

**Question W1:** Define Primary Key, Foreign Key and Unique Key.

* Primary Key:
* Foreign Key:
* Unique Key:

**Solution W1:**   
A primary key is a column or set of columns in a table that uniquely identifies each row in the table. It cannot contain null values and must be unique across all rows in the table.

A foreign key is a column or set of columns in a table that is used to establish a link between the data in two tables. The foreign key references the primary key of another table, creating a parent-child relationship between the two tables.

A unique key is a column or set of columns that contains unique values, but it can contain null values. It is used to enforce the uniqueness of the values in a specific column or set of columns and prevent duplicate values in the table.

**Question W2:** Define a CTE and how you would use it.

Solution W2: A Common Table Expression (CTE) is a named temporary result set that you can reference within a SELECT, INSERT, UPDATE, or DELETE statement. It is defined using the WITH clause followed by a SELECT statement.

A CTE can be used to simplify complex queries by breaking them down into smaller, more manageable pieces. For example, you can use a CTE to perform a complex calculation or to retrieve a specific subset of data, and then use that result set in the main query.

Here's an example of how to use a CTE:

WITH cte\_name (column1, column2, column3) AS (

SELECT column1, column2, column3

FROM table1

WHERE condition

)

SELECT \*

FROM cte\_name

WHERE condition;

In this example, we first define a CTE named "cte\_name" which is the result of a SELECT statement from table1 where condition is true. The CTE result is used as a table in the main SELECT statement.

Note that CTEs are only visible to the query in which they are defined and they are not stored physically like a table.

**Question W3:** How can you enforce use of a particular index?

**Solution W3:**   
There are a few ways to enforce the use of a particular index in a SQL query.

1. Use the FORCE INDEX hint: You can use the FORCE INDEX hint in the query to tell the database to use a specific index. The syntax for the FORCE INDEX hint is as follows:   
     
   SELECT column1, column2

FROM table1

WHERE column3 = 'value'

FORCE INDEX (index\_name)

1. Use the USE INDEX hint: Similar to the INDEX hint, the USE INDEX hint tells the database to use a specific index. The syntax for the USE INDEX hint is as follows:

SELECT column1, column2

FROM table1

WHERE column3 = 'value'

USE INDEX (index\_name)

1. Use the IGNORE INDEX hint: You can use the IGNORE INDEX hint to tell the database to ignore a specific index. This is useful when you want to ensure that a specific index is not used in a query. The syntax for the IGNORE INDEX hint is as follows:

SELECT column1, column2

FROM table1

WHERE column3 = 'value'

IGNORE INDEX (index\_name)

1. Use the FORCE/USE/IGNORE INDEX hint in the JOIN clause: You can also use the INDEX hints in the JOIN clause to specify which index should be used for the join operation.

SELECT column1, column2

FROM table1

FORCE INDEX (index\_name)

JOIN table2

ON table1.column = table2.column

It's worth noting that using index hints can have an impact on the performance of the query and should be used with caution, as the optimizer is designed to select the best index for a query and using hints can override the optimizer's decision.

**Question W4:** What are the best reasons for using a stored procedures instead of direct T-SQL?

**Solution W4:**   
There are several reasons why you might choose to use a stored procedure instead of direct T-SQL:

1. Reusability: Stored procedures allow you to encapsulate a specific set of operations and reuse them across multiple applications or systems. This can make your code more modular and maintainable.
2. Security: Stored procedures can help to improve security by limiting the types of operations that can be performed on the database. You can use permissions to restrict which users or roles have access to specific stored procedures, and you can use encryption to protect sensitive data.
3. Performance: Stored procedures can improve performance by reducing network traffic and reducing the amount of parsing that needs to be done. When a stored procedure is executed, the query plan is compiled and stored in the cache, so the procedure can be executed more quickly the next time it's called.
4. Abstraction: Stored procedures provide an abstraction layer between the application and the underlying data, which can make it easier to change the underlying data schema without affecting the application.
5. Concurrency control: Stored procedures can be used to control concurrent access to the database, by using transactions and locking mechanisms.
6. Business Logic: Stored procedures can handle complex business logic that can't be handled by a simple query.
7. Centralized management: Stored procedures can be centrally managed, which means that changes to the logic can be made in one place and will be reflected across all applications that use the stored procedure.
8. Portability: Stored procedures can be used across multiple database platforms, meaning that you can write a stored procedure once and use it across different databases.

It's worth noting that these are general advantages, and each particular case should be evaluated considering its specific needs. Additionally, it's important to consider the trade-offs between using stored procedures versus other alternatives, such as dynamic SQL or ORM.

**Question W5:** What are the steps you would take to improve the performance of a poor performing query?

**Solution W5:**

Here are some steps that you can take to improve the performance of a poor performing query:

1. Understand the query: Start by understanding the query, its purpose, and the data it is working with. This will help you identify any potential issues with the query.
2. Check the execution plan: Use the SQL Server Management Studio (SSMS) to view the query execution plan. This will show you how the query is being executed and where the performance bottlenecks are.
3. Check the indexes: Make sure that the appropriate indexes are in place. If indexes are missing, create them. If the indexes are fragmented, rebuild them.
4. Check the statistics: Make sure that the statistics on the tables are up to date. If statistics are outdated, update them.
5. Check the data types: Make sure that the data types used in the query match the data types of the columns in the table.
6. Check the joins: Make sure that the joins are performed on the appropriate columns, and that the join conditions are selective.
7. Check the subqueries: Make sure that the subqueries are not returning more data than necessary.
8. Check the scalar functions: Avoid using scalar functions in the query, as they can significantly impact the performance of the query.
9. Check for blocking: Check for any blocking that might be occurring in the database.
10. Use Temporary Table or Table Variables: If a query is performing several joins or subqueries, try to use temporary tables or table variables to reduce the number of joins or subqueries.
11. Use Stored Procedures: If the query is executed frequently, consider using a stored procedure to improve performance.
12. Test and Monitor: Finally, test the query performance and monitor it to make sure that the changes you made have improved the performance of the query.

It's worth noting that these steps are not necessarily the only steps you should take, and that each case should be evaluated on its own. Additionally, it's important to consider that improving query performance may involve trade-offs and that a solution that improves performance for one query might have a negative impact on other queries or the overall system.

**Question T1:** Basic SELECT statement with Concatenation, Criteria and Ordering

* 1. Return the orders for customers that are married, have an occupation of Professional and were placed in December 2013.
     1. Include fields FullName (First + Last), EnglishOccupation ,EnglishEducation, DateFirstPurchase, SalesOrderNumber, OrderDate, ProductAlternateKey,EnglishProductName, StandardCost, ListPrice, DealerPrice and SalesAmount.
     2. Order by LastName, FirstName, EnglishProductName.

**Solution T1:   
  
Solution Description:**

Step 1: Analyzed all the tables using Database Diagrams and found that three tables such   
 as FactInternetSales, DimProduct and DimCustomer are required for the analysis

Step 2: Joined the tables FactInternetsales and DimProduct using INNER JOIN on ProductKey field and   
 joined the resultant table with DimCustomer on CustomerKey field

Step 3: Filtered the records with customers that are married, have an   
 occupation of Professional and were placed in December 2013

Step 4: Selected the necessary fields FullName (First + Middle + Last),   
 EnglishOccupation ,EnglishEducation, DateFirstPurchase, SalesOrderNumber, /

OrderDate, ProductAlternateKey,EnglishProductName, StandardCost, ListPrice,   
 DealerPrice and SalesAmount from the joined tables

Step 5 : Finally, Ordered the records by LastName, FirstName, EnglishProductName.

**T-SQL Query:**  
  
SELECT

(c.FirstName + ' ' + c.LastName) as FullName,

c.EnglishOccupation,

c.EnglishEducation,

c.DateFirstPurchase,

s.SalesOrderNumber,

s.OrderDate,

p.ProductAlternateKey,

p.EnglishProductName,

p.StandardCost,

p.ListPrice,

p.DealerPrice,

s.SalesAmount

FROM

FactInternetSales s

JOIN DimProduct p ON s.ProductKey = p.ProductKey

JOIN DimCustomer c ON s.CustomerKey = c.CustomerKey

WHERE

c.MaritalStatus = 'M'

AND c.EnglishOccupation = 'Professional'

AND s.OrderDate >= '2013-12-01'

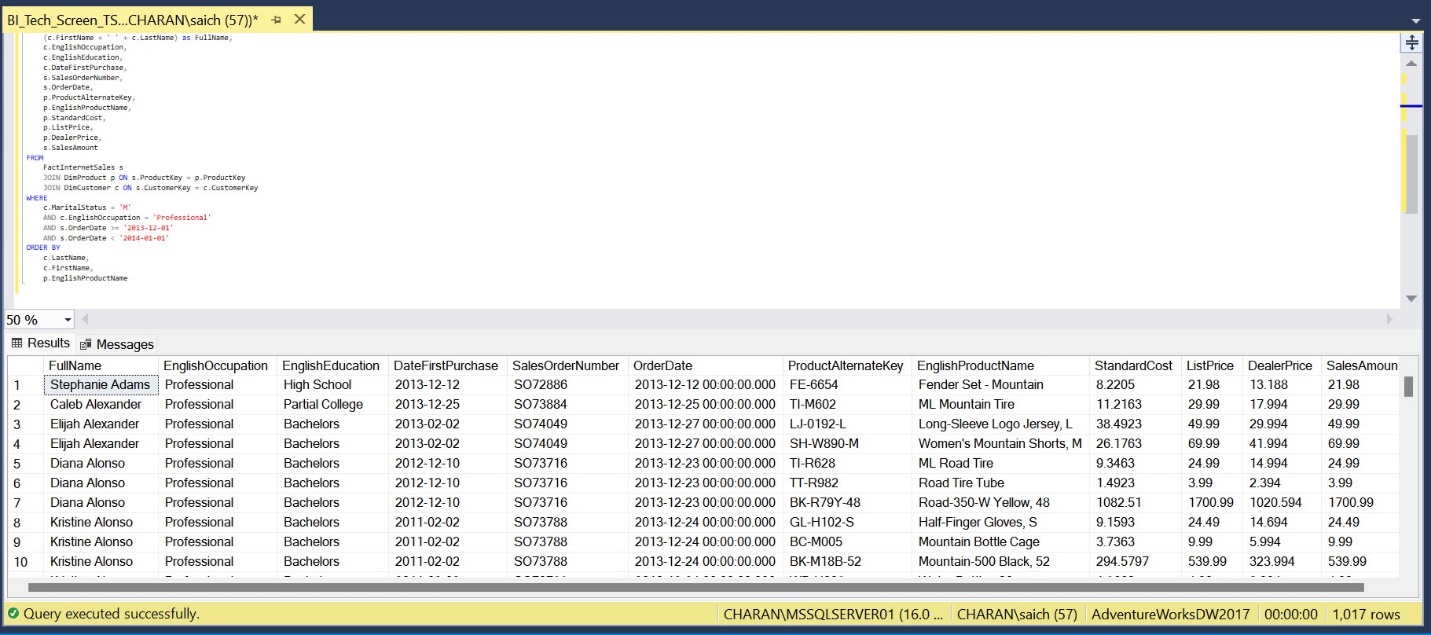
AND s.OrderDate < '2014-01-01'

ORDER BY

c.LastName,

c.FirstName,

p.EnglishProductName

**Screenshot:**   
  


**Question T2:** Basic Aggregation statement with Criteria and Ordering

* 1. Using a copy of the SQL from Question 1, modify to return the count of orders and total sales for customers that are not married, have an occupation of Management and were placed in 2013.
     1. Include fields FullName (First + Last), EnglishOccupation ,EnglishEducation, DateFirstPurchase, OrdersCount, TotalSales.
     2. Order by LastName, FirstName, TotalSales DESC.

**Solution T2:**

**Solution Description:**

Step 1: Analyzed all the tables using Database Diagrams and found that two tables

FactInternetSales and Dim Customer are required for the analysis

Step 2: Joined the tables Fact Internet Sales and DIM Customer using INNER JOIN on   
 CustomerKey field

Step 3: Filtered the resulted table with the customers that are not married,

have an occupation of Management and were placed in 2013.

Step 4:Grouped the fields FirstName, LastName, EnglishOccupation ,

EnglishEducation, DateFirstPurchase to get the aggregated columns

Step 5: Selected the necessary Columns FullName (First + Last), EnglishOccupation ,

EnglishEducation, DateFirstPurchase from the joined tables and used aggregate functions

Count and Sum on the resulted table

Step 6: Ordered the records by LastName, FirstName, TotalSales Desc

**T-SQL Query:**  
  
SELECT

(c.FirstName + ' ' + c.LastName) as FullName,

c.EnglishOccupation,

c.EnglishEducation,

c.DateFirstPurchase,

COUNT(s.SalesOrderNumber) AS OrdersCount,

SUM (s.SalesAmount) AS TotalSales

FROM

FactInternetSales s

JOIN DimCustomer c ON s.CustomerKey = c.CustomerKey

WHERE

c.MaritalStatus != 'M'

AND c.EnglishOccupation = 'MANAGEMENT'

AND s.OrderDate >= '2013-01-01'

AND s.OrderDate < '2014-01-01'

GROUP BY

c.FirstName,

c.LastName,

c.EnglishOccupation,

c.EnglishEducation,

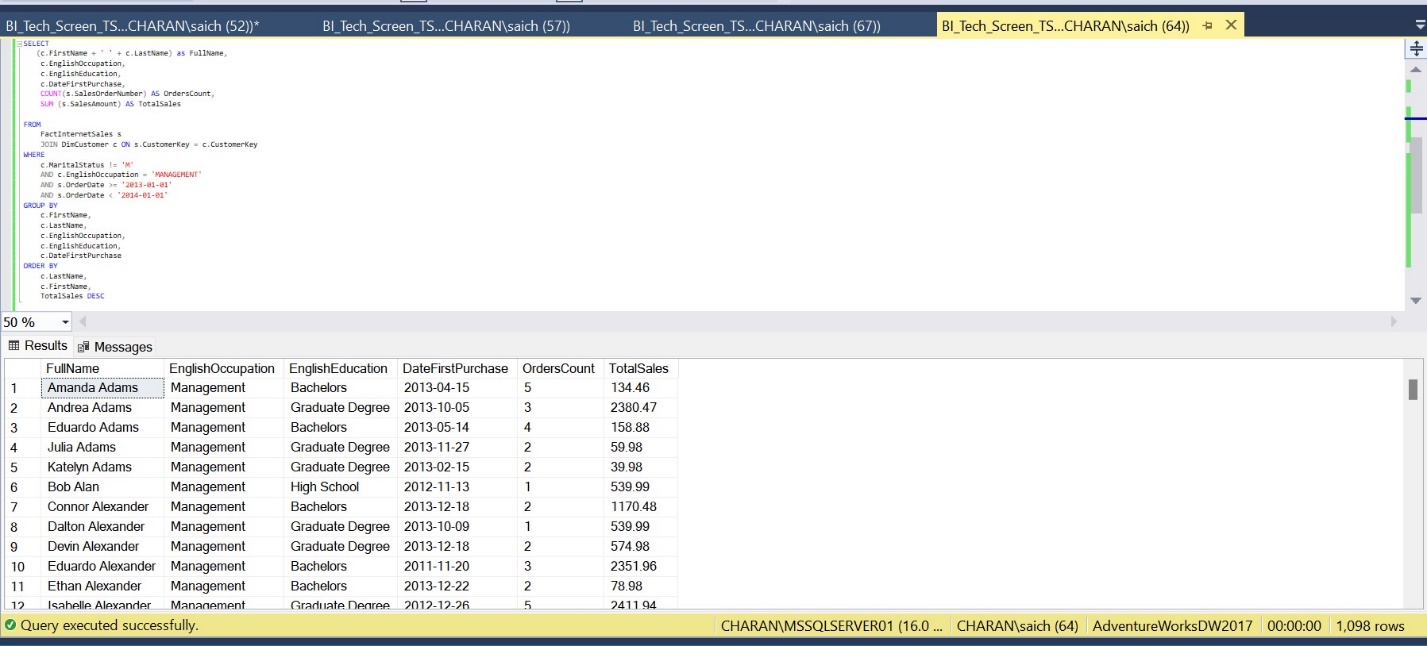
c.DateFirstPurchase

ORDER BY

c.LastName,

c.FirstName,

TotalSales DESC

**Screenshot:**   
  


**Question T3:** SELECT statement with Concatenation, Functions, Criteria and Ordering

* 1. Using a copy of the SQL from Question 1, modify to return the orders, with a new field (see below), for customers with English Occupation of Skilled Manual and were placed in 2012.
     1. Include fields FullName (First + Last), EnglishOccupation ,EnglishEducation, DateFirstPurchase, SalesOrderNumber, OrderDate, ProductAlternateKey, EnglishProductName, StandardCost, ListPrice, DealerPrice and SalesAmount.
     2. Add new field NewSalesOrderKey, after SalesOrderNumber, which is created based on the following:
        1. Modify the Sales Order Number by removing the SO prefix and replace it with the Order Date.
        2. Append the Sales Order Line Number to the end of the string, prefixed with a ".“
        3. Convert the entire value to a float data type.
     3. Order by LastName, FirstName, NewSalesOrderKey.

**Solution T3:**   
  
**Solution Description:**

Step 1: Analyzed all the tables using Database Diagrams and found that three tables   
 such as FactInternetSales, DimProduct and DimCustomer are required for the analysis

Step 2: Joined the tables FactInternetsales and DimProduct using INNER JOIN on ProductKey field and

joined the resultant table with DimCustomer on CustomerKey field

Step 3: Filtered the records with customers that have an

occupation of Skilled Manual and were placed in 2012

Step 4: Selected the necessary Columns FullName (First + Last), EnglishOccupation ,  
 EnglishEducation, DateFirstPurchase, SalesOrderNumber, OrderDate,   
 ProductAlternateKey, EnglishProductName, StandardCost, ListPrice, DealerPrice   
 and SalesAmount from the joined tables

Step 5: Added new field NewSalesOrderKey by removing the SO prefix and replace it with   
 the Order Date and appended the Sales Order Line Number to the end of the string,

prefixed with a "." using Cast and Replace functions and converted the entire value to a float data type.

Step 6: Finally, Ordered the records by LastName, FirstName, NewSalesOrderKey

**T-SQL Query:**  
  
SELECT

(c.FirstName + ' ' + c.LastName) as FullName,

c.EnglishOccupation,

c.EnglishEducation,

c.DateFirstPurchase,

s.SalesOrderNumber,

CAST(REPLACE(s.SalesOrderNumber,'SO',CONVERT(varchar(10),s.OrderDate,112)) +'.'

+CAST(S.SalesOrderLineNumber AS nvarchar) AS float) AS NewSalesOrderKey,

s.OrderDate,

p.ProductAlternateKey,

p.EnglishProductName,

p.StandardCost,

p.ListPrice,

p.DealerPrice,

s.SalesAmount

FROM

FactInternetSales s

JOIN DimProduct p ON s.ProductKey = p.ProductKey

JOIN DimCustomer c ON s.CustomerKey = c.CustomerKey

WHERE

c.EnglishOccupation = 'Skilled Manual'

AND s.OrderDate >= '2012-01-01'

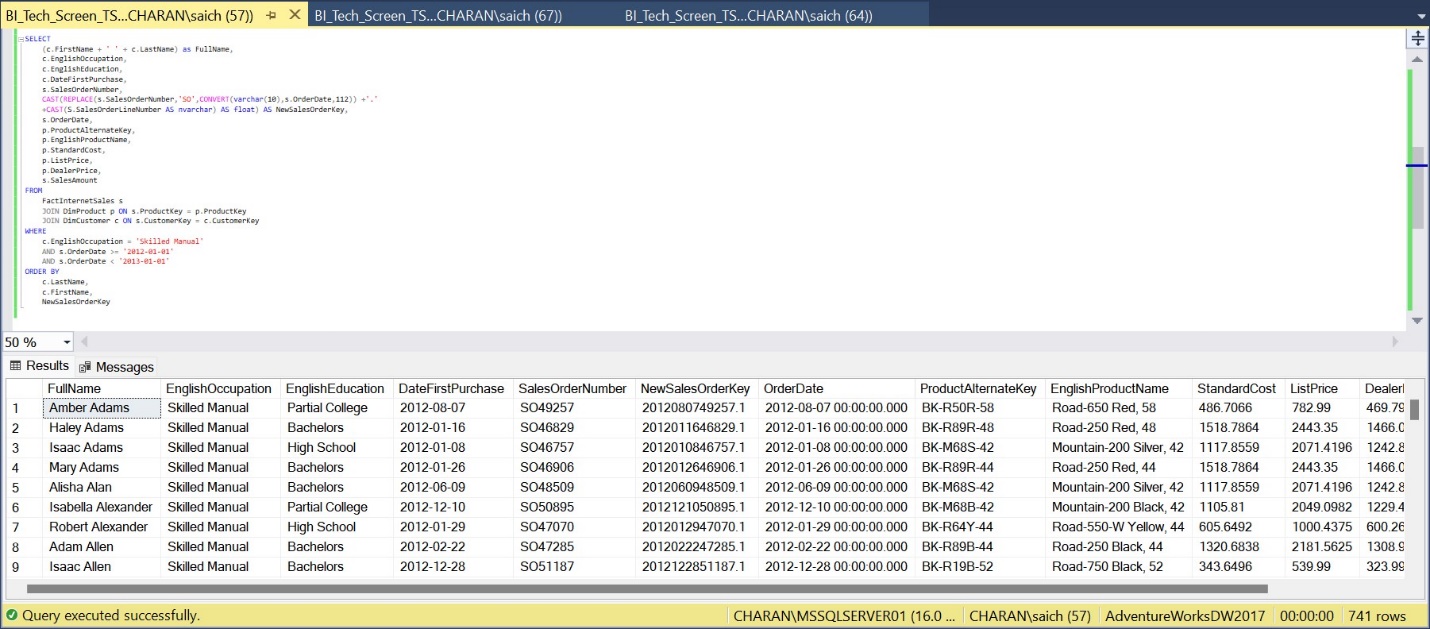
AND s.OrderDate < '2013-01-01'

ORDER BY

c.LastName,

c.FirstName,

NewSalesOrderKey

**Screenshot:**  
  


**Question T4:** Query Analysis and Covering Index

* 1. Based on the SQL created from Question 2, analyze the plans, and add the appropriate covering indexes. This includes the following steps:
     1. Execute existing query and save the pre-update execution plan and messages.
     2. Create/execute needed indexes.
     3. Execute optimized query and save the post-update execution plan and messages.

**Solution T4:**   
  
 Solution Description:

Step 1: Included the execution Plan and ran the T-SQL query created from Question 2.

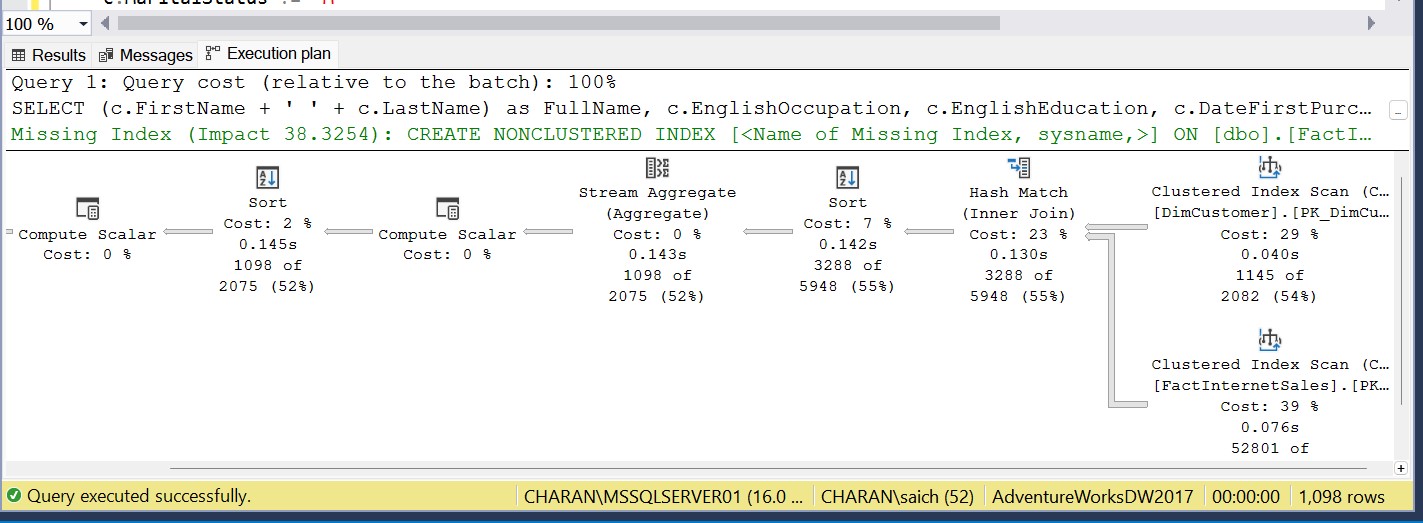
Step 2: Saved and analyzed the execution Plan and messages.

Step 3: Figured out that creating indexes can reduce the server load and optimize the query

Step 3: Created Non-Clustered Indexes on the OrderDate field of

FactInternetSales

Step 4: Ran the optimized Query and saved the execution Plan and messages

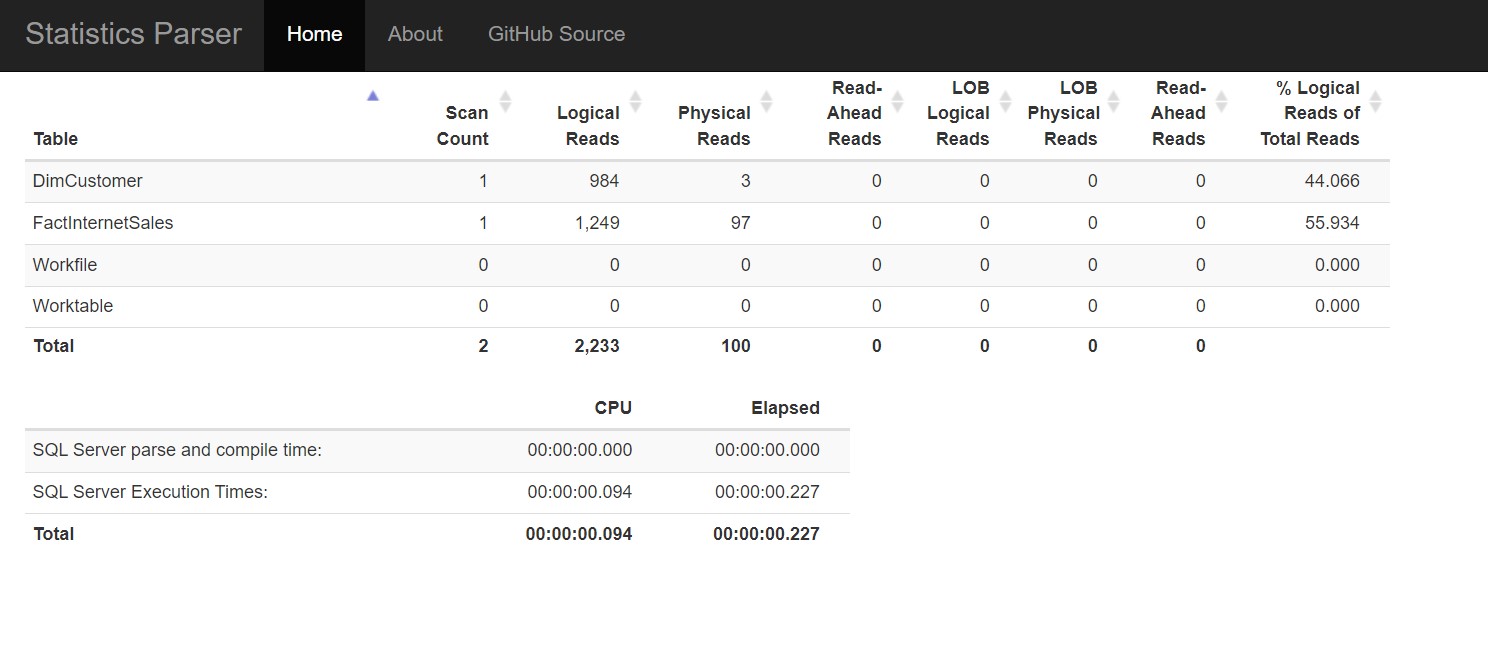
Step 5: Analyzed both the queries and presented the results.   
  
Pre-Execution Plan Screenshot:   
  
  
  
**Pre execution message screenshot**   
  
**Graphical user interface, text, application, email

Description automatically generated**

Sometimes reading SQL Server output from Statistics IO and Statistics Time can be a total drag.   
I have used <https://statisticsparser.com/> help to Parse. output will be formatted and totaled Like below format.   
  
**Pre execution statisticsparser.com screenshot:**

Top of Form

Bottom of Form

  
  
**Optimized SQL Query:**   
  
SET STATISTICS IO ON

SET STATISTICS TIME ON

CREATE NONCLUSTERED INDEX idx\_FactInternetSales\_OrderDate

ON FactInternetSales (OrderDate)

INCLUDE (CustomerKey,SalesAmount)

;

SELECT

(c.FirstName + ' ' + c.LastName) as FullName,

c.EnglishOccupation,

c.EnglishEducation,

c.DateFirstPurchase,

COUNT(s.SalesOrderNumber) AS OrdersCount,

SUM (s.SalesAmount) AS TotalSales

FROM

FactInternetSales s

JOIN DimCustomer c ON s.CustomerKey = c.CustomerKey

WHERE

c.MaritalStatus != 'M'

AND c.EnglishOccupation = 'MANAGEMENT'

AND s.OrderDate >= '2013-01-01'

AND s.OrderDate < '2014-01-01'

GROUP BY

c.FirstName,

c.LastName,

c.EnglishOccupation,

c.EnglishEducation,

c.DateFirstPurchase

ORDER BY

c.LastName,

c.FirstName,

TotalSales DESC

SET STATISTICS TIME OFF

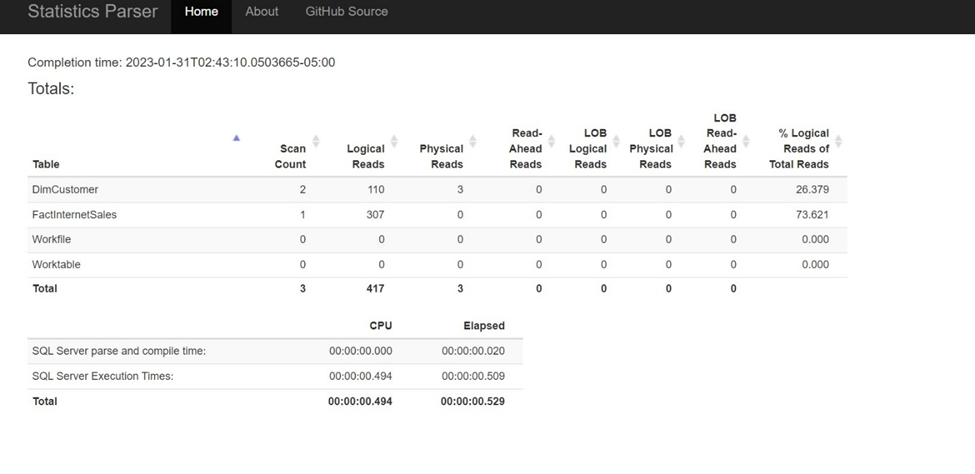
SET STATISTICS IO OFF

**Post update execution plan screenshot:**

**Timeline

Description automatically generated**  
   
**Post update execution message:**   
Graphical user interface, text, application, Word

Description automatically generated

**Pre Update statisticsparser.com screenshot:  
**  
  
**Comparing the pre and post :**

Pre message:

The FactInternetSales table was scanned through index scan once with a logical read of 1249 and a physical read of 97.

The DimCustomer table was scanned through index scan once with a logical read of 984 and a physical read of 3.

CPU time was 94 ms, and elapsed time was 227 ms.

Post message:

The FactInternetSales table was scanned through index seek once with a logical read of 307 and a physical read of 0.

The DimCustomer table was scanned through index seek twice with a total logical read of 110 and a physical read of 3.

CPU time was 46 ms, and elapsed time was 305 ms.

Based on the statistics, the second query performed better than the first one, with lower logical and physical reads and lower CPU time.

**Question T5:** Query Analysis and Optimization

* 1. Analyze the provided query and optimized based on analysis (Hint: Examine criteria and use execution plan for analysis.). This includes the following steps:
     1. Analyze existing query, using built in tools.
     2. Execute existing query and save pre-update execution plan and messages.
     3. Optimize/adjust query, as needed, as well as source tables.

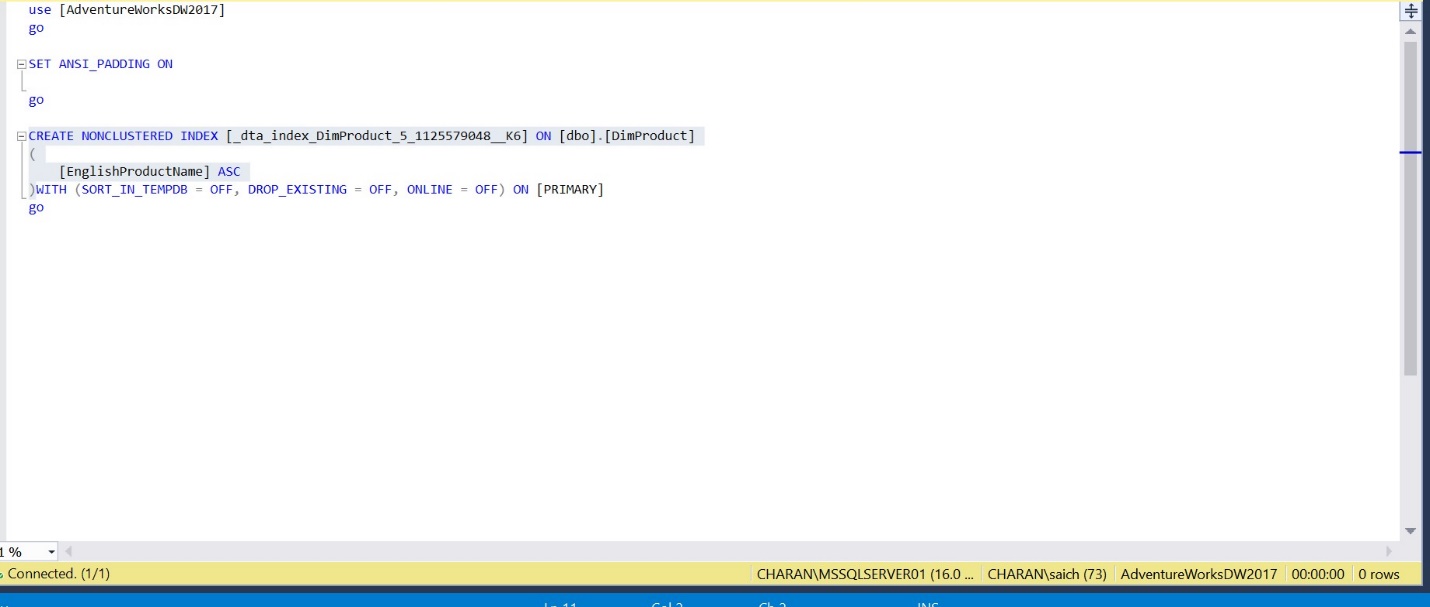
Execute optimized query and save the post-update execution plan and messages.  
  
  
**Solution T5:**   
  
**Solution Description:**

Step1: Analyzed query in database engine tuning advisor and saved the analysis.

Step2: Executed the query and saved the pre update execution plan and messages

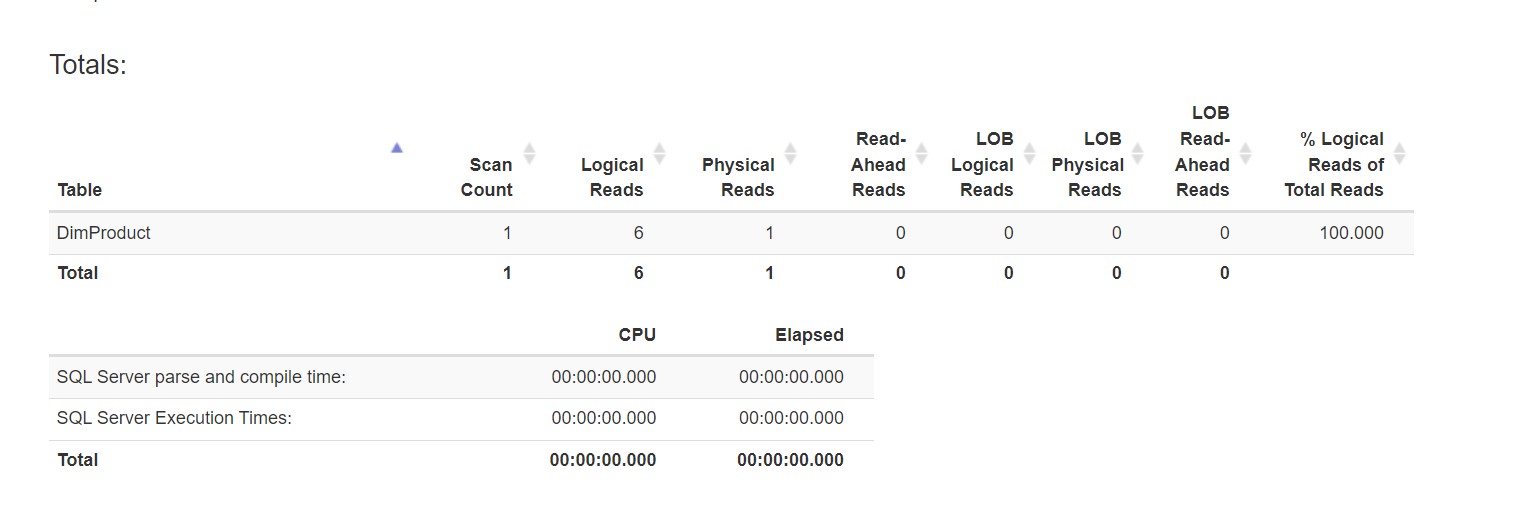
Step3: Followed the given tuning method suggested by tuning advisor and figured that there is no difference between the optimization

Step4: Created non clustered index on Dim Product (EnglishProductName) and removed the function key word Left and updated the query

Step5: Executed optimized query and saved the results.  
  
  
  
**In built tool Database engine tuning advisor screenshot:**  
  
  
Pre-Execution Plan Screenshot:   
Graphical user interface, text, application, email

Description automatically generated

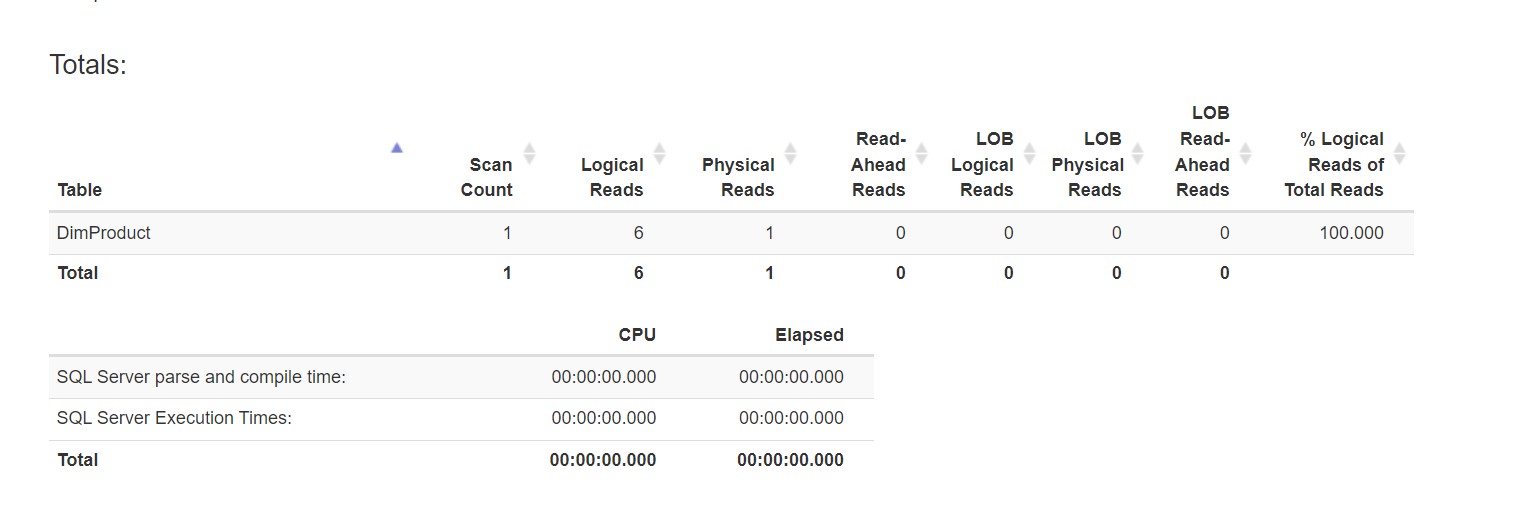
Table

Description automatically generated  
  
  
**Pre execution message screenshot**   
  
****

Sometimes reading SQL Server output from Statistics IO and Statistics Time can be a total drag.   
I have used <https://statisticsparser.com/> help to Parse. output will be formatted and totaled Like below format.   
  
**Pre execution statisticsparser.com screenshot:**

Top of Form

Bottom of Form

  
  
**Optimized SQL Query:**   
  
CREATE NONCLUSTERED INDEX   
[\_dta\_index\_DimProduct\_5\_1125579048\_\_K6] ON [dbo].[DimProduct]

([EnglishProductName])

SET STATISTICS IO ON

SELECT

dp.EnglishProductName

FROM

dbo.DimProduct dp

WHERE

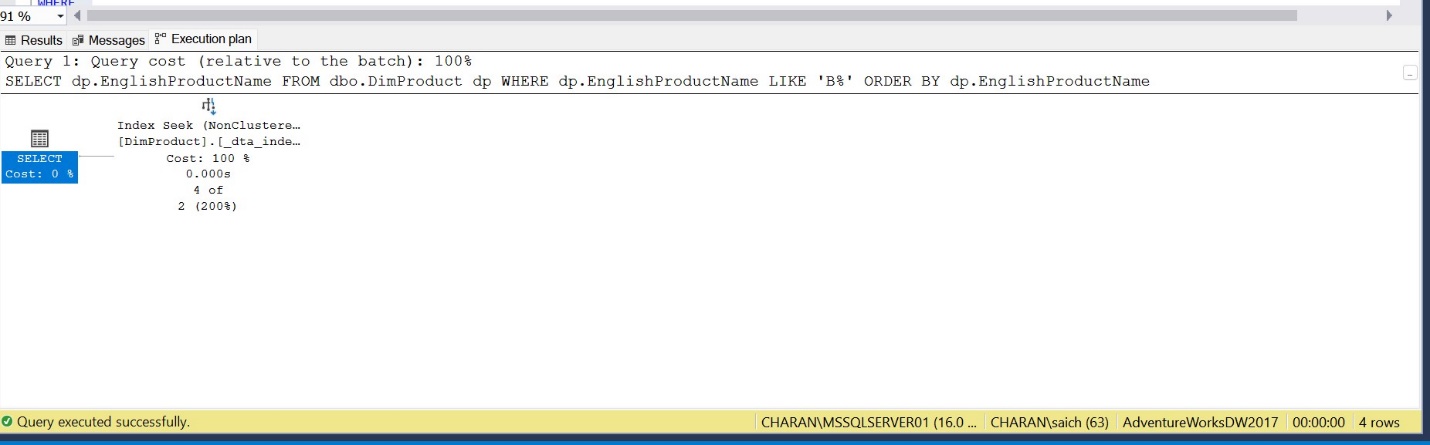
dp.EnglishProductName LIKE 'B%'

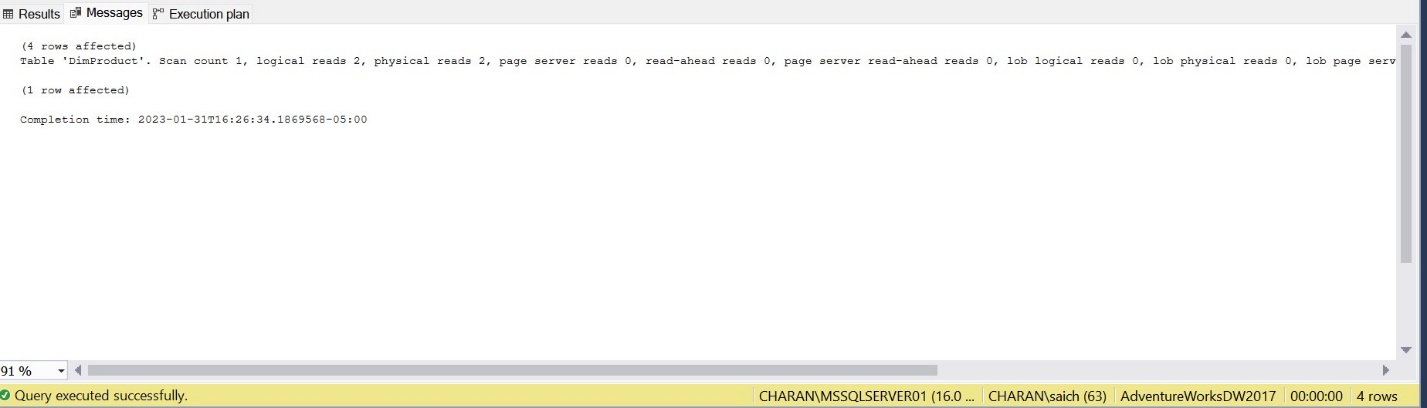
ORDER BY

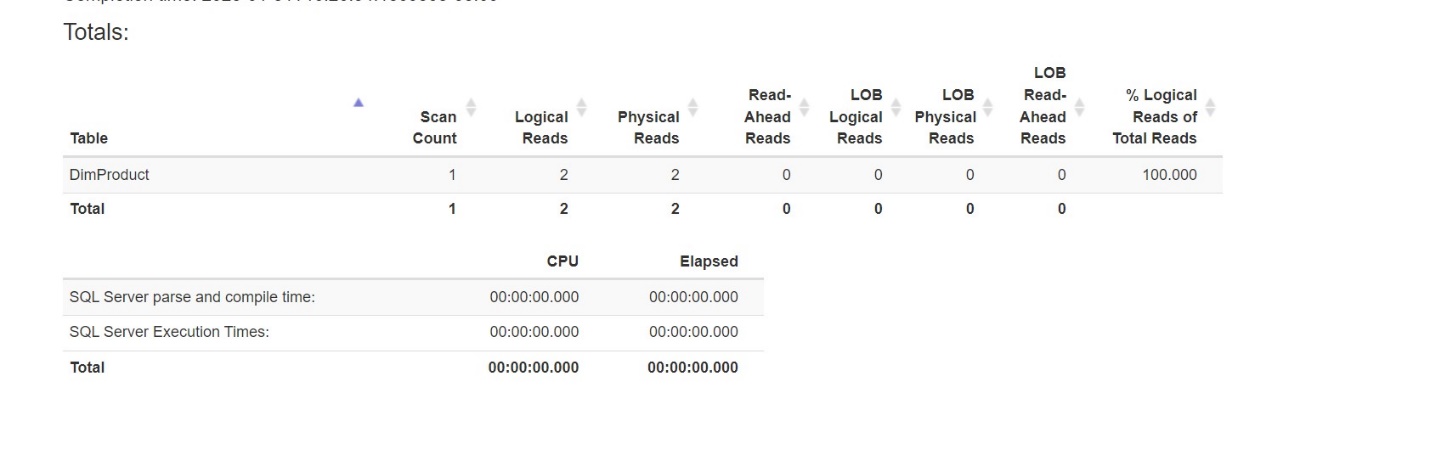
dp.EnglishProductName

SET STATISTICS IO OFF

**Post update execution plan screenshot:**

****Table

Description automatically generated  
   
**Post update execution message:**   


**Pre Update statisticsparser.com screenshot:  
**  
  
**Comparing the pre and post:**

* + 1. The two queries affect 4 rows in the "DimProduct" table. Query 1 took longer to complete compared to Query 2. Query 1 had higher logical reads (6) compared to Query 2 (2). Query 1 had 1 physical read and 11 read-ahead reads, while Query 2 had 2 physical reads and 0 read-ahead reads.  
         
       The two queries differ significantly in terms of the number of actual rows that are read. In the first query, all 606 records in the table are scanned, resulting in a large number of reads. However, when the query was optimized by removing a function and adding a non-clustered index, the number of reads was significantly reduced to only 4 rows.