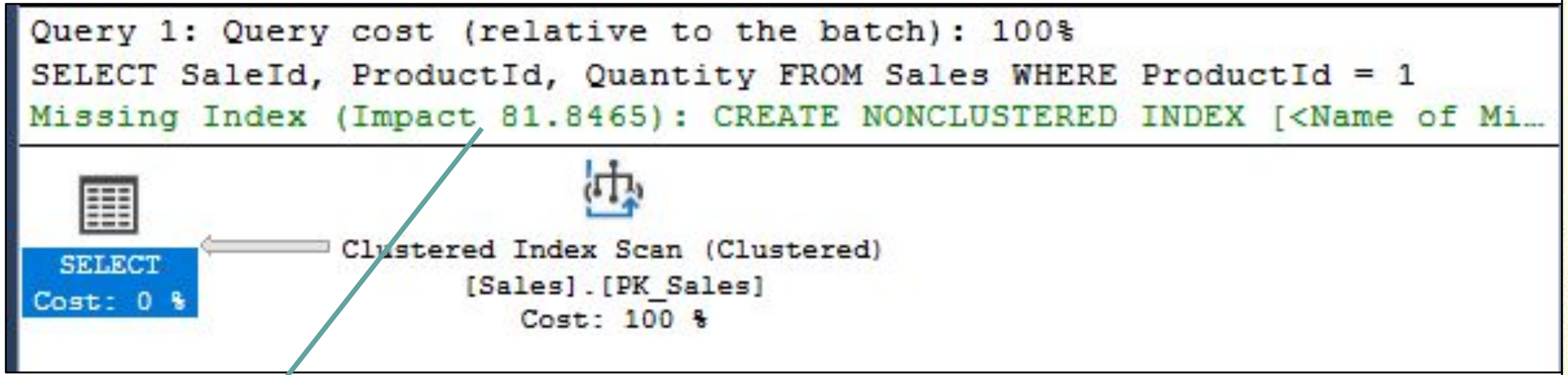
Seek Predicates

Seek Keys[1]: Prefix: [F:\!BDB\2018_19\DEVELOPMENT\EN\WORK\11PERFORMANCE\PROJECTS\APPPETRENKO\APPPETRENKO\BREADPETRENKO.MDF].[dbo].[Sales].ProductId = Scalar Operator(CONVERT_IMPLICIT(int,[@1],0))

Example

4. Add new Quantity field in SELECT

```sql
SELECT SaleId, ProductId, Quantity
FROM Sales
WHERE ProductId = 1;
```



Right click – Missing Index Details

Clustered Index Scan (Clustered)

Scanning a clustered index, entirely or only a range.

Physical Operation	Clustered Index Scan
Logical Operation	Clustered Index Scan
Estimated Execution Mode	Row
Storage	RowStore
Estimated I/O Cost	0.0653472
Estimated Operator Cost	0.0943825 (100%)
Estimated CPU Cost	0.0290353
Estimated Subtree Cost	0.0943825
Estimated Number of Executions	1
Estimated Number of Rows	8750
Estimated Number of Rows to be Read	26253
Estimated Row Size	17 B
Ordered	False
Node ID	0

Predicate

[F:\!BDB\2018_19\DEVELOPMENT\EN\WORK\11PERFORMANCE\PROJECTS\APPPETRENKO\APPPETRENKO\BREADPETRENKO.MDF].[dbo].[Sales].[ProductId]=(1)

Object

[F:\!BDB\2018_19\DEVELOPMENT\EN\WORK\11PERFORMANCE\PROJECTS\APPPETRENKO\APPPETRENKO\BREADPETRENKO.MDF].[dbo].[Sales].[PK_Sales]

Output List

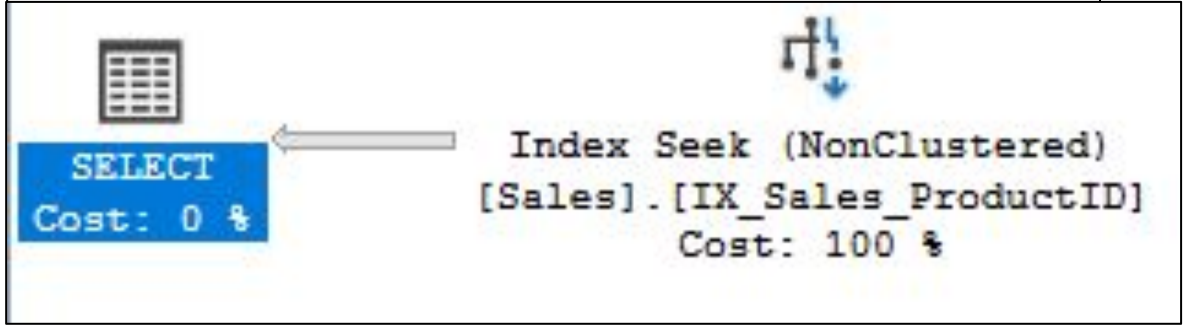
[F:\!BDB\2018_19\DEVELOPMENT\EN\WORK\11PERFORMANCE\PROJECTS\APPPETRENKO\APPPETRENKO\BREADPETRENKO.MDF].[dbo].[Sales].SaleId, [F:\!BDB\2018_19\DEVELOPMENT\EN\WORK\11PERFORMANCE\PROJECTS\APPPETRENKO\APPPETRENKO\BREADPETRENKO.MDF].[dbo].[Sales].ProductId, [F:\!BDB\2018_19\DEVELOPMENT\EN\WORK\11PERFORMANCE\PROJECTS\APPPETRENKO\APPPETRENKO\BREADPETRENKO.MDF].[dbo].[Sales].Quantity

Example

5. Include columns

```
DROP INDEX IX_Sales_ProductID
ON Sales(ProductID);
```

```
CREATE INDEX IX_Sales_ProductID_Inc
ON Sales (ProductId)
INCLUDE (SaleId,Quantity)
```




```
SELECT SaleId, ProductId, Quantity
FROM Sales
WHERE ProductId = 1;
```

Index Seek (NonClustered)

Scan a particular range of rows from a nonclustered index.

Physical Operation	Index Seek
Logical Operation	Index Seek
Estimated Execution Mode	Row
Storage	RowStore
Estimated Operator Cost	0.0250043 (100%)
Estimated I/O Cost	0.0152223
Estimated Subtree Cost	0.0250043
Estimated CPU Cost	0.009782
Estimated Number of Executions	1
Estimated Number of Rows	8750
Estimated Number of Rows to be Read	8750
Estimated Row Size	17 B
Ordered	True
Node ID	0

Object

[F:\!BDB\2018_19\DEVELOPMENT\EN\WORK\11PERFORMANCE\PROJECTS\APPPETRENKO\APPPETRENKO\BREADPETRENKO.MDF].[dbo].[Sales].[IX_Sales_ProductID_Inc]

Output List

[F:\!BDB\2018_19\DEVELOPMENT\EN\WORK\11PERFORMANCE\PROJECTS\APPPETRENKO\APPPETRENKO\BREADPETRENKO.MDF].[dbo].[Sales].SaleId, [F:\!BDB\2018_19\DEVELOPMENT\EN\WORK\11PERFORMANCE\PROJECTS\APPPETRENKO\APPPETRENKO\BREADPETRENKO.MDF].[dbo].[Sales].ProductId, [F:\!BDB\2018_19\DEVELOPMENT\EN\WORK\11PERFORMANCE\PROJECTS\APPPETRENKO\APPPETRENKO\BREADPETRENKO.MDF].[dbo].[Sales].Quantity

Seek Predicates

Seek Keys[1]: Prefix: [F:\!BDB\2018_19\DEVELOPMENT\EN\WORK\11PERFORMANCE\PROJECTS\APPPETRENKO\APPPETRENKO\BREADPETRENKO.MDF].[dbo].[Sales].ProductId = Scalar Operator(CONVERT_IMPLICIT(int,[@1],0))

Example (extension)

6. Add column and condition

```
SELECT SaleId, Sale_date, ProductId, Quantity
FROM Sales
WHERE ProductId = 1 AND Sale_date='01.01.2018';
```

```
CREATE INDEX IX_Sales_Sale_date
ON Sales (Sale_date)
```

Clustered Index Scan (Clustered)

Scanning a clustered index, entirely or only a range.

Physical Operation	Clustered Index Scan
Logical Operation	Clustered Index Scan
Estimated Execution Mode	Row
Storage	RowStore
Estimated I/O Cost	0.0653472
Estimated Operator Cost	0.0943825 (100%)
Estimated CPU Cost	0.0290353
Estimated Subtree Cost	0.0943825
Estimated Number of Executions	1
Estimated Number of Rows	3.4576
Estimated Number of Rows to be Read	26253
Estimated Row Size	20 B
Ordered	False
Node ID	0

Predicate

[F:\!BDB\2018_19\DEVELOPMENT\EN\WORK\11PERFORMANCE\PROJECTS\APPPETRENKO\APPPETRENKO\BREADPETRENKO.MDF].[dbo].[Sales].[ProductId]=(1) AND [F:\!BDB\2018_19\DEVELOPMENT\EN\WORK\11PERFORMANCE\PROJECTS\APPPETRENKO\APPPETRENKO\BREADPETRENKO.MDF].[dbo].[Sales].[Sale_date]='2018-01-01'

Object

[F:\!BDB\2018_19\DEVELOPMENT\EN\WORK\11PERFORMANCE\PROJECTS\APPPETRENKO\APPPETRENKO\BREADPETRENKO.MDF].[dbo].[Sales].[PK_Sales]

Output List

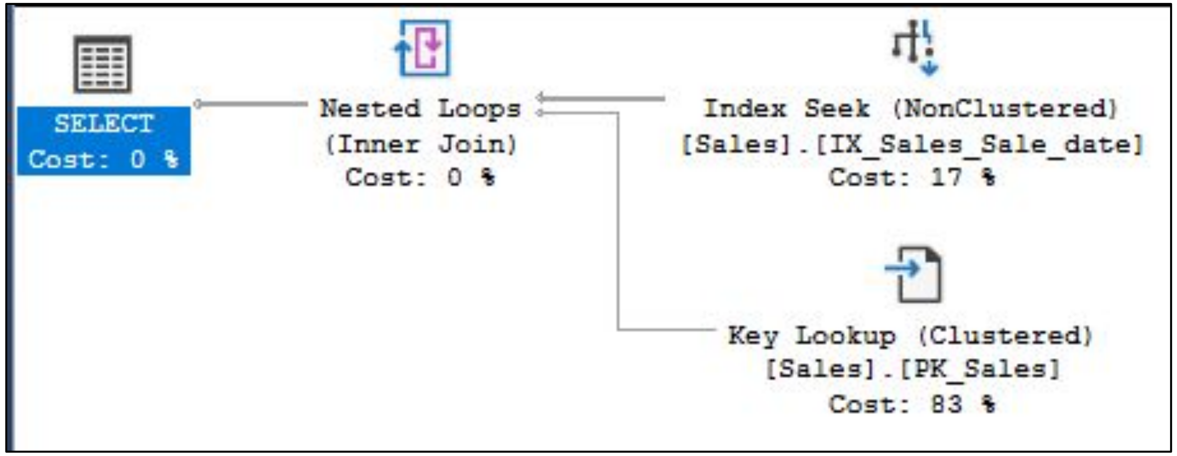
[F:\!BDB\2018_19\DEVELOPMENT\EN\WORK\11PERFORMANCE\PROJECTS\APPPETRENKO\APPPETRENKO\BREADPETRENKO.MDF].[dbo].[Sales].SaleId, [F:\!BDB\2018_19\DEVELOPMENT\EN\WORK\11PERFORMANCE\PROJECTS\APPPETRENKO\APPPETRENKO\BREADPETRENKO.MDF].[dbo].[Sales].Sale_date, [F:\!BDB\2018_19\DEVELOPMENT\EN\WORK\11PERFORMANCE\PROJECTS\APPPETRENKO\APPPETRENKO\BREADPETRENKO.MDF].[dbo].[Sales].ProductId, [F:\!BDB\2018_19\DEVELOPMENT\EN\WORK\11PERFORMANCE\PROJECTS\APPPETRENKO\APPPETRENKO\BREADPETRENKO.MDF].[dbo].[Sales]...

Example (extension)

6. Add column and condition

```sql
CREATE INDEX IX_Sales_Sale_date
ON Sales (Sale_date)
```

```sql
SELECT SaleId, Sale_date, ProductId, Quantity
FROM Sales
WHERE ProductId = 1 AND Sale_date='01.01.2018';
```



Key Lookup (Clustered)

Uses a supplied clustering key to lookup on a table that has a clustered index.

Physical Operation	Key Lookup
Logical Operation	Key Lookup
Estimated Execution Mode	Row
Storage	RowStore
Estimated I/O Cost	0.003125
Estimated Operator Cost	0.0161762 (83%)
Estimated CPU Cost	0.0001581
Estimated Subtree Cost	0.0161762
Estimated Number of Executions	5.98908
Estimated Number of Rows	3.4576
Estimated Row Size	13 B
Ordered	True
Node ID	3

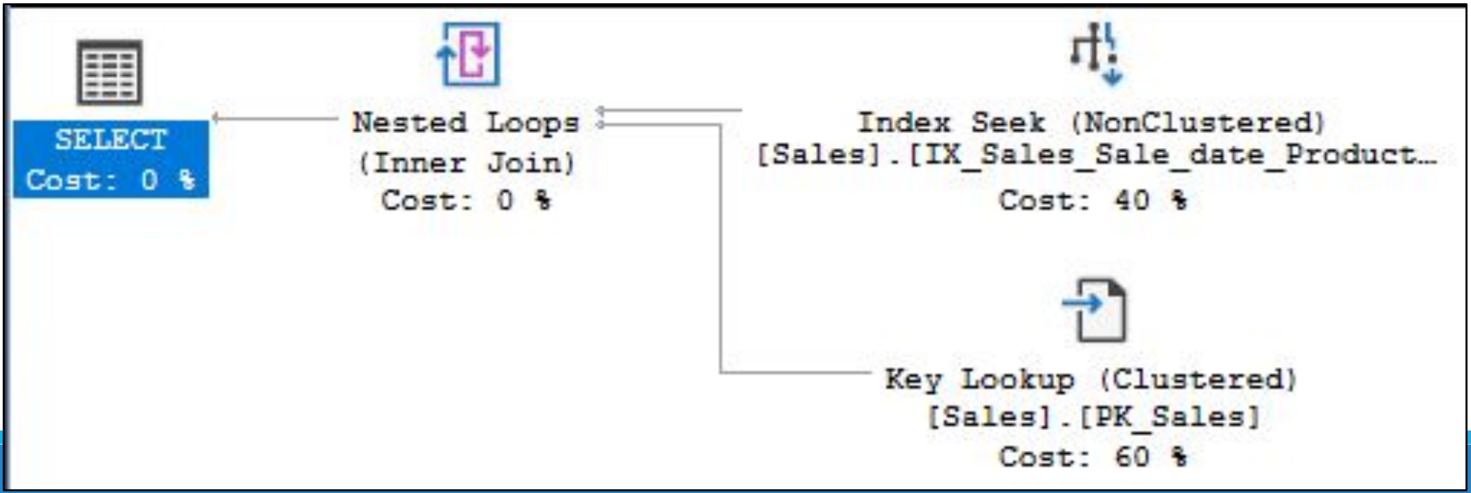
Example (extension- from tips)

6. Add column and condition

```
DROP INDEX IX_Sales_Sale_date
ON Sales;
```

```
CREATE INDEX IX_Sales_Sale_date_ProductId
ON Sales ([Sale_date],[ProductId])
```

```
SELECT SaleId, Sale_date, ProductId, Quantity
FROM Sales
WHERE ProductId = 1 AND Sale_date='01.01.2018';
```



Index Seek (NonClustered)

Scan a particular range of rows from a nonclustered index.

Physical Operation	Index Seek
Logical Operation	Index Seek
Estimated Execution Mode	Row
Storage	RowStore
Estimated Operator Cost	0.0032842 (40%)
Estimated I/O Cost	0.003125
Estimated Subtree Cost	0.0032842
Estimated CPU Cost	0.0001592
Estimated Number of Executions	1
Estimated Number of Rows	1.99658
Estimated Number of Rows to be Read	1.99658
Estimated Row Size	18 B
Ordered	True
Node ID	1

Key Lookup (Clustered)

Uses a supplied clustering key to lookup on a table that has a clustered index.

Physical Operation	Key Lookup
Logical Operation	Key Lookup
Estimated Execution Mode	Row
Storage	RowStore
Estimated Operator Cost	0.004983 (60%)
Estimated I/O Cost	0.003125
Estimated Subtree Cost	0.004983
Estimated CPU Cost	0.0001581
Estimated Number of Executions	1.996578
Estimated Number of Rows	1
Estimated Row Size	9 B
Ordered	True
Node ID	3

Test questions

en:

1. What functions does the query optimizer perform?

2. What is the purpose of the indexes?

3. Compare the **Estimated execution plan** with **Actual execution plan** .

ru:

1. Какие функции выполняет оптимизатор запросов?

2. Каково назначение индексов?

3.Сравните **предполагаемый план выполнения** с **действитель-ным планом выполнения**.