Use case 1: NOT IN

The aim is to find the userid of users who haven't posted anything on their profile yet.

Note that postUserId is a foreign key in the POSTS relation that refers to the primary key userId of the USERS relation. The number of tuples in POSTS is much lesser than the number of tuples in USERS, hence this query is a valid and logical one to make.

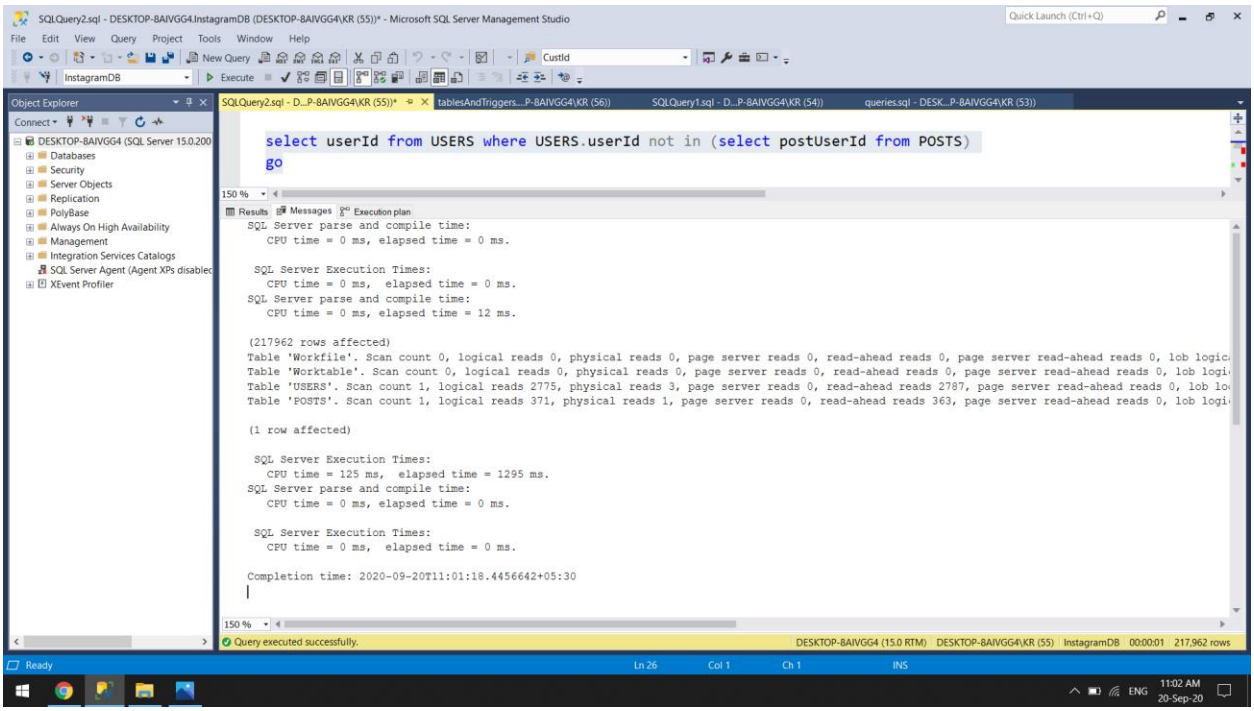
At this point, the indexes are:

- On USERS: index on the userId key which is the primary key of this relation. PK__USERS__CB9A1CFF004D37AB is clustered, unique, primary key located on FG1
- On POSTS: index on postId key which the primary key of this relation. PK__POSTS__DD0C739A88B15966 is clustered, unique, primary key located on FG1

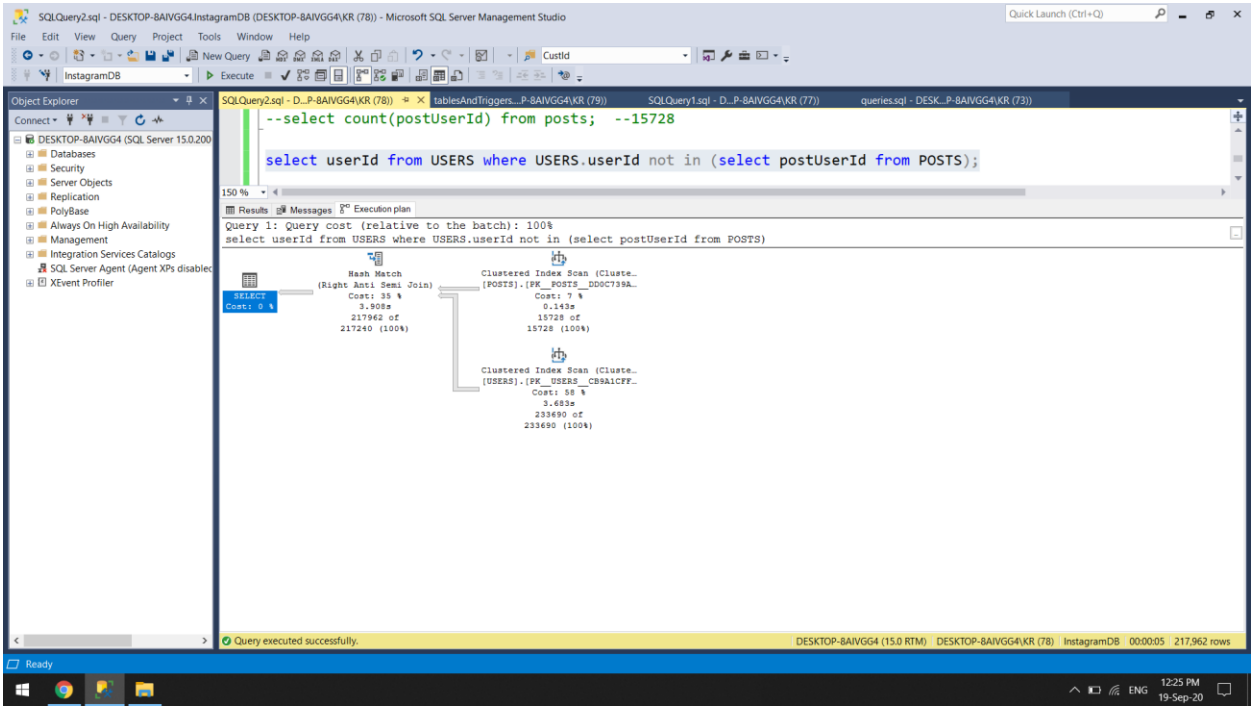
Version 1: select userId from USERS where USERS.userId not in (select postUserId from POSTS)

This query is written using not in. It is perhaps the simplest form of writing the query to get the desired output, however, it takes more time for the query compiler to process this query as compared to the next two versions.

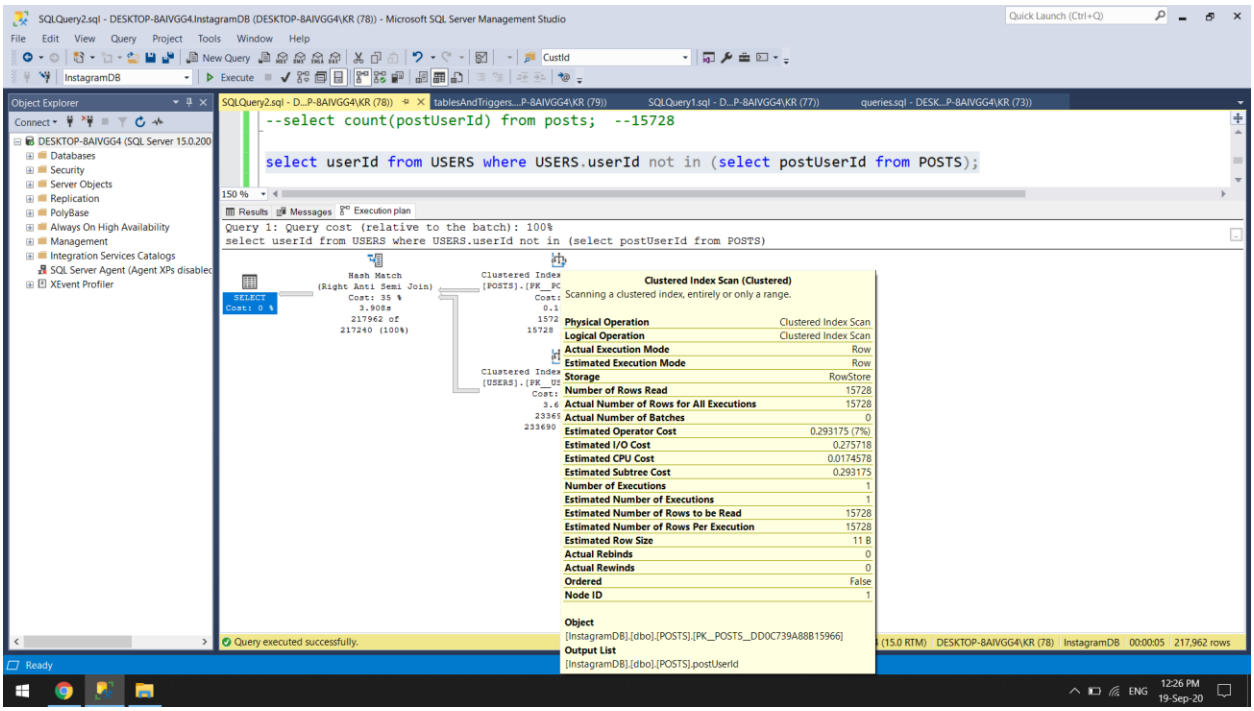
Notice for the USERS table, scan count is 1, number of logical reads is 2775 and number of physical reads is 3. And for the POSTS table, scan count is 1, number of logical reads is 371 and number of physical reads is 1. Also see that the read-aheads in this and each of the following versions is non-zero, because I have cleared the cache buffers before every statement and so the pages will not be fetched from the cache.



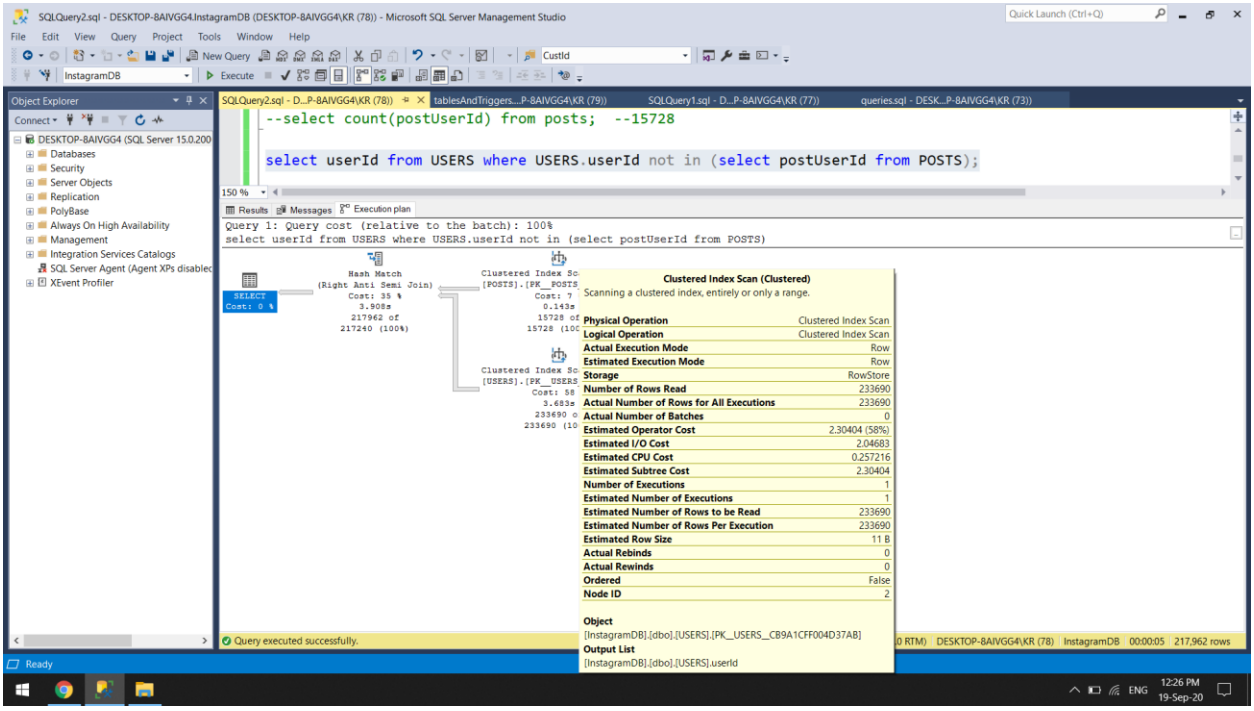
The execution plan is as follows- The clustered index scan on POSTS costs 7% and the clustered index scan on USERS costs 58%.



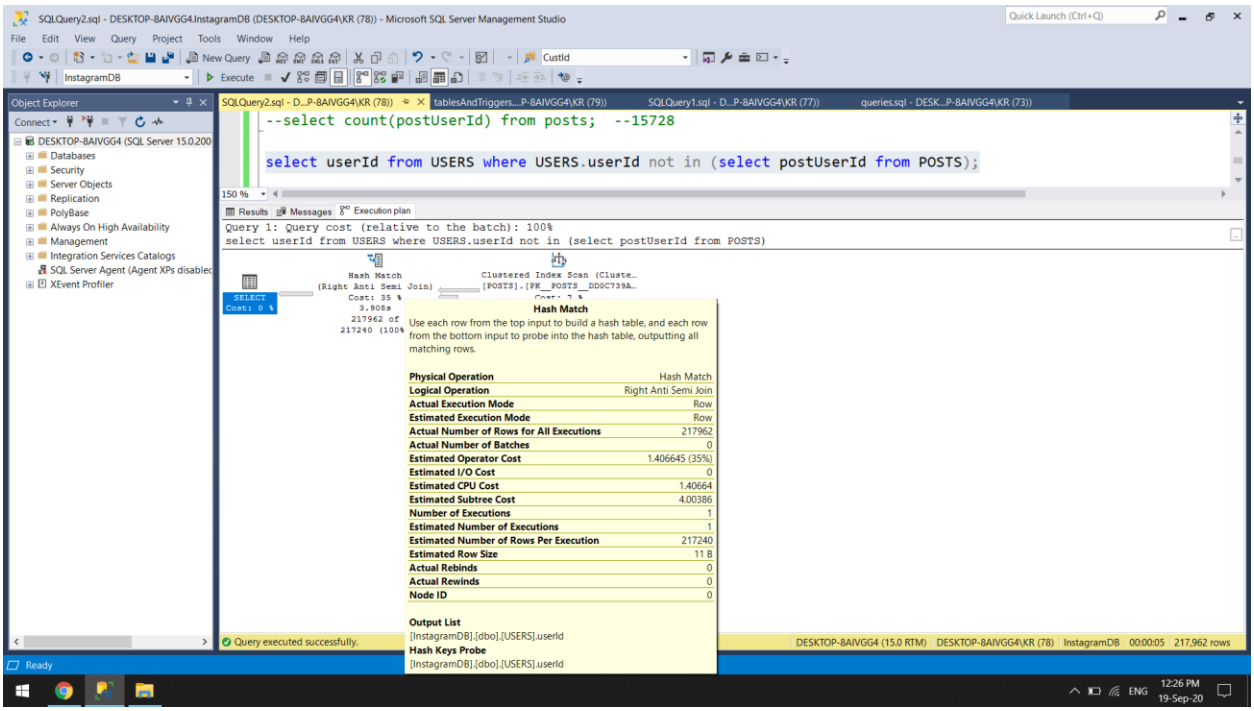
The details of the clustered index scan on POSTS is shown. The output list contains postUserId because that is what is projected in the inner/sub-query.



The details of the clustered index scan on USERS is shown. The output list contains userId because that is what is projected in the outer query.



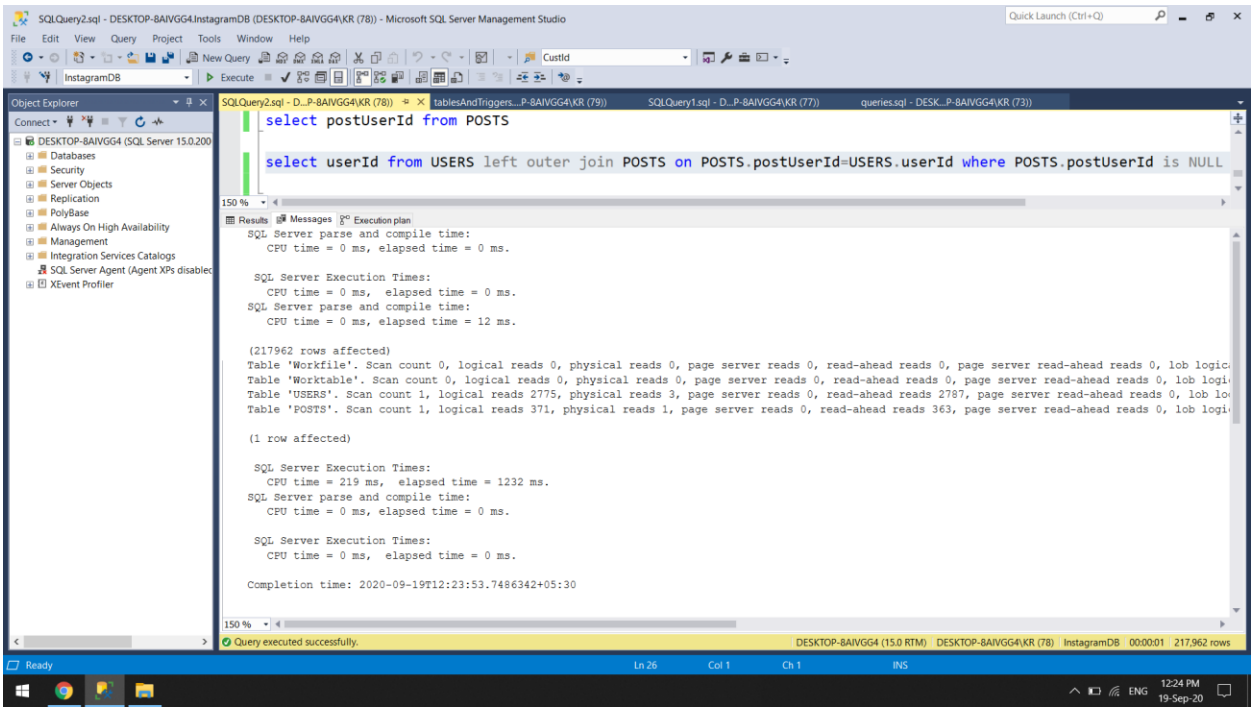
The hash match users each row from POSTS to build a hash table and each row from USERS to probe into the hash table, and then outputs the matching rows. Note that what is projected (output list) is userId from USERS. This makes up about 35% of the cost of the query.



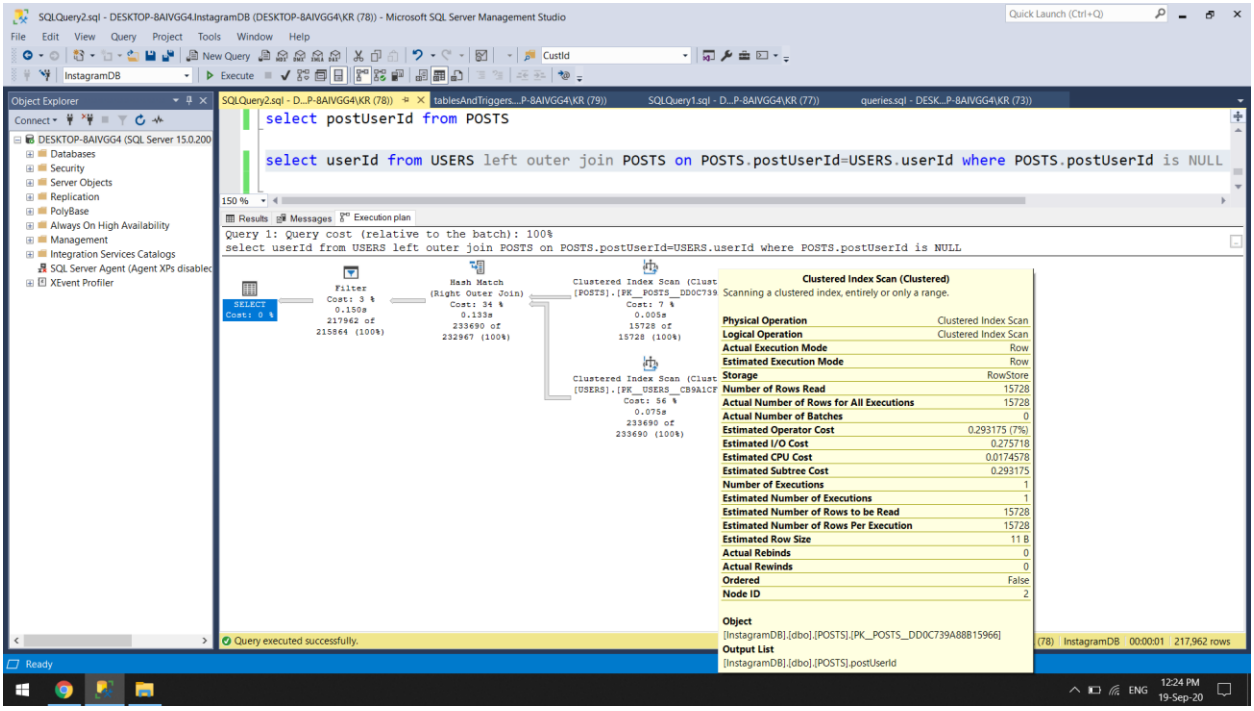
Version 2: select userId from USERS left outer join POSTS on POSTS.postUserId=USERS.userId where POSTS.postUserId is NULL

This query is written using left outer join.

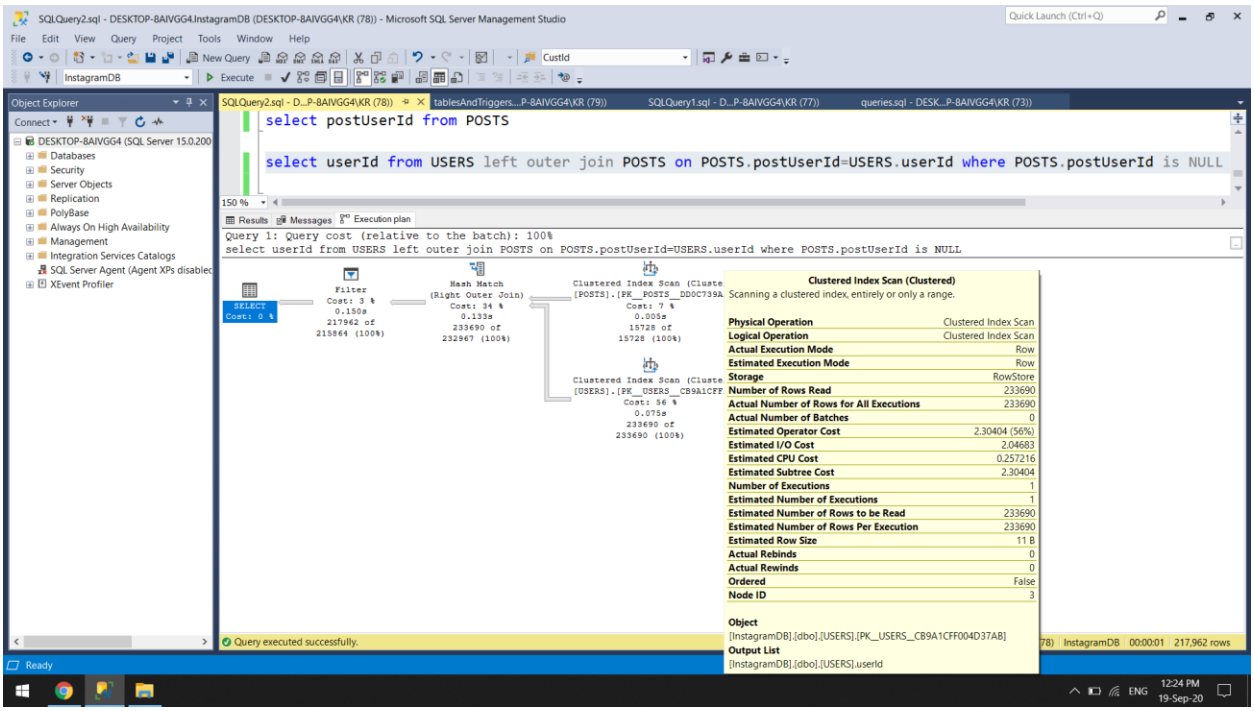
For the USERS table, scan count is 1, number of logical reads is 2775 and number of physical reads is 3. And for the POSTS table, scan count is 1, number of logical reads is 371 and number of physical reads is 1. This is the same as in Version 1. Looking at this, we can say that the query compiler executes in the same way (most optimum way) to achieve the end result, irrespective of how the query is written.



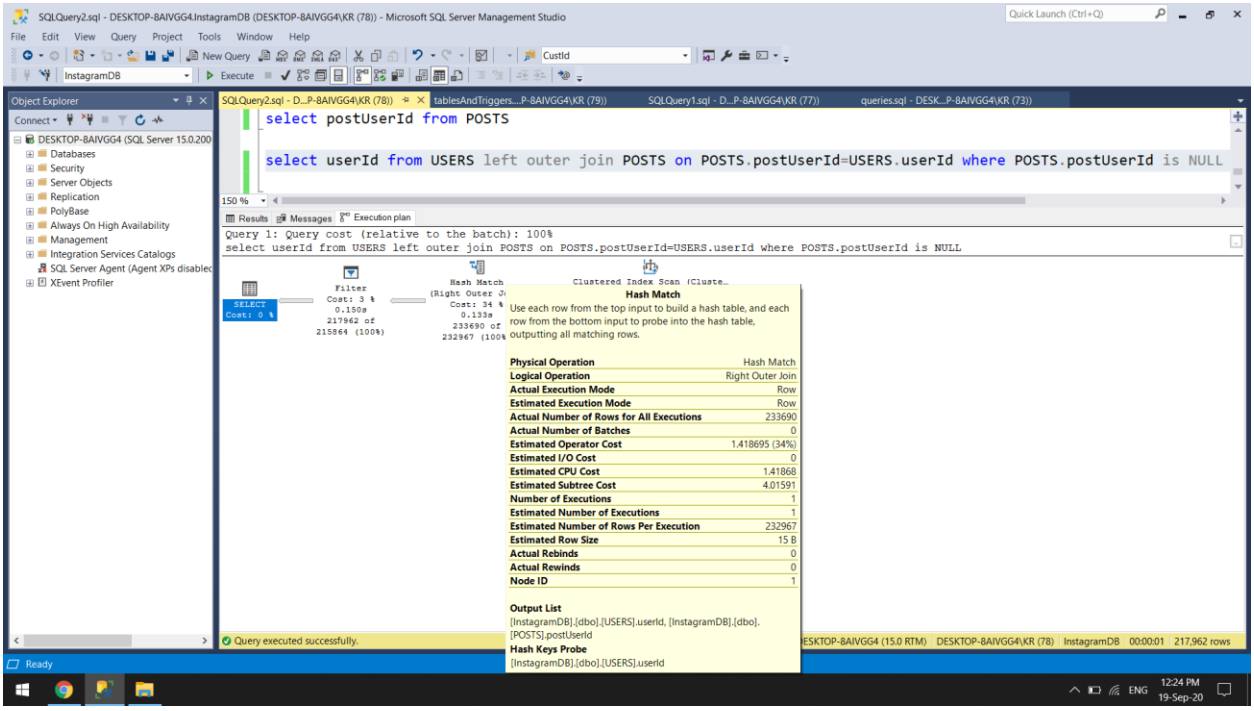
This is the execution plan of the query. Here, the cost of clustered index scan on POSTS is the same as in Version 1.



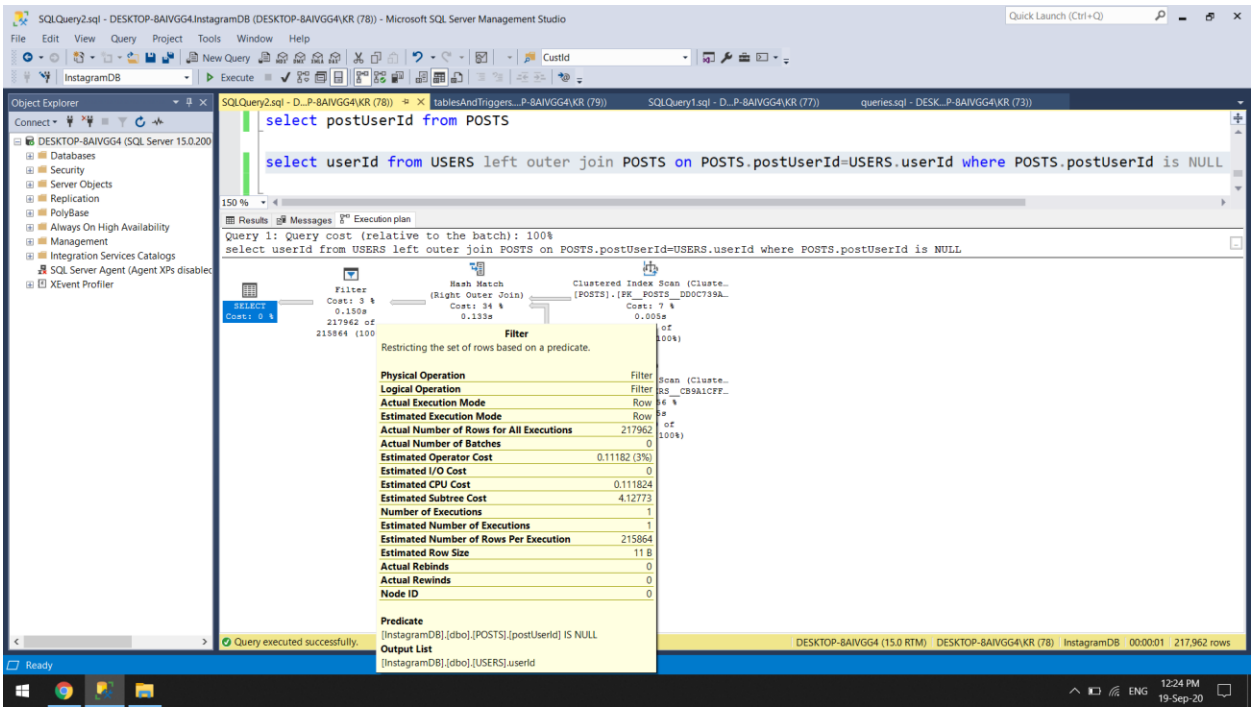
The clustered index scan on USERS has come down to 56% from 58%, but there is an additional filter step after the hash match which wasn't there in Version 1.



Unlike in Version 1 where the output list for hash match was only userId, here the output list has userId from USERS and postUserId from POSTS.



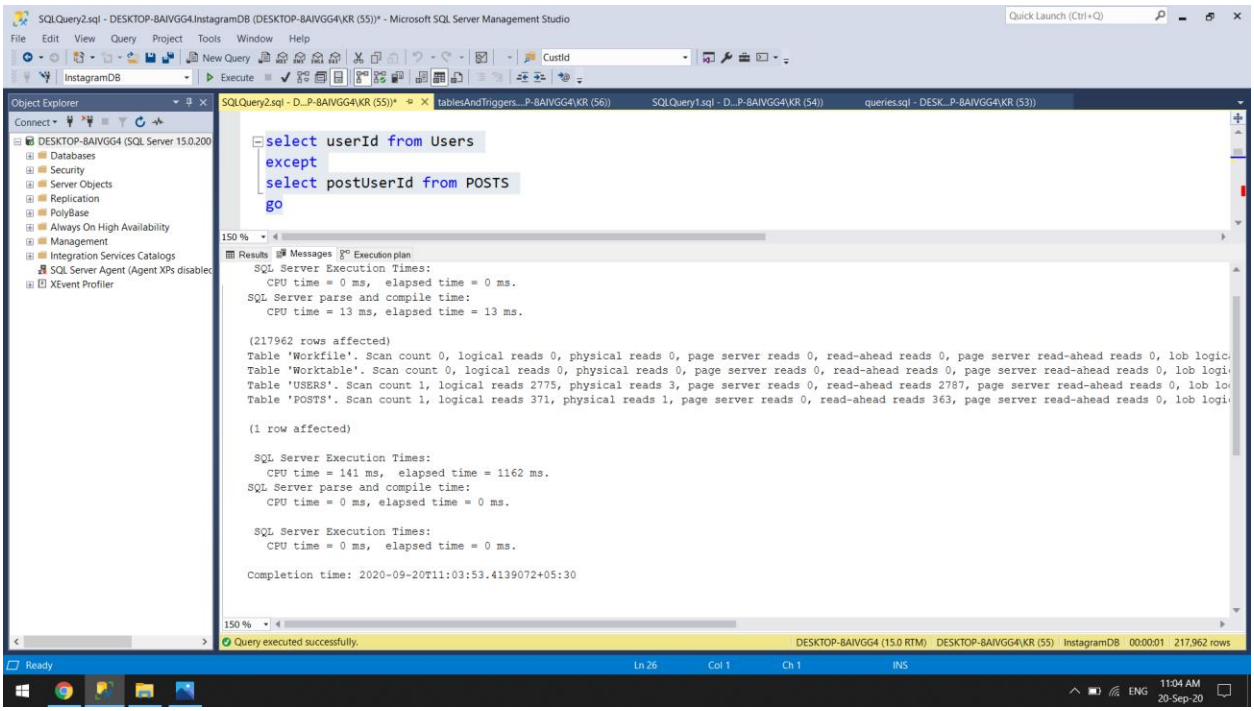
After this step is when the filtering happens for the where condition that we have specified in the query, i.e, in this step, only the values that do have postUserId as NULL as filtered and the userId for those rows make the output.



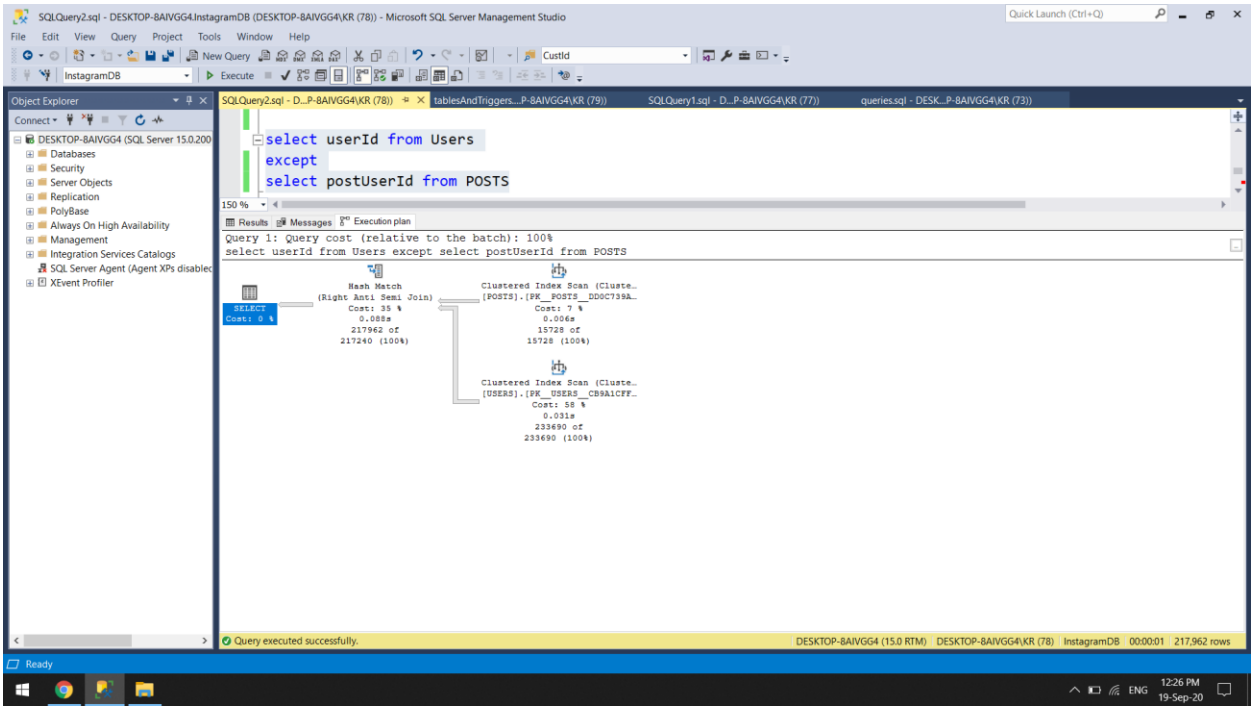
Version 3: select userId from Users except select postUserId from POSTS

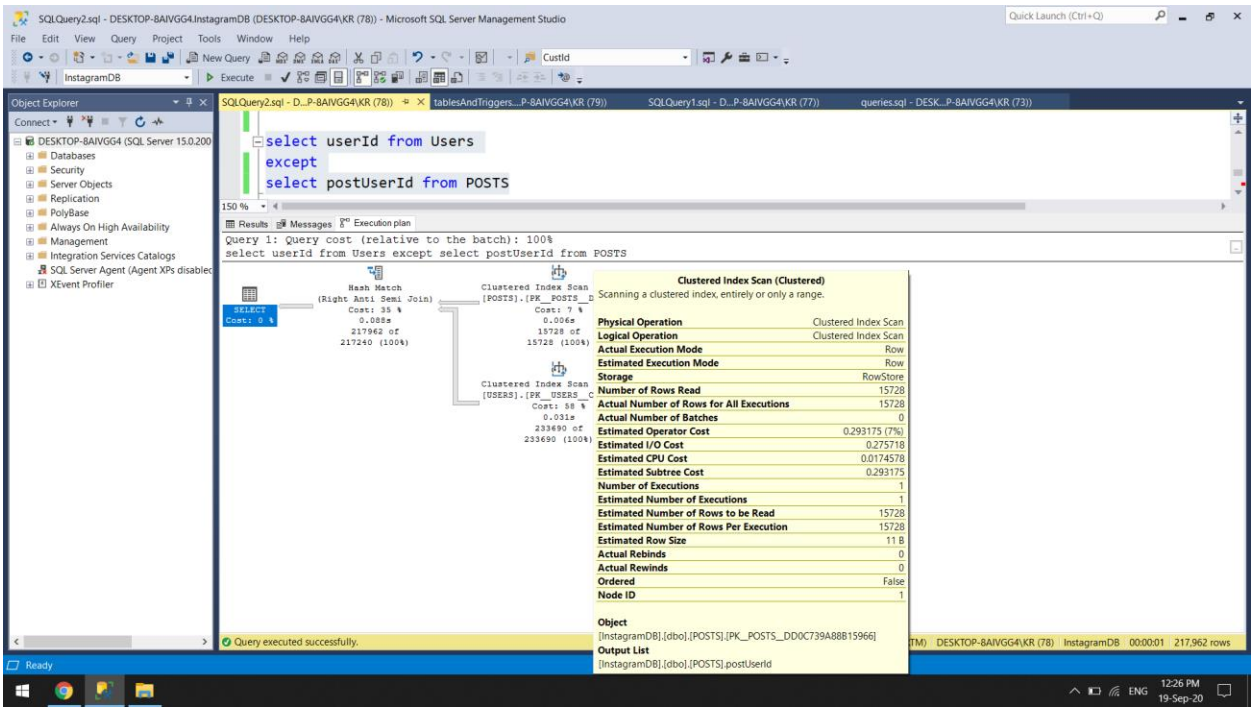
`The query is written using except which is just like MINUS for SQL Server.`

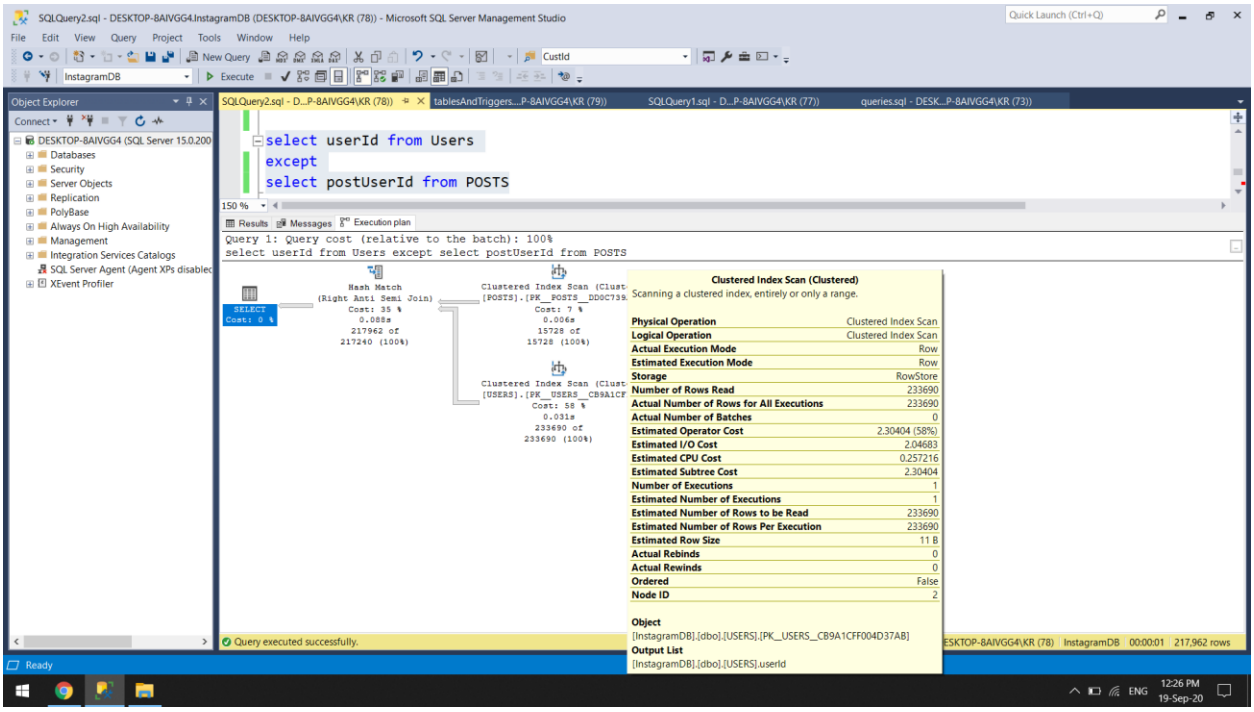
For the USERS table, scan count is 1, number of logical reads is 2775 and number of physical reads is 3. And for the POSTS table, scan count is 1, number of logical reads is 371 and number of physical reads is 1. This is the same as in Versions 1 and 2. Looking at this, we can say that the query compiler executes in the same way (most optimum way) to achieve the end result, irrespective of how the query is written.

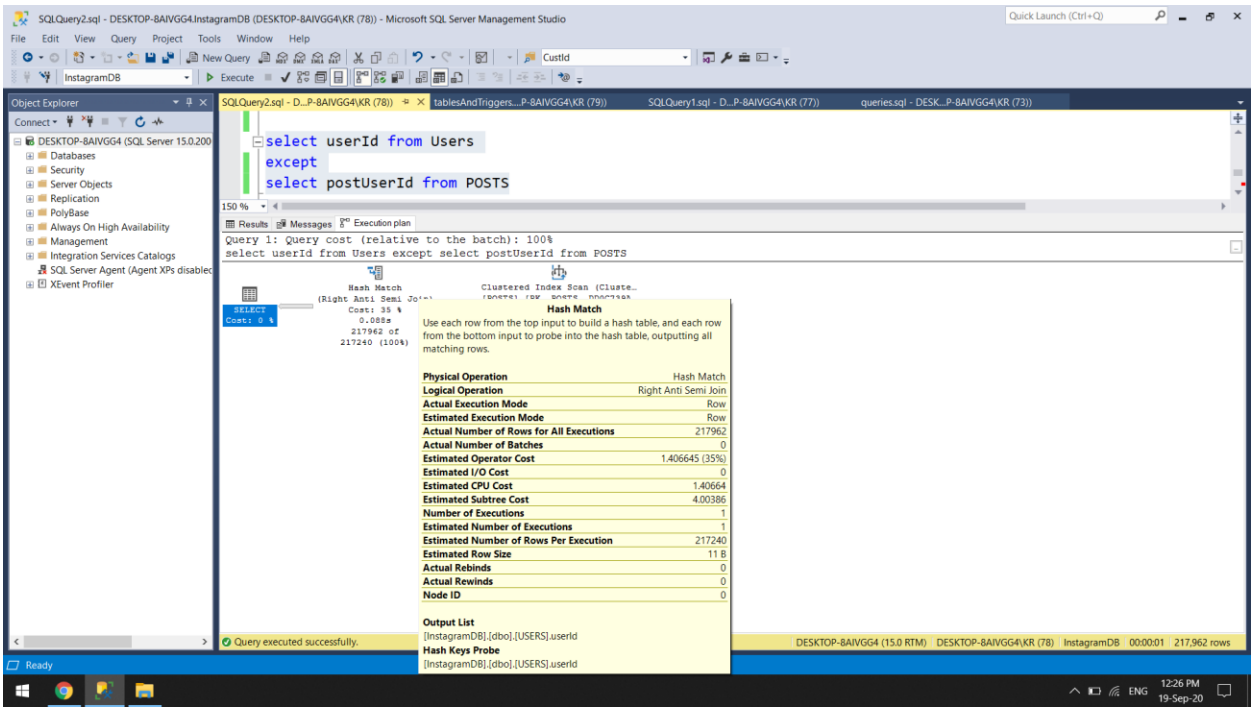


This is the execution plan of the query. It looks exactly like the one in Version1.




SQLQuery2.sql - DESKTOP-8AIVGG4.InstagramDB (DESKTOP-8AIVGG4\KR (78)) - Microsoft SQL Server Management Studio
select userId from Users
except
select postUserId from POSTS
Query 1: Query cost (relative to the batch): 100%
select userId from Users except select postUserId from POSTS
Clustered Index Scan (Clustered)
Scanning a clustered index, entirely or only a range.
Physical Operation Clustered Index Scan
Logical Operation Clustered Index Scan
Actual Execution Mode Row
Estimated Execution Mode Row
Storage RowStore
Number of Rows Read 15728
Actual Number of Rows for All Executions 15728
Actual Number of Batches 0
Estimated Operator Cost 0.293175 (7%)
Estimated I/O Cost 0.275718
Estimated CPU Cost 0.0174578
Estimated Subtree Cost 0.293175
Number of Executions 1
Estimated Number of Executions 1
Estimated Number of Rows to be Read 15728
Estimated Number of Rows Per Execution 15728
Estimated Row Size 11 B
Actual Rebinds 0
Actual Rewinds 0
Ordered False
Node ID 1
Object
[InstagramDB].[dbo].[POSTS].[PK__POSTS__DD0C739A88B15966]
Output List
[InstagramDB].[dbo].[POSTS].postUserId
Query executed successfully.
DESKTOP-8AIVGG4\KR (78) InstagramDB 00:00:01 217,962 rows
Ready


SQLQuery2.sql - DESKTOP-8AIVGG4.InstagramDB (DESKTOP-8AIVGG4\KR (78)) - Microsoft SQL Server Management Studio
select userId from Users
except
select postUserId from POSTS
Query 1: Query cost (relative to the batch): 100%
select userId from Users except select postUserId from POSTS
Clustered Index Scan (Clustered)
Scanning a clustered index, entirely or only a range.
Physical Operation Clustered Index Scan
Logical Operation Clustered Index Scan
Actual Execution Mode Row
Estimated Execution Mode Row
Storage RowStore
Number of Rows Read 233690
Actual Number of Rows for All Executions 233690
Actual Number of Batches 0
Estimated Operator Cost 2.30404 (58%)
Estimated I/O Cost 2.04683
Estimated CPU Cost 0.257216
Estimated Subtree Cost 2.30404
Number of Executions 1
Estimated Number of Executions 1
Estimated Number of Rows to be Read 233690
Estimated Number of Rows Per Execution 233690
Estimated Row Size 11 B
Actual Rebinds 0
Actual Rewinds 0
Ordered False
Node ID 2
Object
[InstagramDB].[dbo].[USERS].[PK__USERS__CB9A1CFF004D37AB]
Output List
[InstagramDB].[dbo].[USERS].userId
Query executed successfully.
DESKTOP-8AIVGG4\KR (78) InstagramDB 00:00:01 217,962 rows
Ready



For the queries of this use case, more than 50% of the cost has been for the clustered index scan on USERS. The clustered index is on the primary key userId, and even our queries are based on userId, so there is no better index that we can create to reduce this cost.

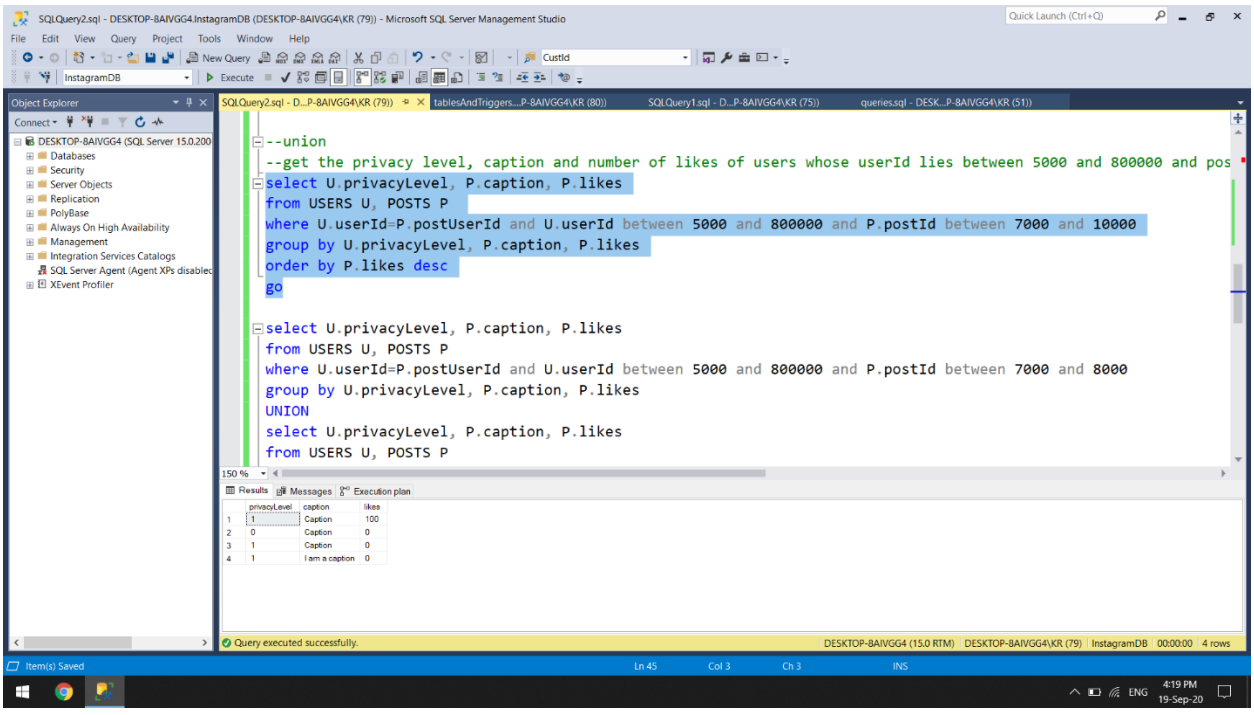
USE CASE 2: UNION

The aim is to get the privacy level, caption and number of likes of users whose userId lies between 5000 and 800000 and postId lies between 7000 and 10000 in the descending order of number of likes.

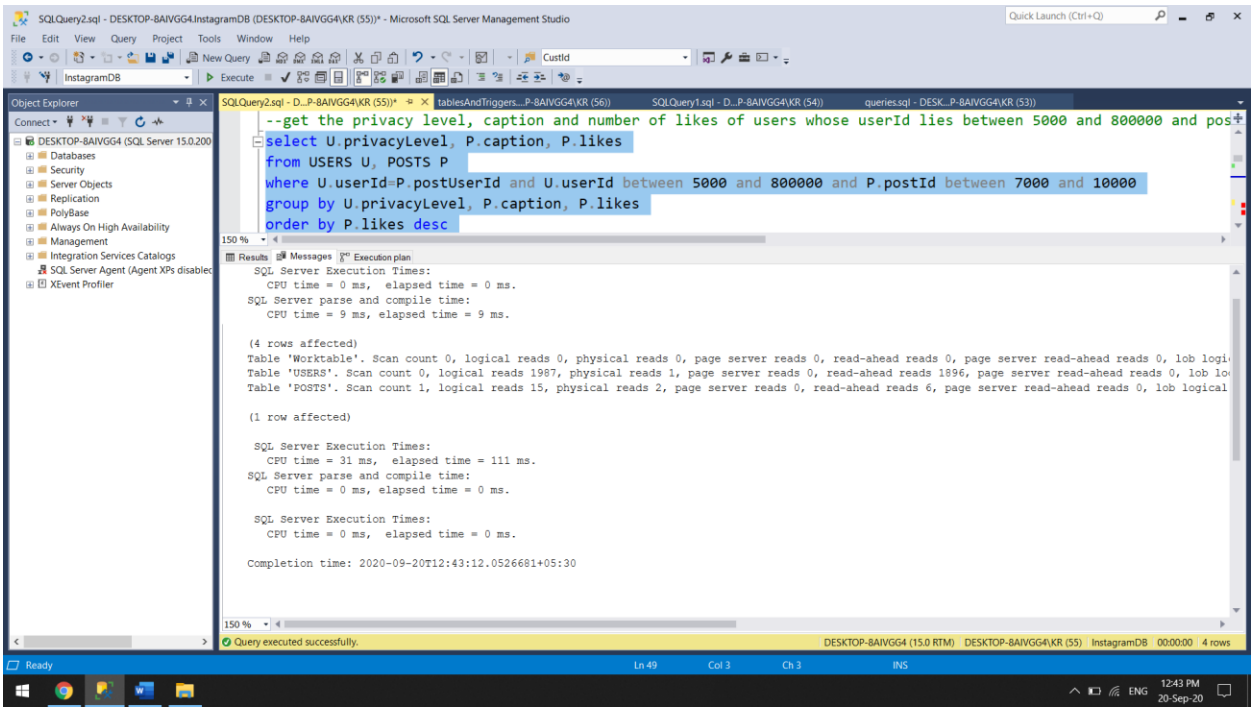
Version 1:
```
select U.privacyLevel, P.caption, P.likes
from USERS U, POSTS P
where U.userId=P.postUserId and U.userId between 5000 and 800000 and P.postId between 7000 and 10000
group by U.privacyLevel, P.caption, P.likes
order by P.likes desc
```

The visualization of this query could be that it fetches the required attributes from both the tables simultaneously, based on the given constraints, all of which are specified together.

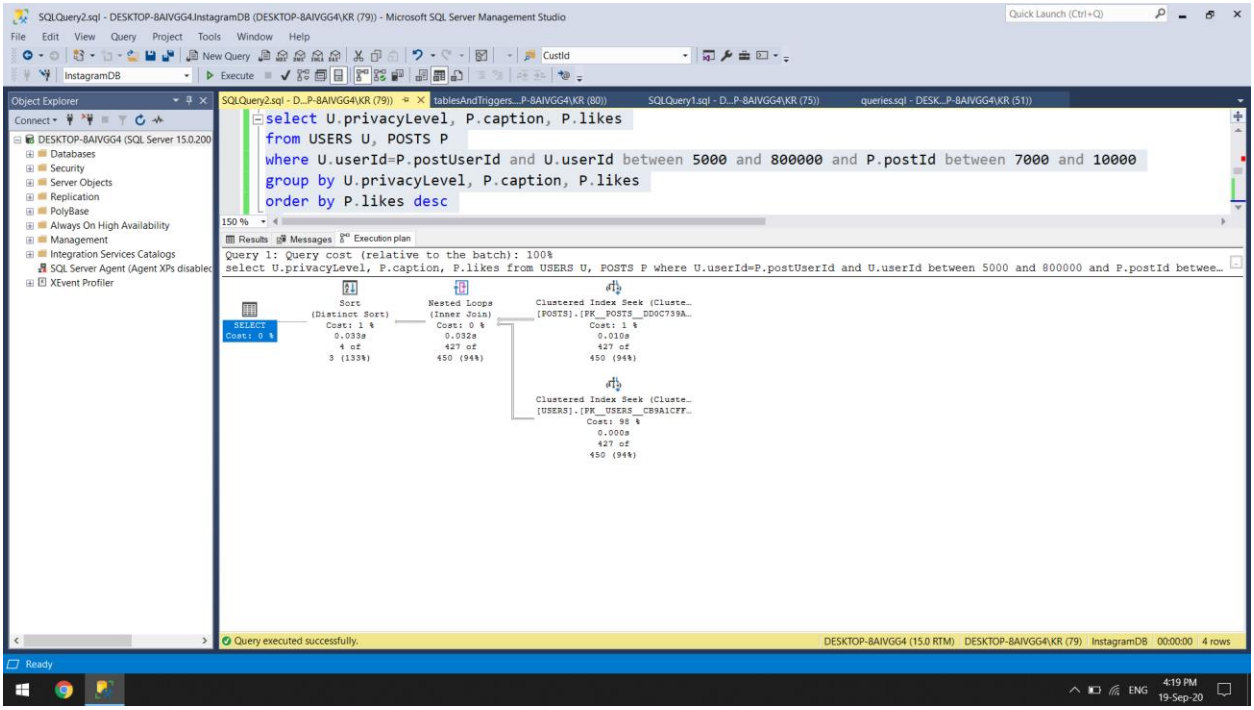
The output of the query is shown.



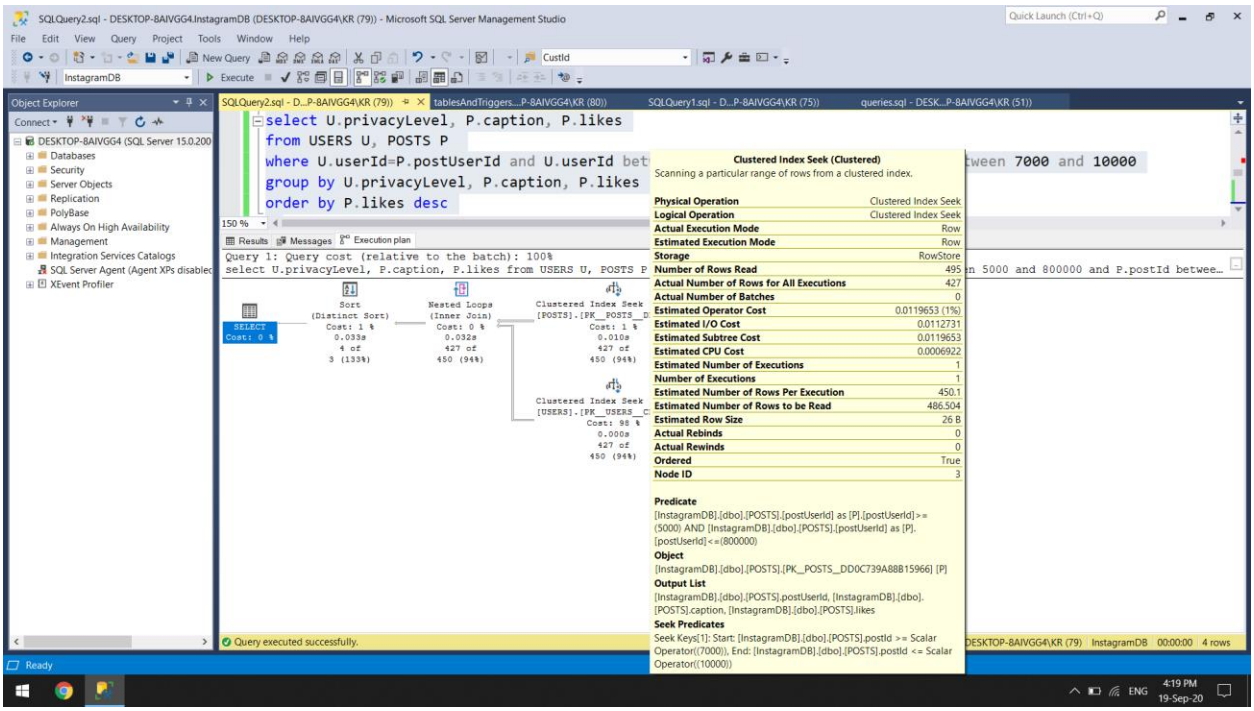
Notice for the USERS table, scan count is 0, number of logical reads is 1987 and number of physical reads is 1. And for the POSTS table, scan count is 1, number of logical reads is 15 and number of physical reads is 2.



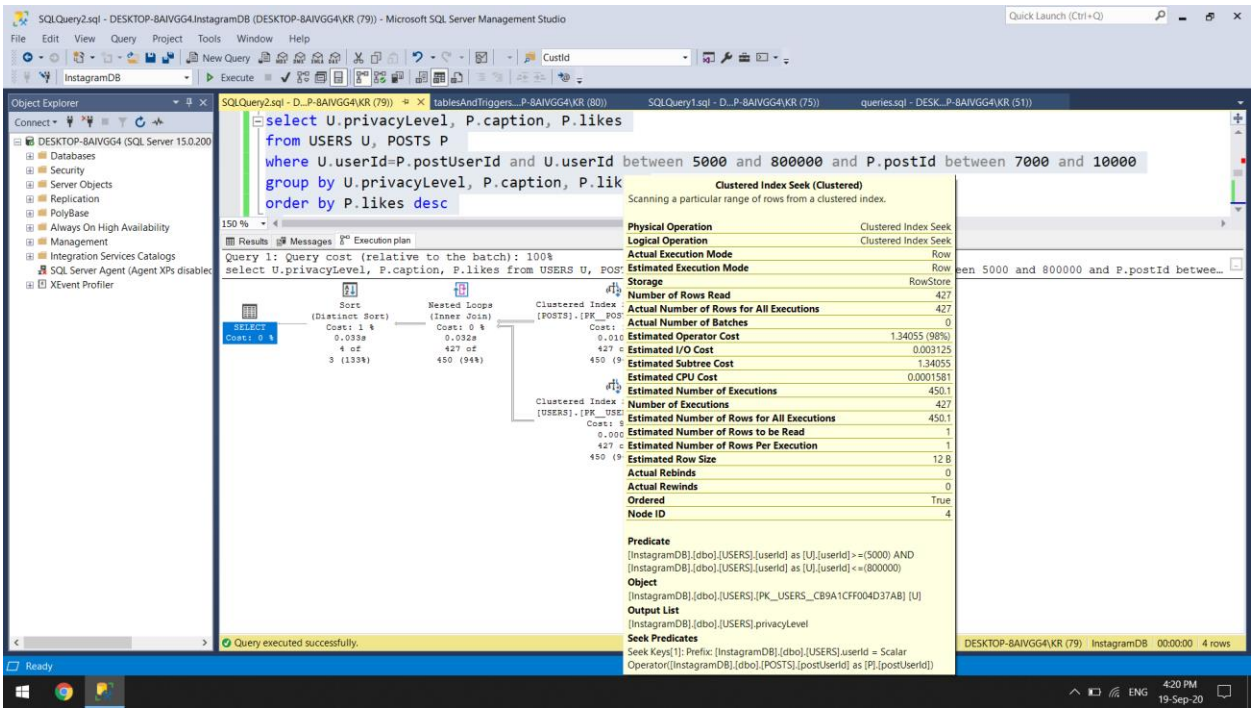
This is the execution plan of the query.



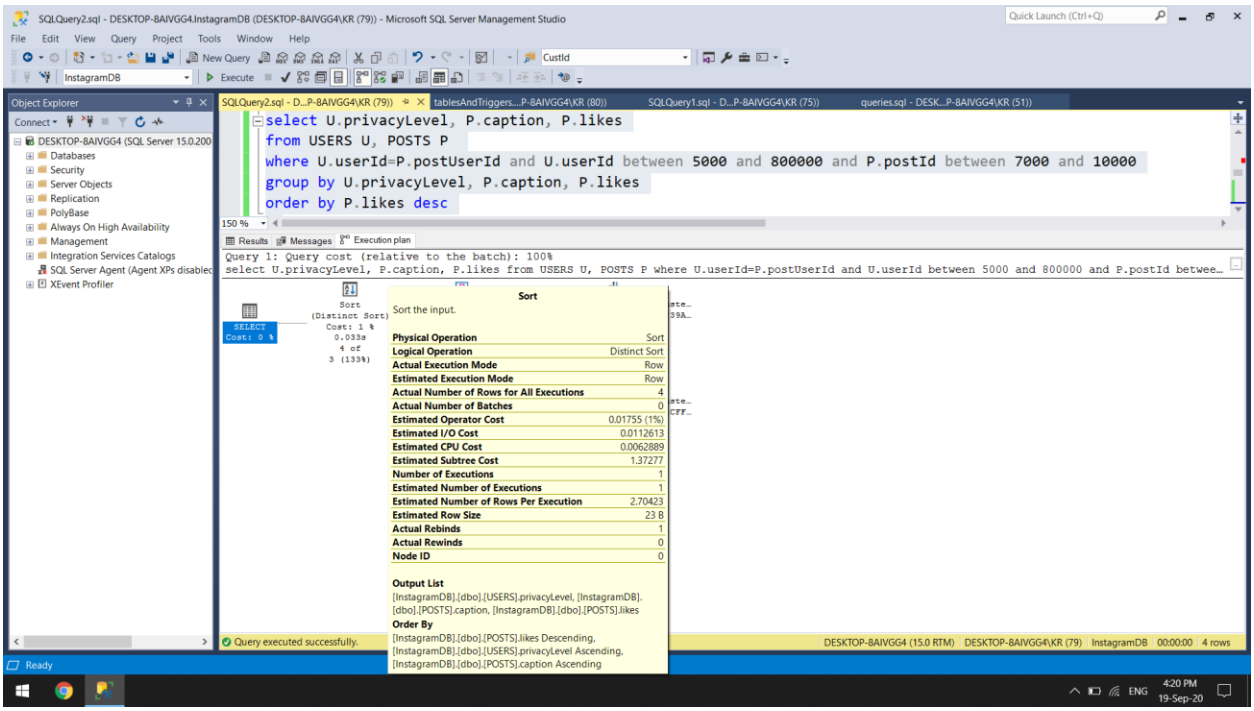
A clustered index scan is performed on POSTS to seek the tuples where the primary key postId is >7000 and <10000, as specified in the query. So only those tuples that satisfy these conditions are projected in the first step.



A clustered index scan is performed on USERS to seek the tuples that are currently clustered by the primary key userId. This can be seen in the seek predicates. After this seeking is done to get the rows based on the index (primary key index PK__USERS__CB9A1CFF004D37AB), filtering is done based on the condition for userId as we can see under Predicate.



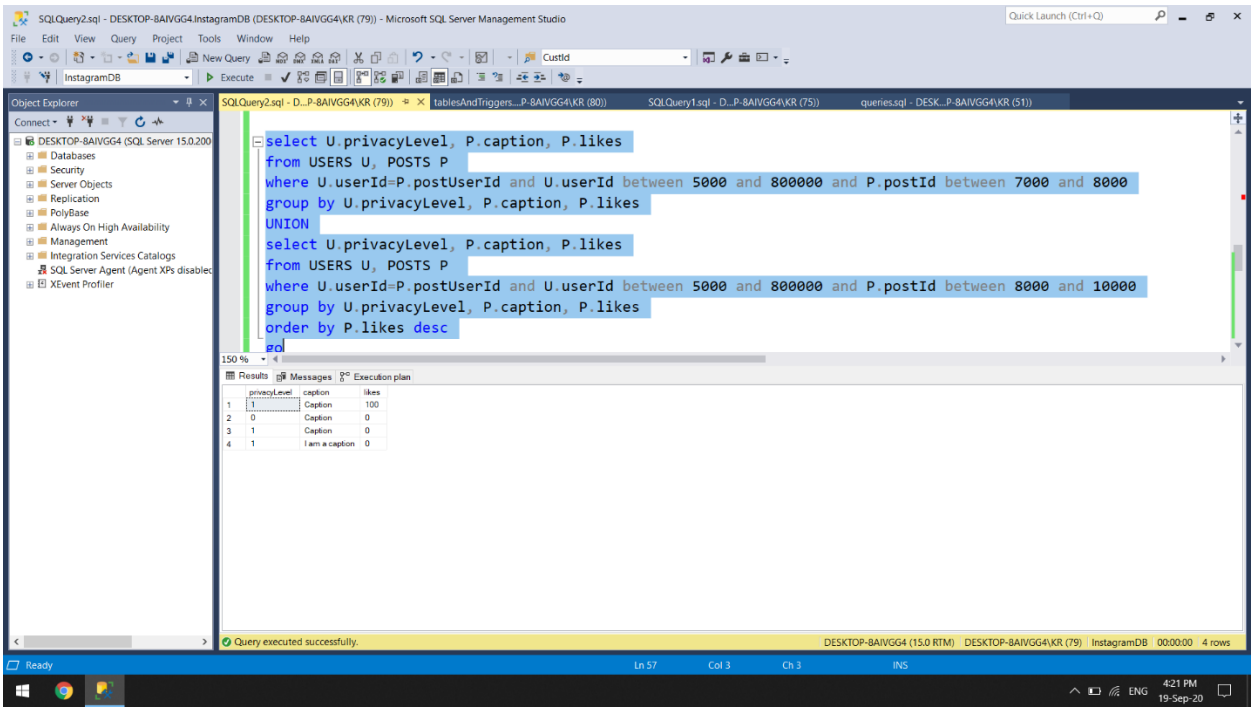
After the inner join, sorting is done to order the rows based on the specified attributed in the query and the output list has all the attributes we mentioned in the selList of the query.



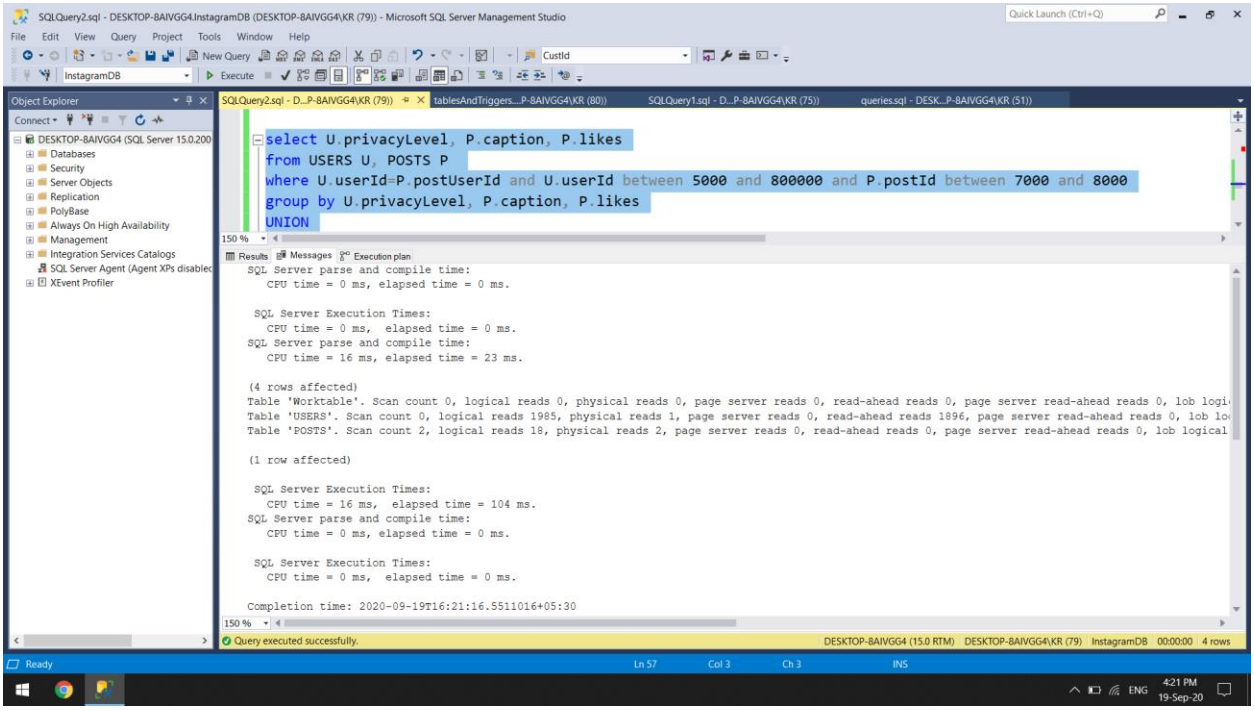
Version 2:
```
select U.privacyLevel, P.caption, P.likes
from USERS U, POSTS P
where U.userId=P.postUserId and U.userId between 5000 and 800000 and P.postId between 7000 and 8000
group by U.privacyLevel, P.caption, P.likes
UNION
select U.privacyLevel, P.caption, P.likes
from USERS U, POSTS P
where U.userId=P.postUserId and U.userId between 5000 and 800000 and P.postId between 8000 and 10000
group by U.privacyLevel, P.caption, P.likes
order by P.likes desc
```

This query uses UNION. Here, we have split the range condition for postUserId into two ranges instead of one and joined the results using the UNION operator.

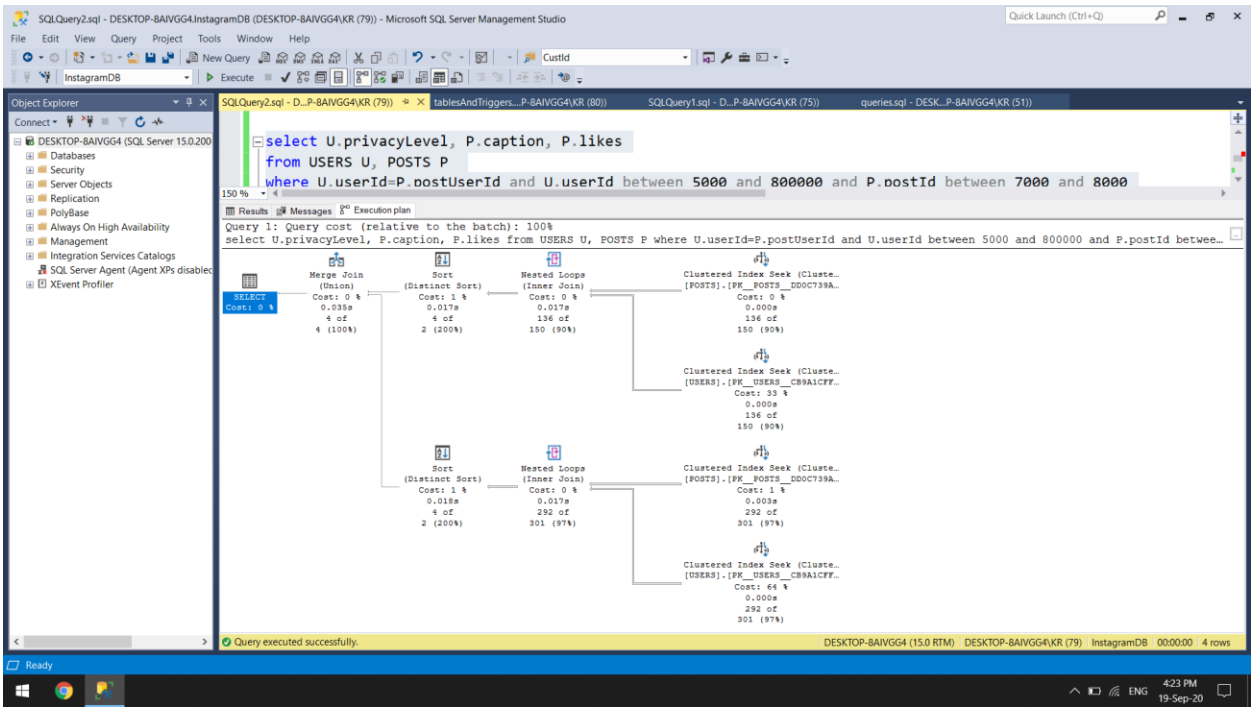
The output of the query is shown. As expected, it is the same as that of Version 1.



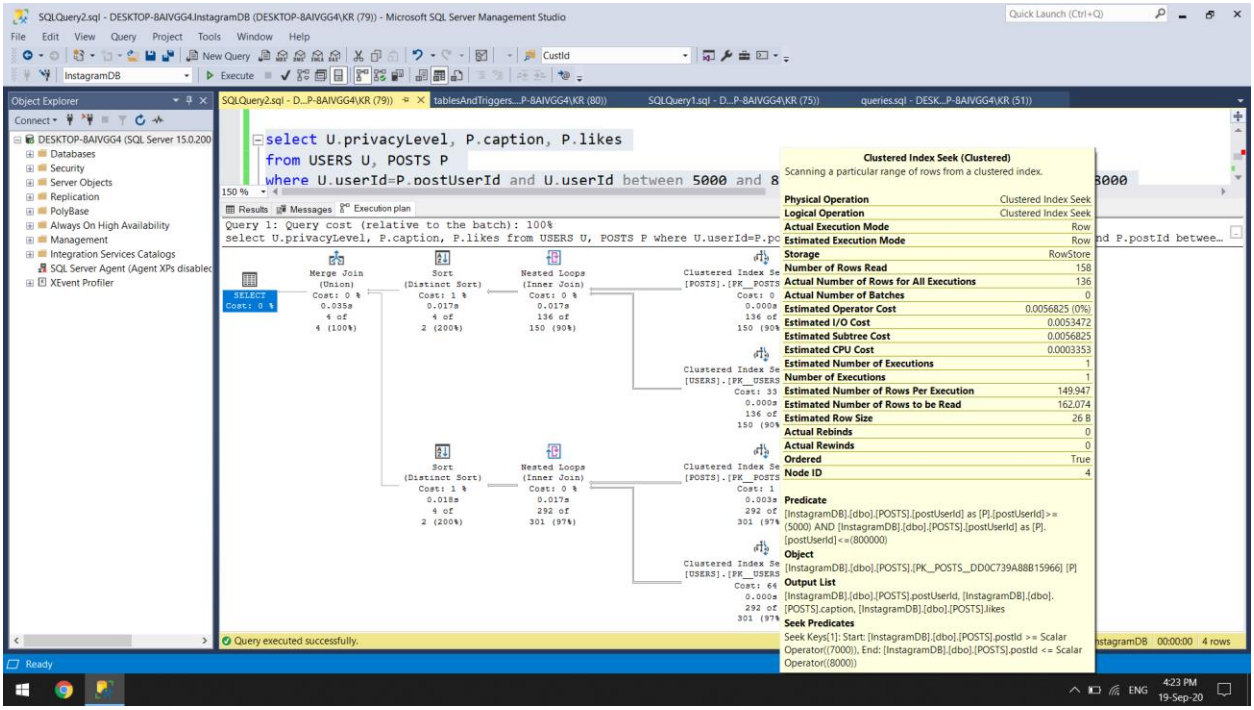
For the USERS table, scan count is 0, number of logical reads is 1985 and number of physical reads is 1. And for the POSTS table, scan count is 2, number of logical reads is 18 and number of physical reads is 2.



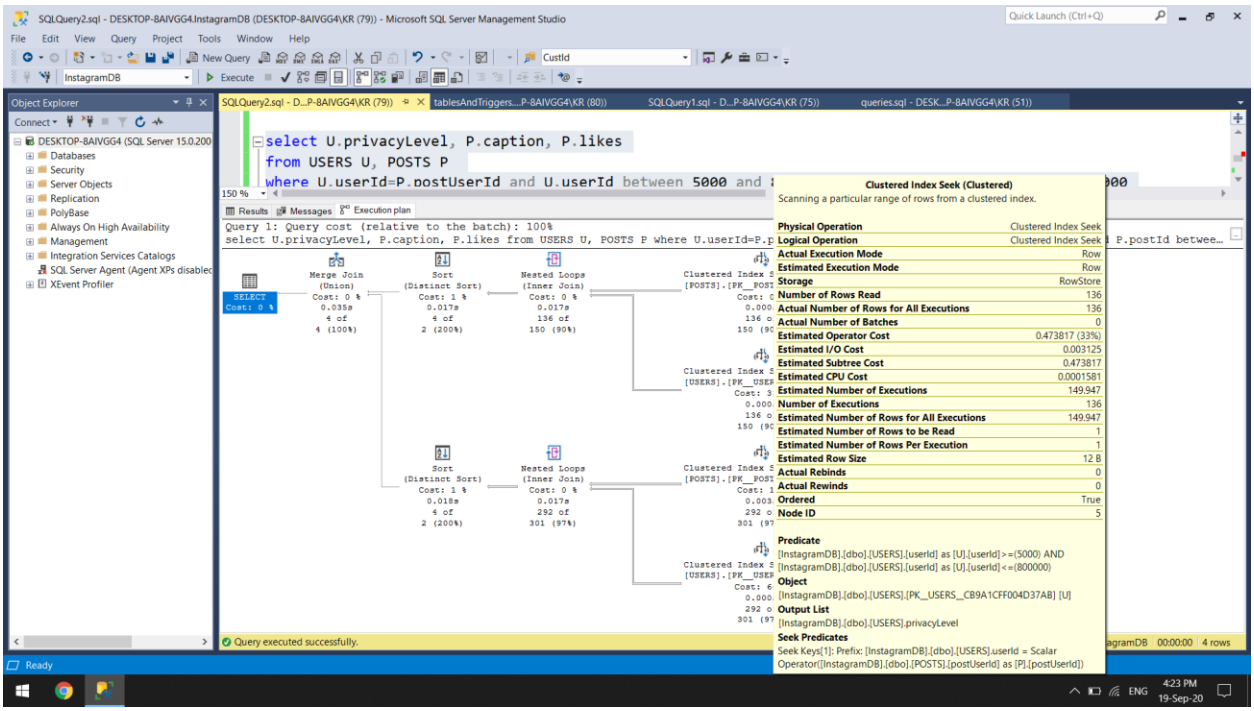
The execution plan is shown below. The same clustered index seek is performed on both the table, for both the ranges separately. The results from both the tables are joined and sorted individually and then finally merged together. The detailed specifications can be seen in the images below.



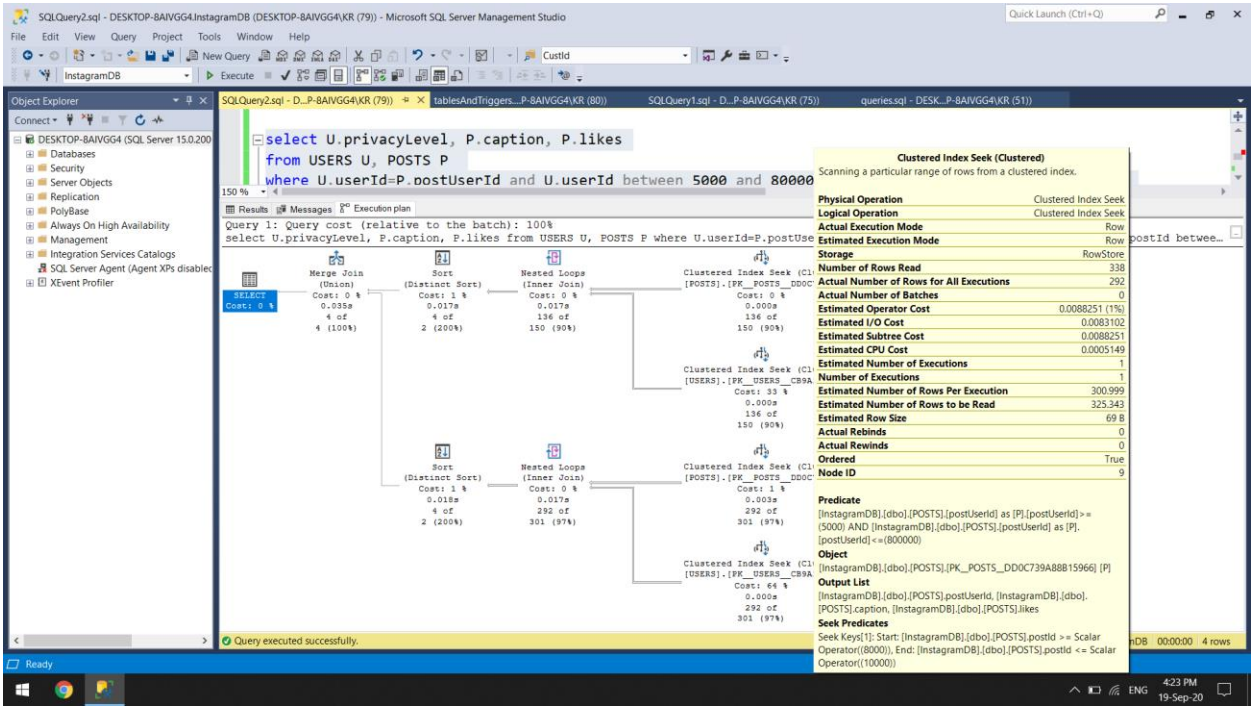
For POSTS table in the first range-



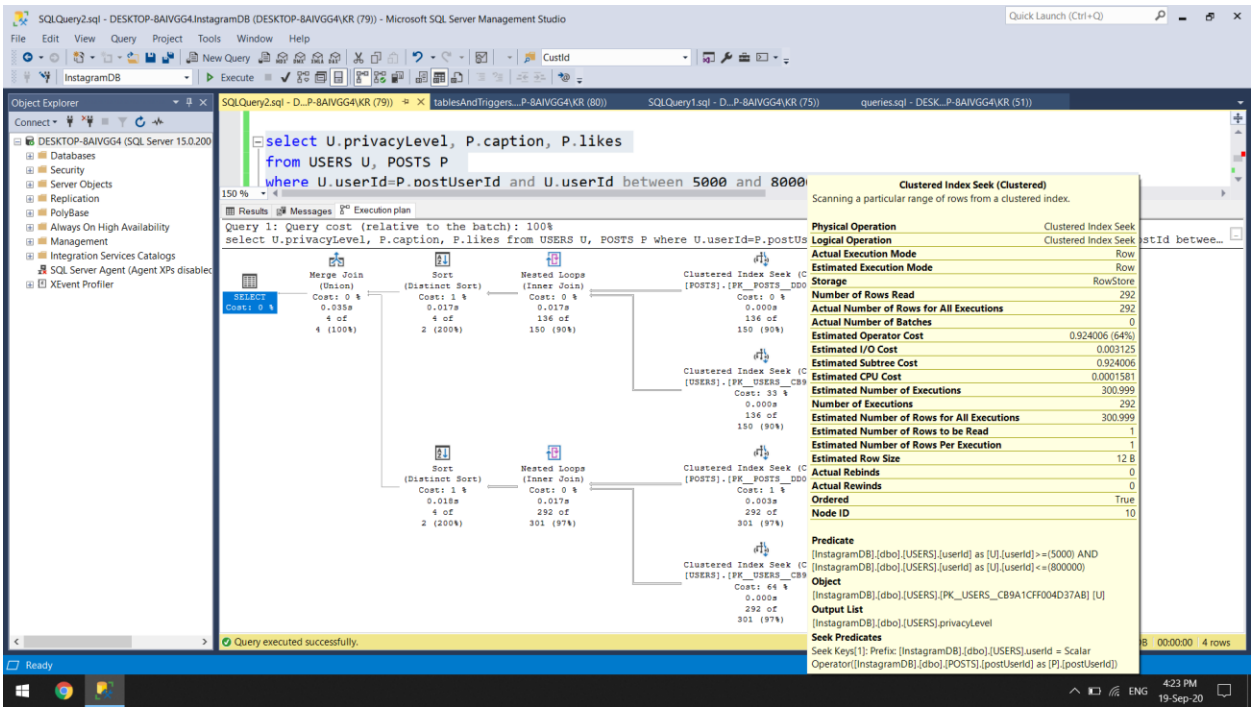
For USERS table in the first range-



For POSTS table in the second range-



For USERS table in the second range-

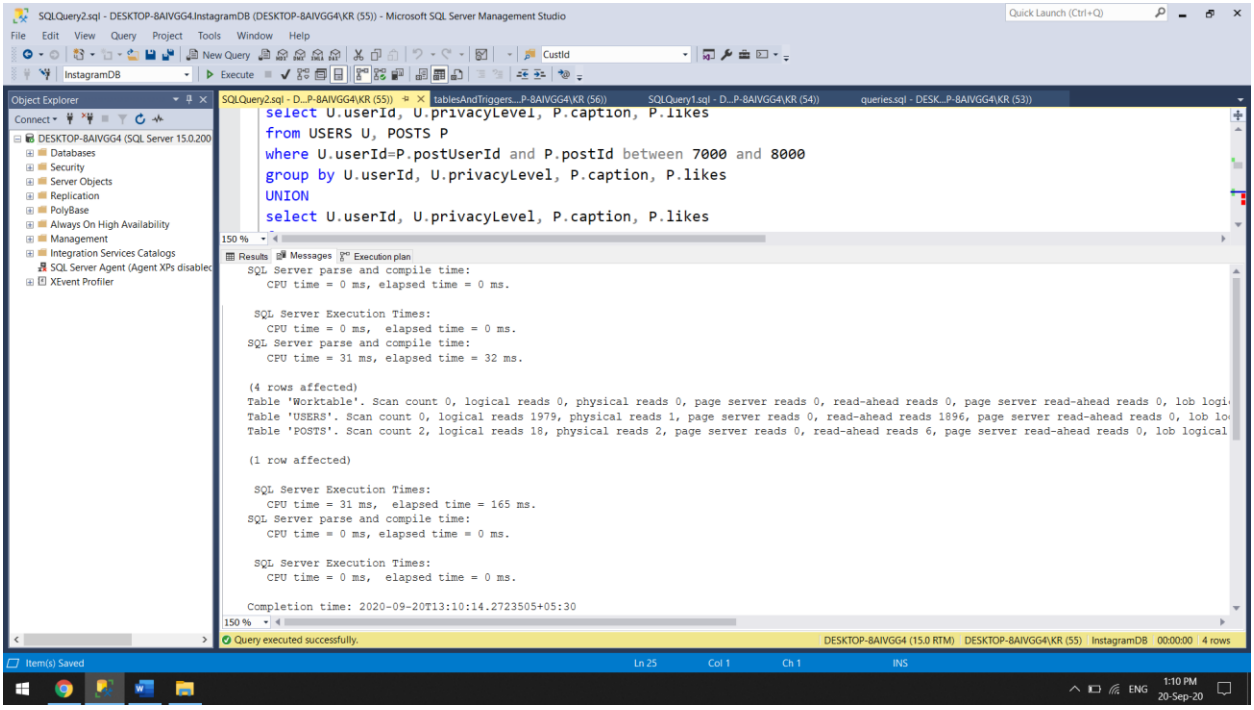


Version 3:

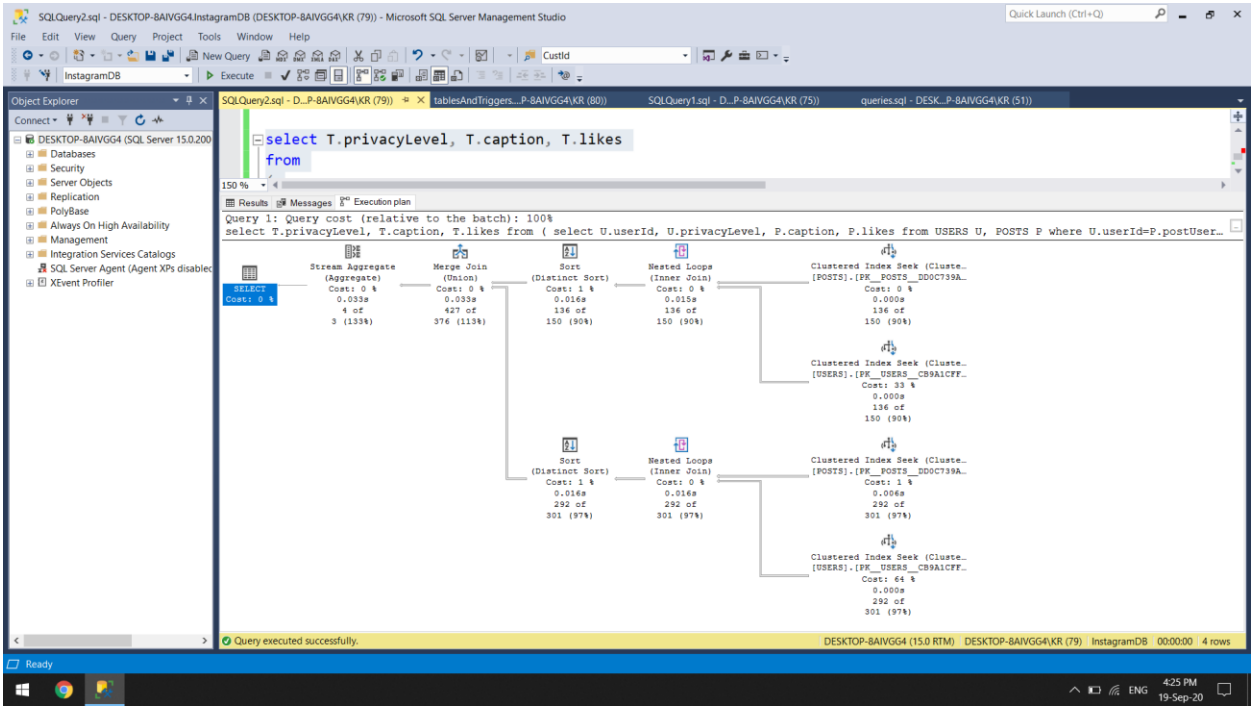
```
select T.privacyLevel, T.caption, T.likes
from
(
select U.userId, U.privacyLevel, P.caption, P.likes
from USERS U, POSTS P
where U.userId=P.postUserId and P.postId between 7000 and 8000
group by U.userId, U.privacyLevel, P.caption, P.likes
UNION
select U.userId, U.privacyLevel, P.caption, P.likes
from USERS U, POSTS P
where U.userId=P.postUserId and P.postId between 8000 and 10000
group by U.userId, U.privacyLevel, P.caption, P.likes
) T
where T.userId between 5000 and 800000
group by T.privacyLevel, T.caption, T.likes
order by T.likes desc
```

In this query, we have selected the required attributes from both the tables by giving a range condition for postId. Instead of giving the entire range at once, we have split the range into two parts- 7000 to 8000 and 8000 to 10000 and joined both the results using the UNION operator. On that result, we have specified the condition for userId and ordered the final result. This is kind of a mix of Version 1 and Version 2 to produce a very elaborate way of writing the query. It seems like the worst case scenario, however, we see that the query compiler optimizes the query to execute it in the same way as Version 2, as depicted in the images below.

The output message of the query is shown below. It is about the same as the previous two versions.

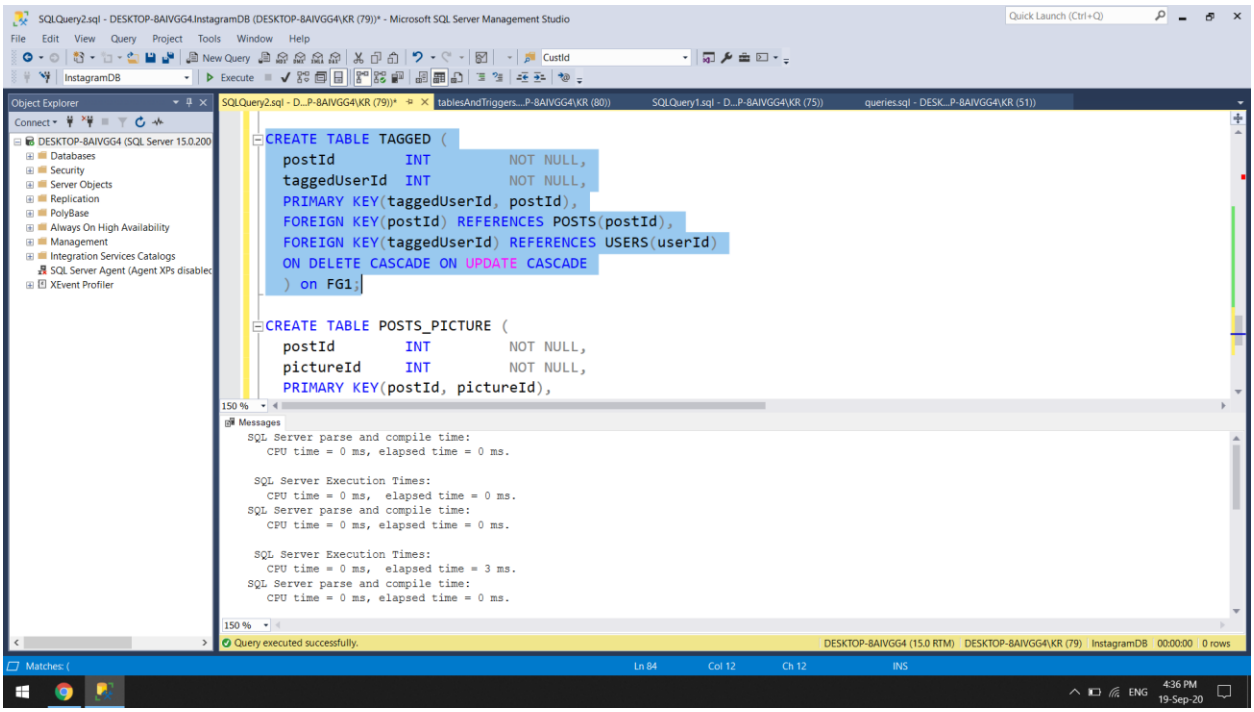


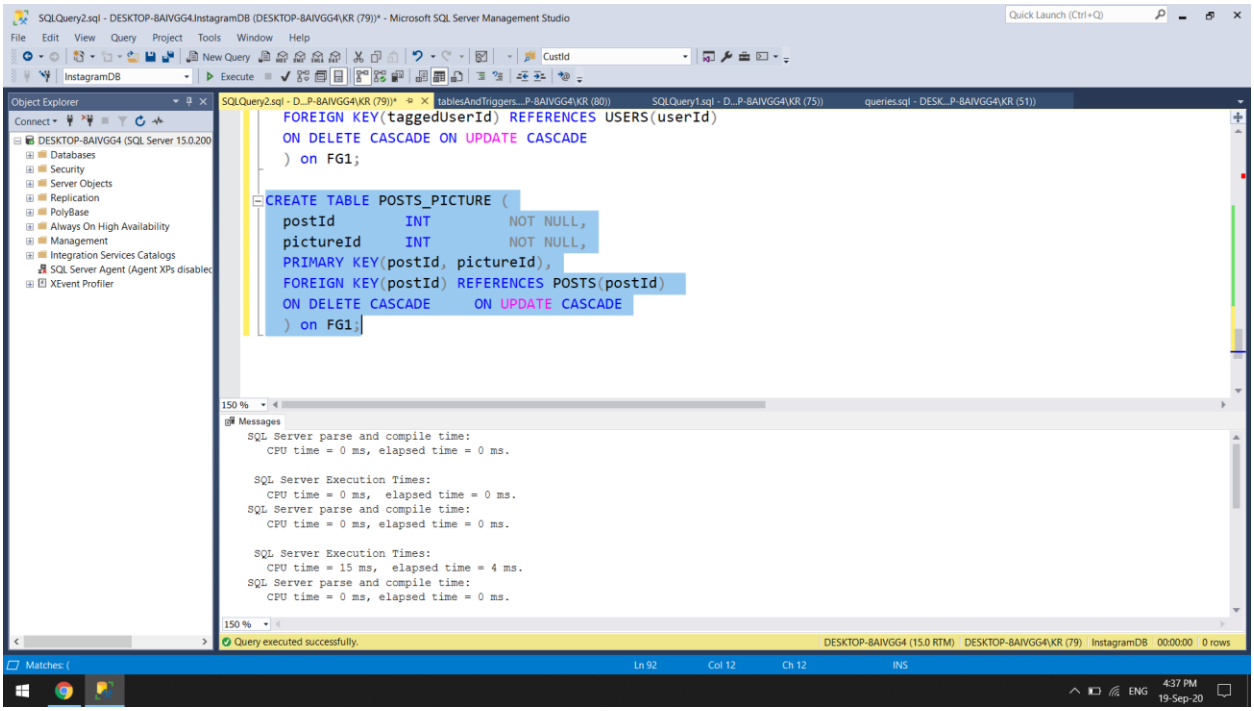
This is the execution plan of the query. It is the same as that of Verison 2, just that we have a Steam Aggregate step before the final output because along with the group by clauses on table U and table P, we have specified the group by clause on the table T which is the union of tables U and P, whereas in Version 2, the group by clause was specified only on U and P. In effect, the execution plan has not changed from that of Version 2.



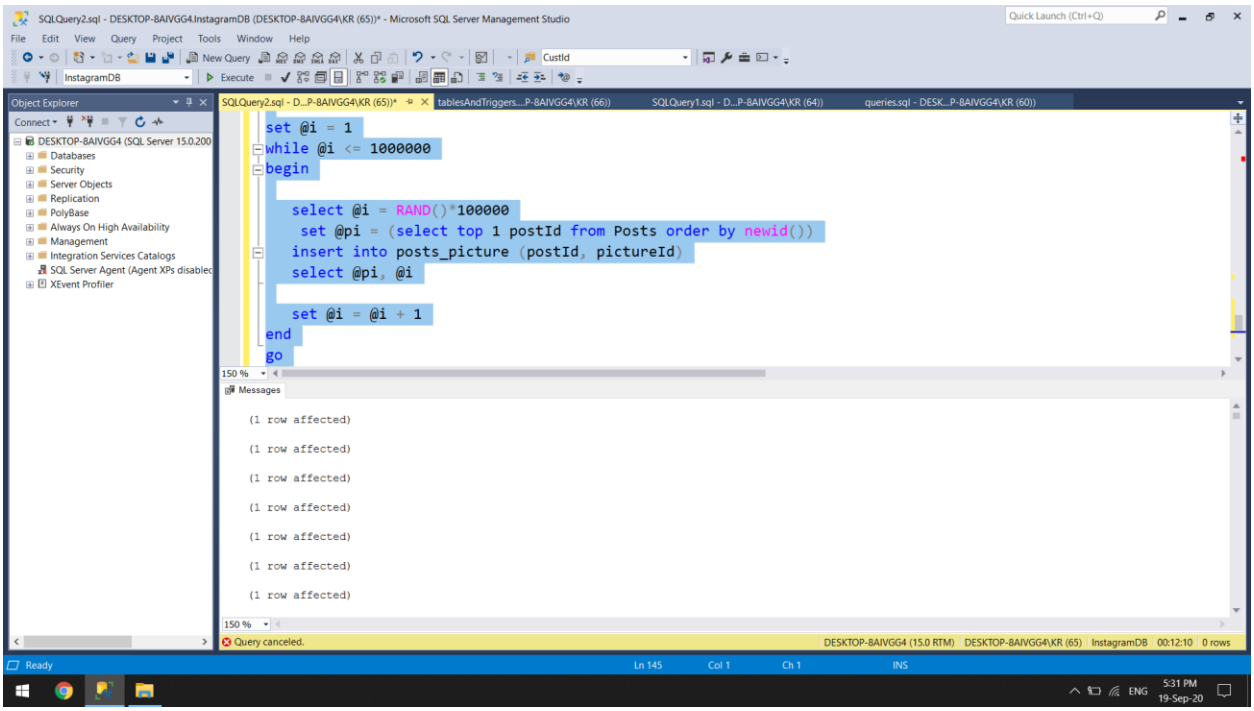
Use case 3: Multi-table join (4 tables)

Until now, we looked only at two tables- USERS and POSTS. For the join operation, we create two more tables- TAGGED and POSTS_PICTURE.

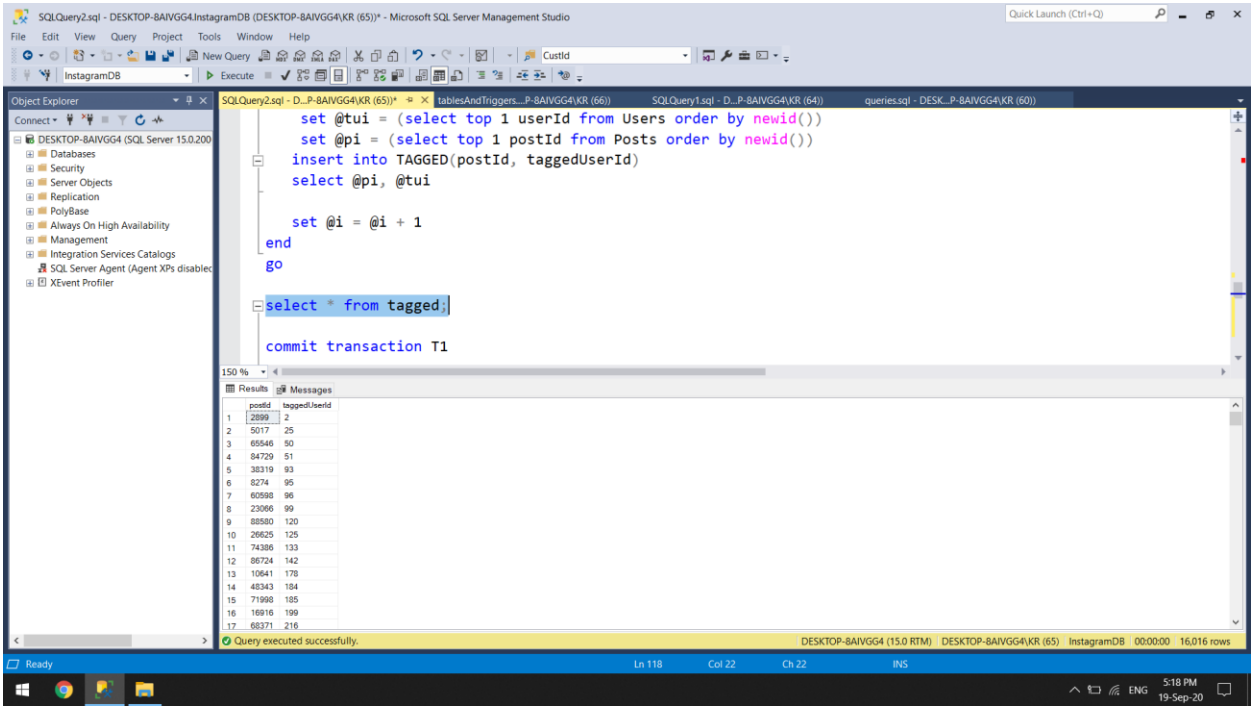


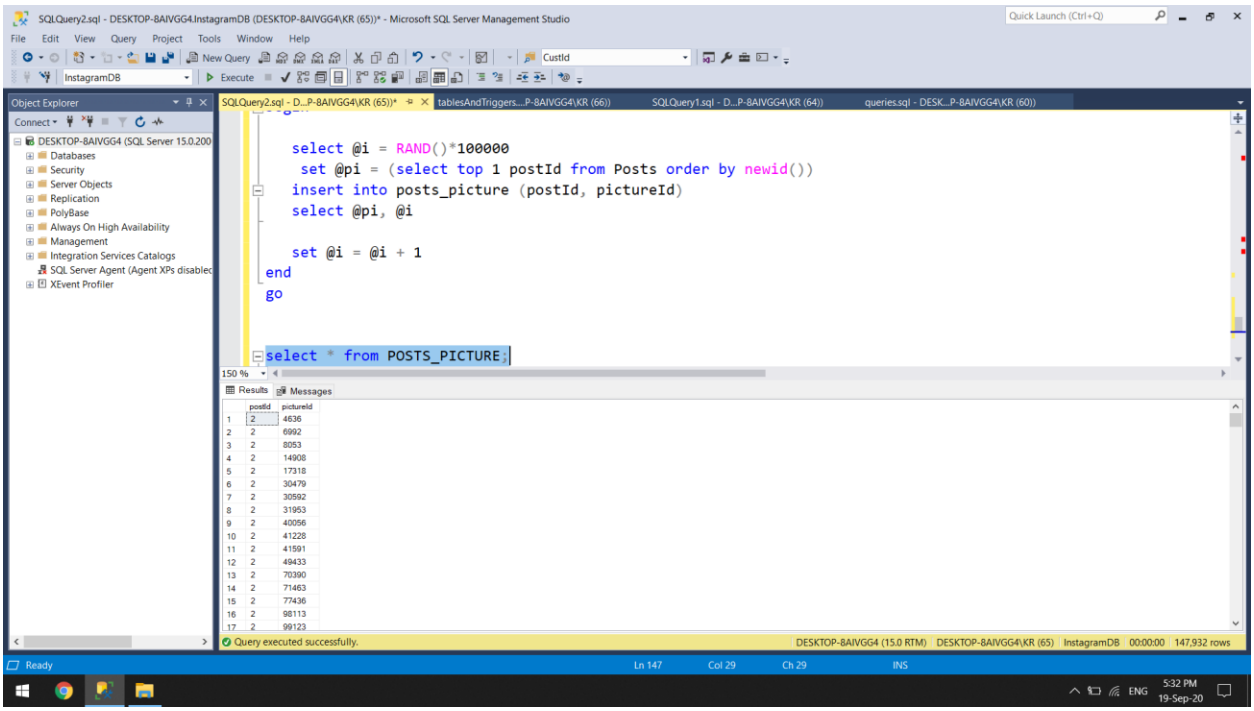


We populate these two tables with a couple of thousands of rows with random values.



We successfully insert 16016 rows into the TAGGED table and 147932 rows into POSTS_PICTURE.

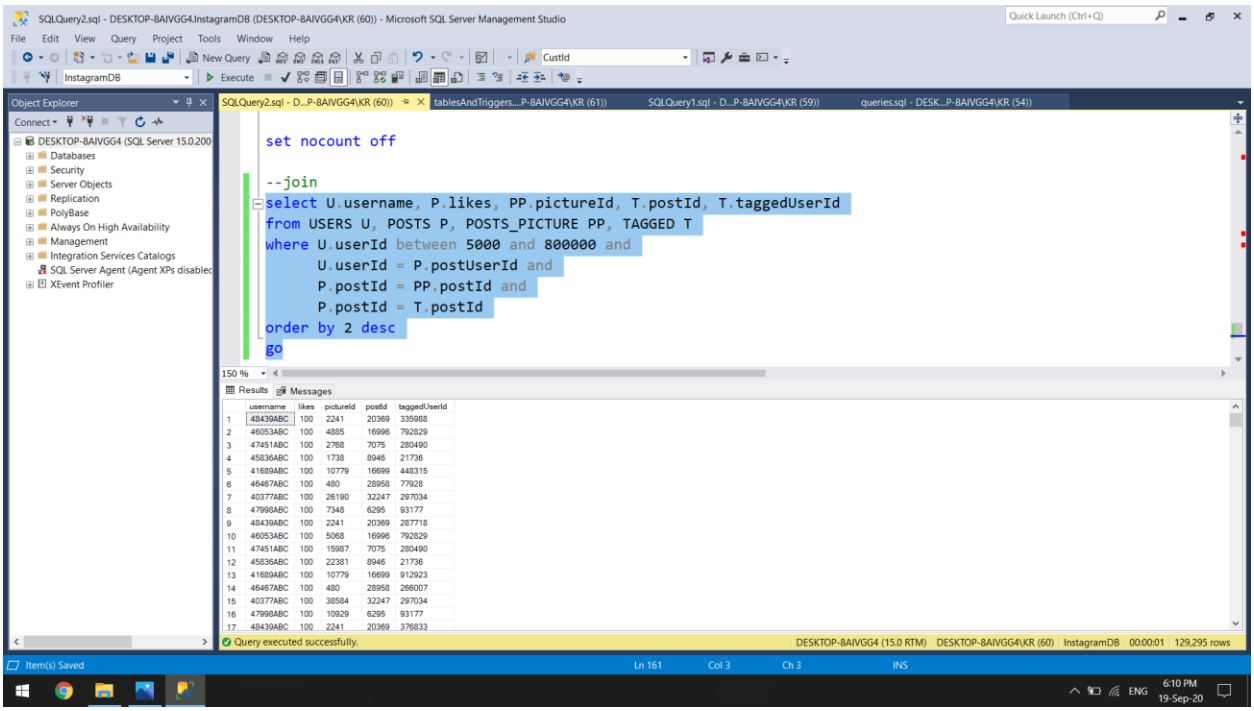




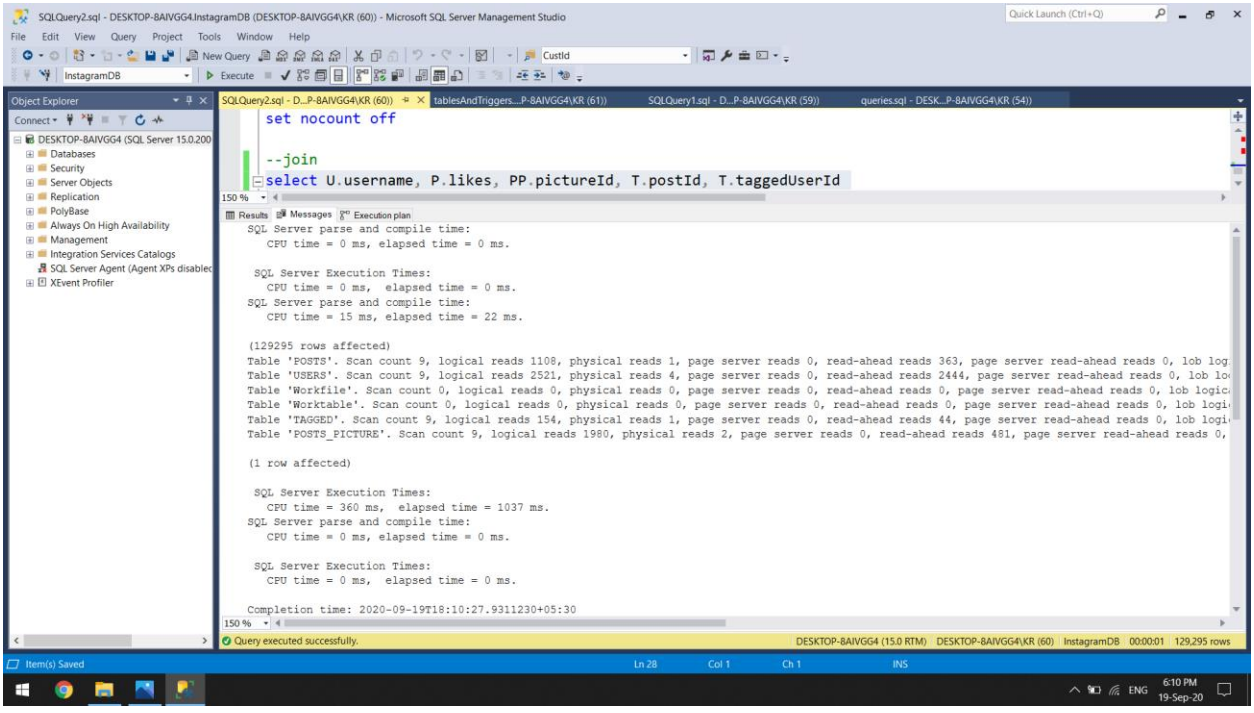
The aim of the join query is to join the 4 tables based on certain conditions. We try specifying conditions on attributes from different tables and see how the execution plan changes. The point to observe in every execution plan is that the tables to join are selected such that the number of rows is the least possible at that particular step.

Version 1:
```
select U.username, P.likes, PP.pictureId, T.postId, T.taggedUserId
from USERS U, POSTS P, POSTS_PICTURE PP, TAGGED T
where U.userId between 5000 and 800000 and
        U.userId = P.postUserId and
        P.postId = PP.postId and
        P.postId = T.postId
order by 2 desc
```

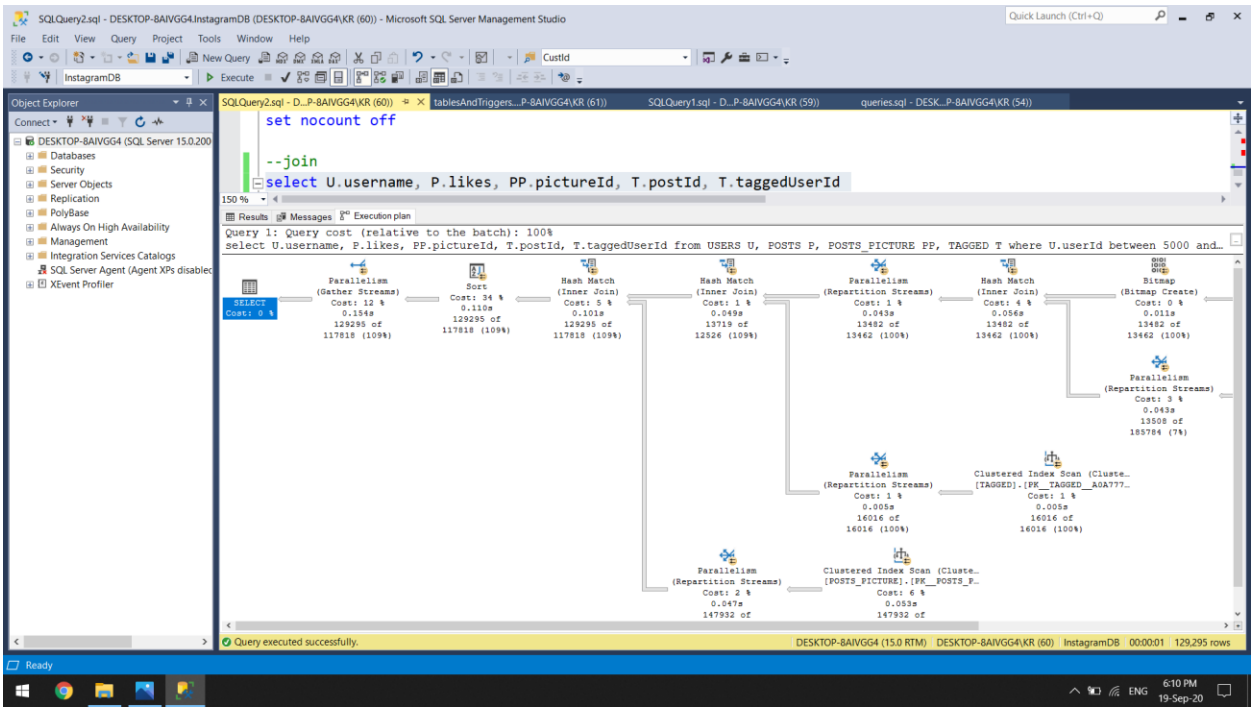
The below query specifies the condition on the primary key, userId, of the USERS table.

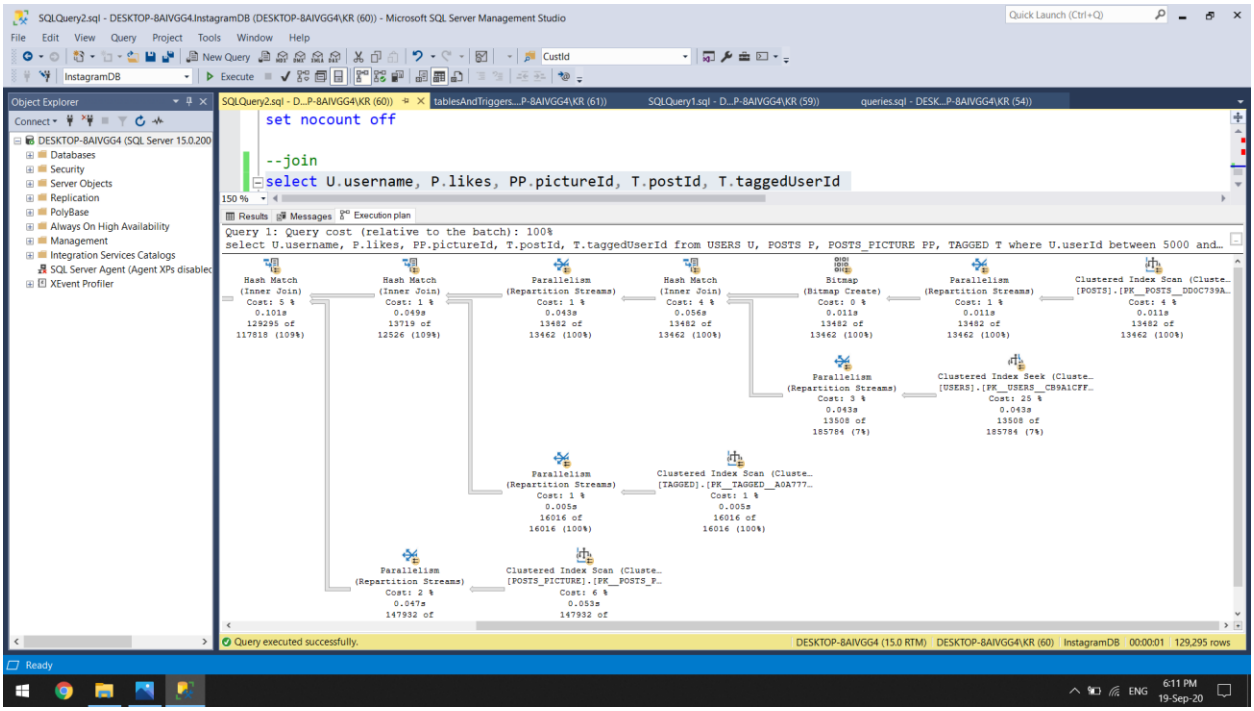


The following is the message we get for this query.

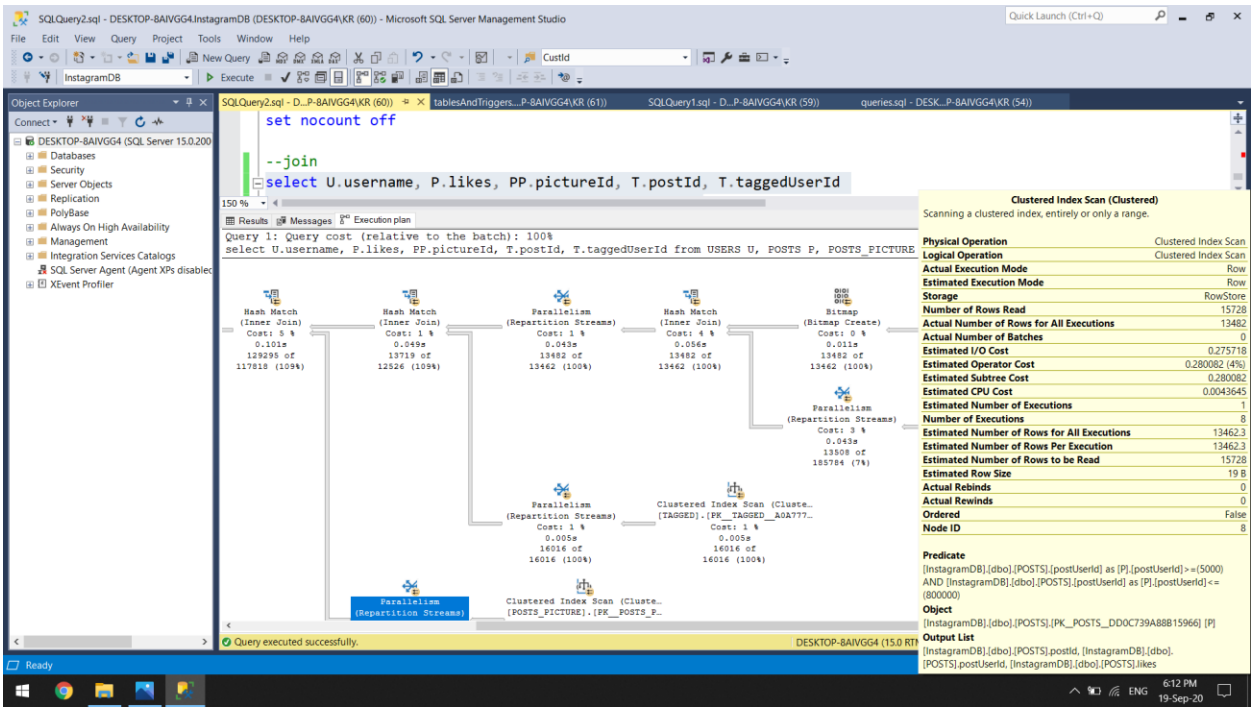


This is the execution plan for the query.

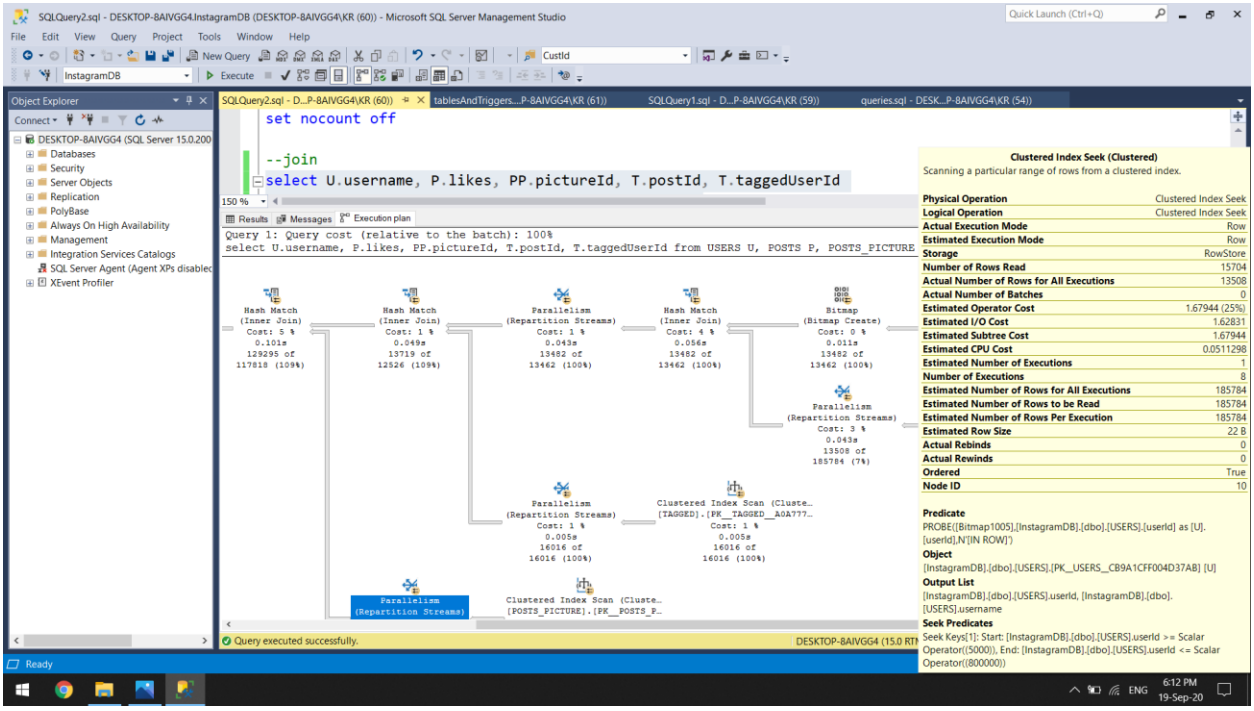




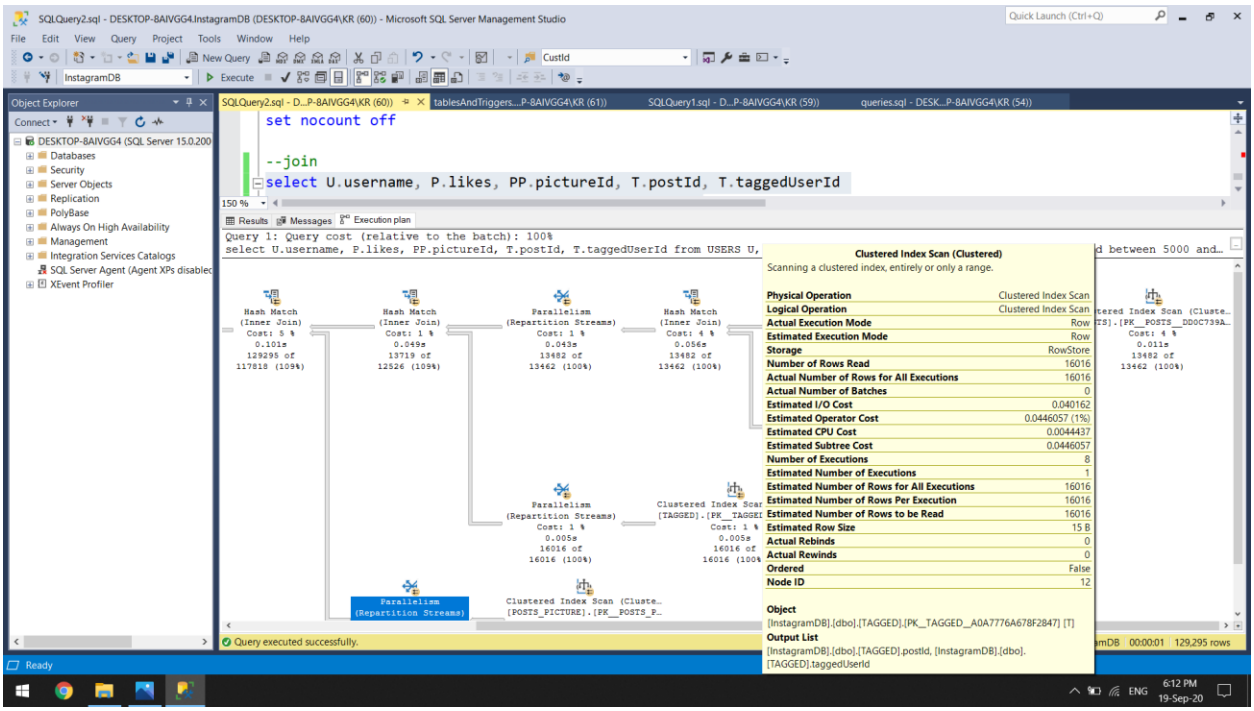
First, 13482 rows are fetched from POSTS.



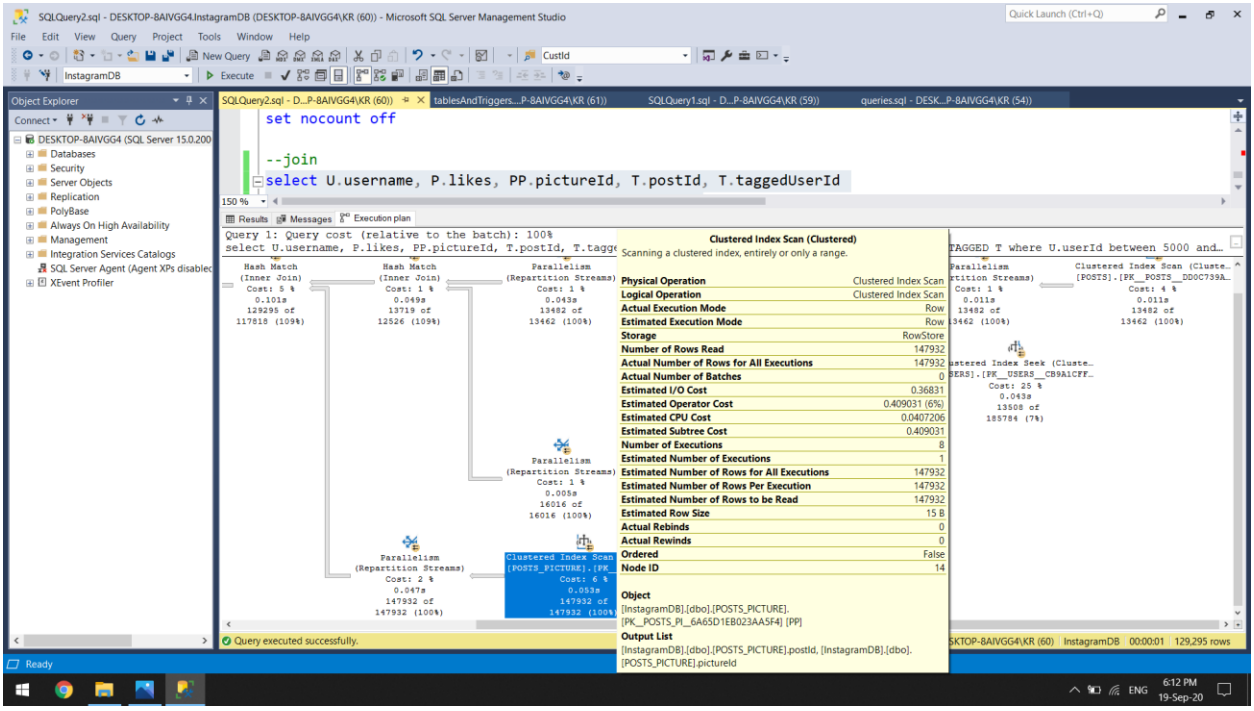
The second table chosen is USERS and 13508 rows (more than that of the previous table) are fetched from it.



The third table chosen is TAGGED and 16016 rows (more than that of the previous table) are fetched from it.



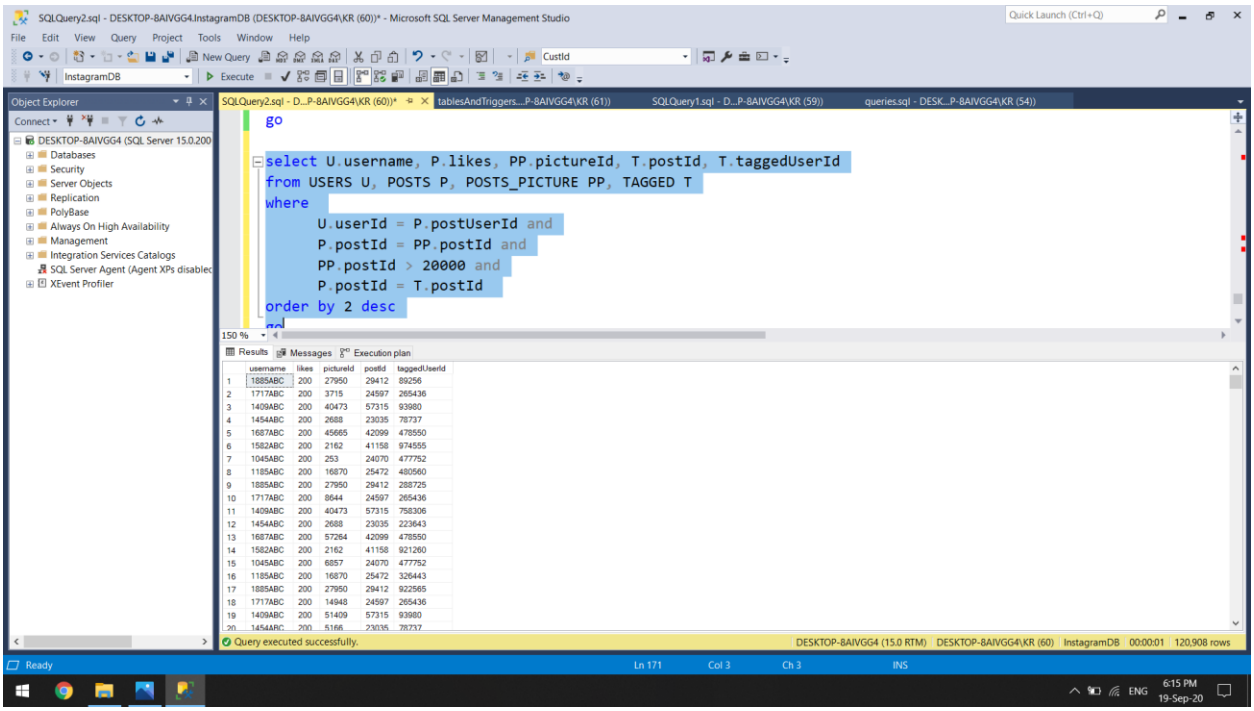
The fourth and final table for the join is POSTS_PICTURE. 147932 rows (more than that of the previous table) are fetched here.



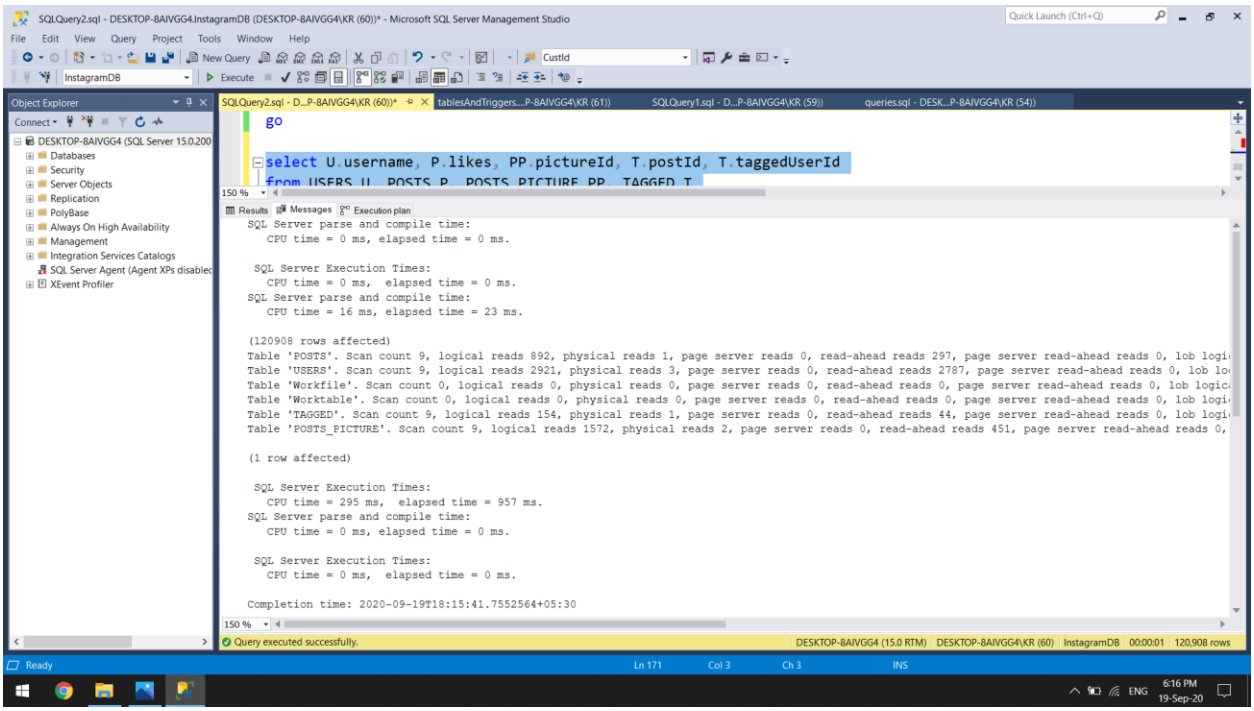
Version 2: select U.username, P.likes, PP.pictureId, T.postId, T.taggedUserId
from USERS U, POSTS P, POSTS_PICTURE PP, TAGGED T
where

```
        U.userId = P.postUserId and
        P.postId = PP.postId and
        PP.postId > 20000 and
        P.postId = T.postId
order by 2 desc
```

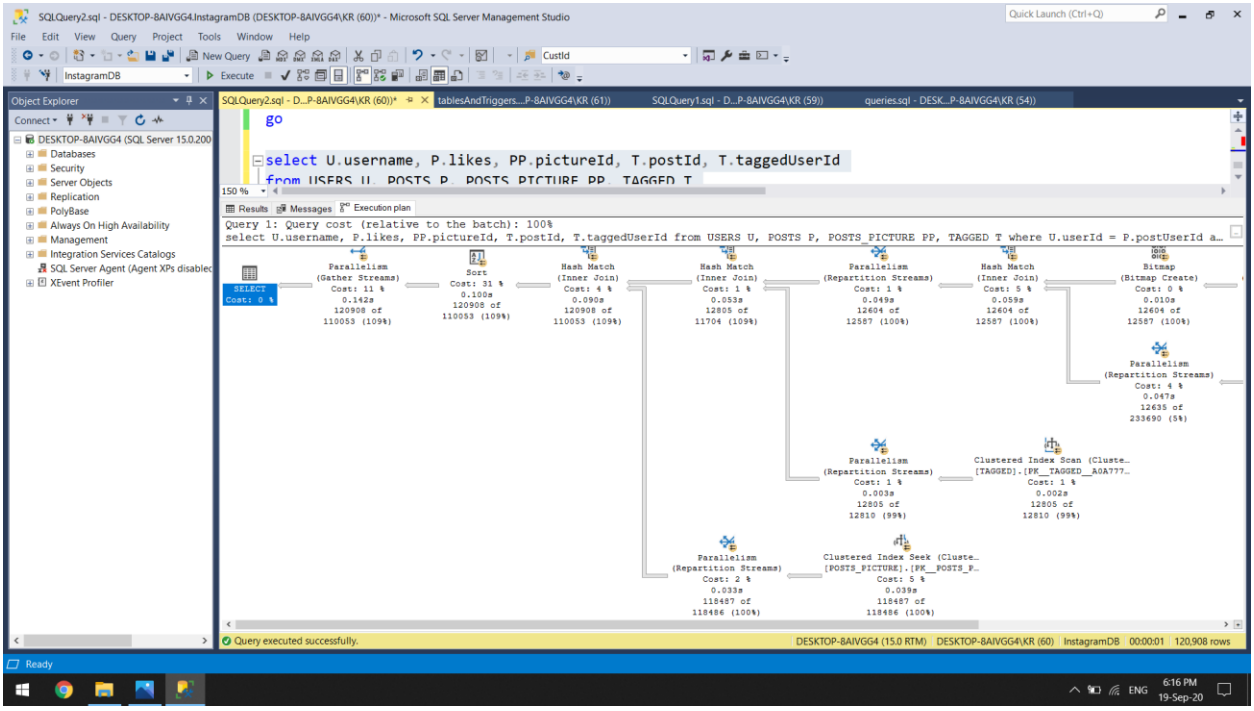
Here, the condition is on postId of POSTS_PICTURE (we have removed the previous condition on userId of USERS).

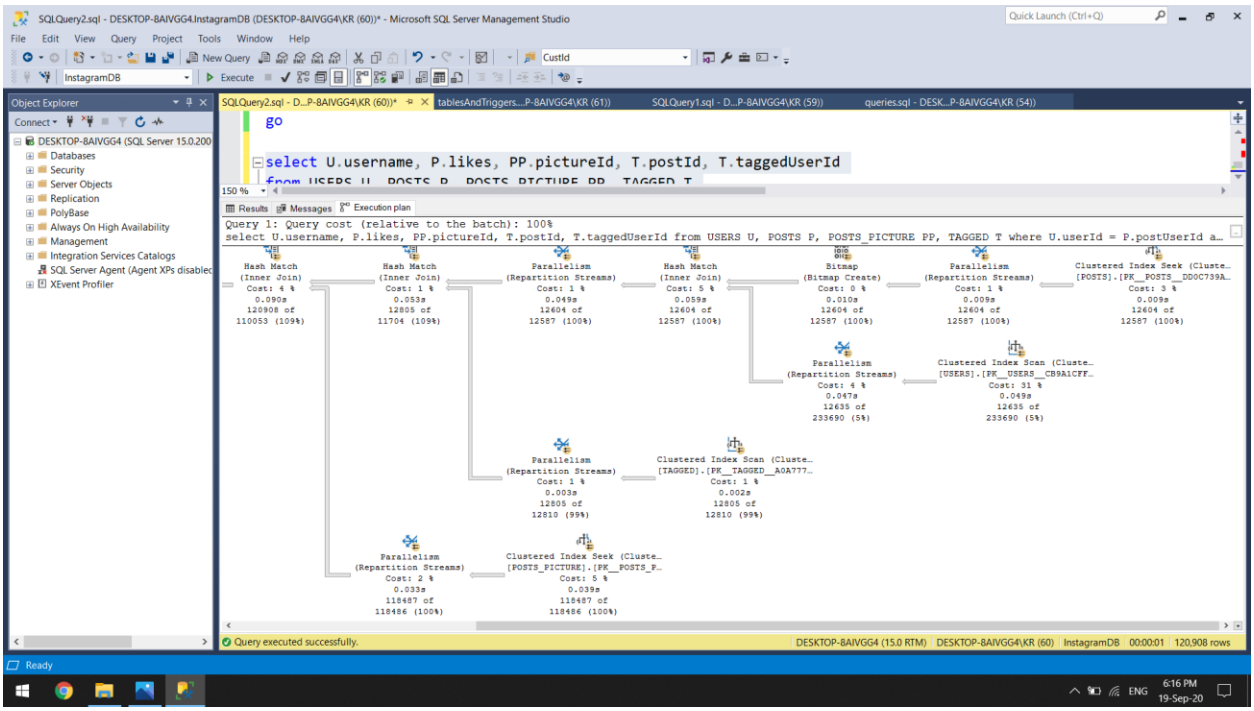


The following is the message output for the query.



The execution plan is shown below. Based on the current condition on postId of POSTS_PICTURE, the first table chosen is POSTS, followed by USERS, then TAGGED and finally POSTS_PICTURE.

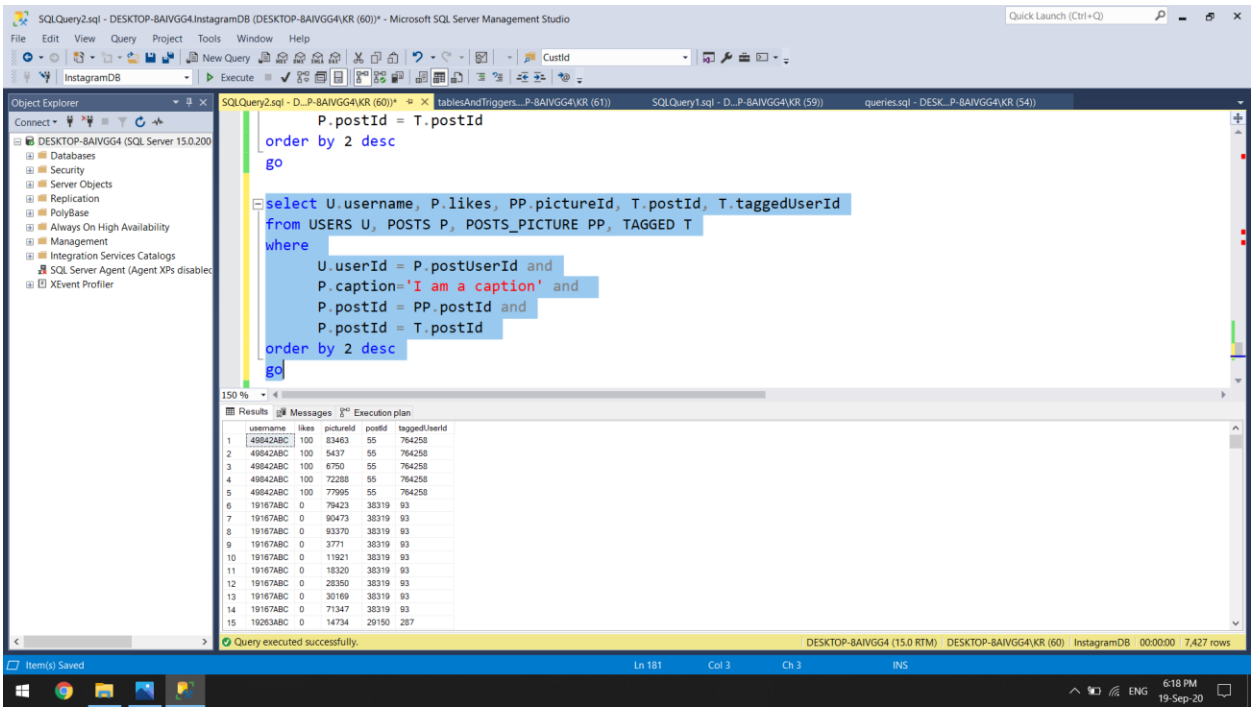




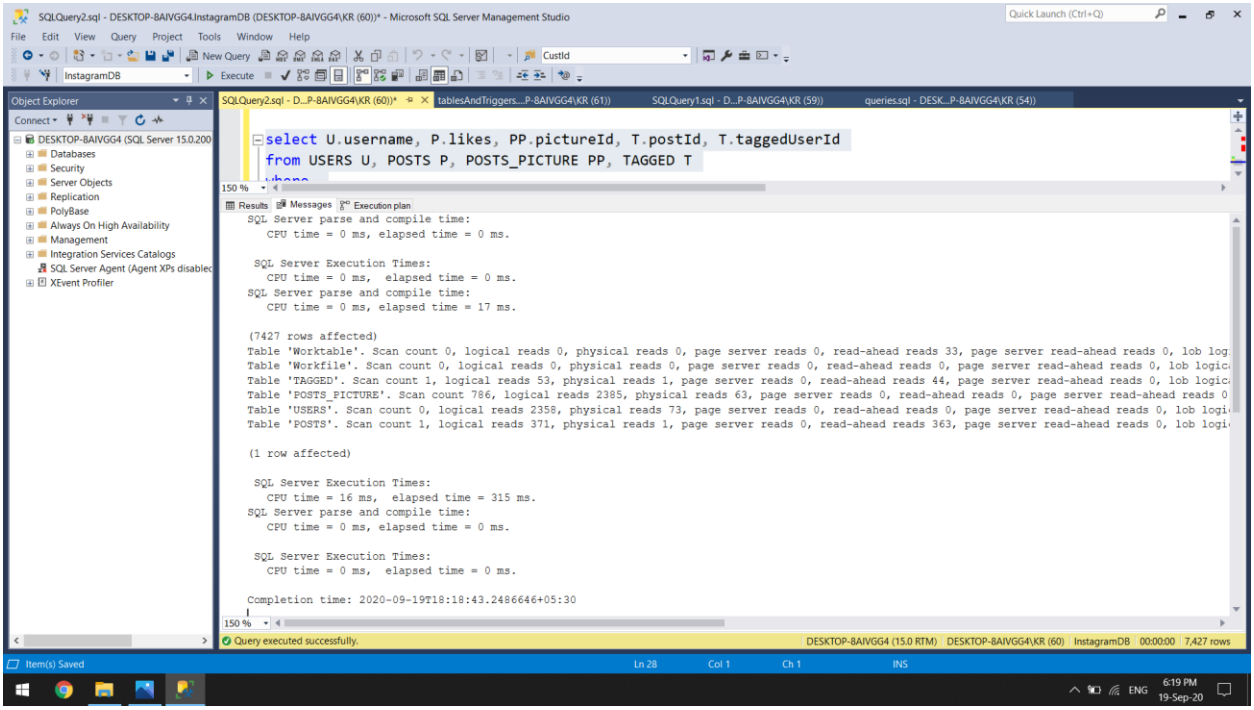
Version 3:

```
select U.username, P.likes, PP.pictureId, T.postId, T.taggedUserId
from USERS U, POSTS P, POSTS_PICTURE PP, TAGGED T
where
        U.userId = P.postUserId and
        P.caption='I am a caption' and
        P.postId = PP.postId and
        P.postId = T.postId
order by 2 desc
```

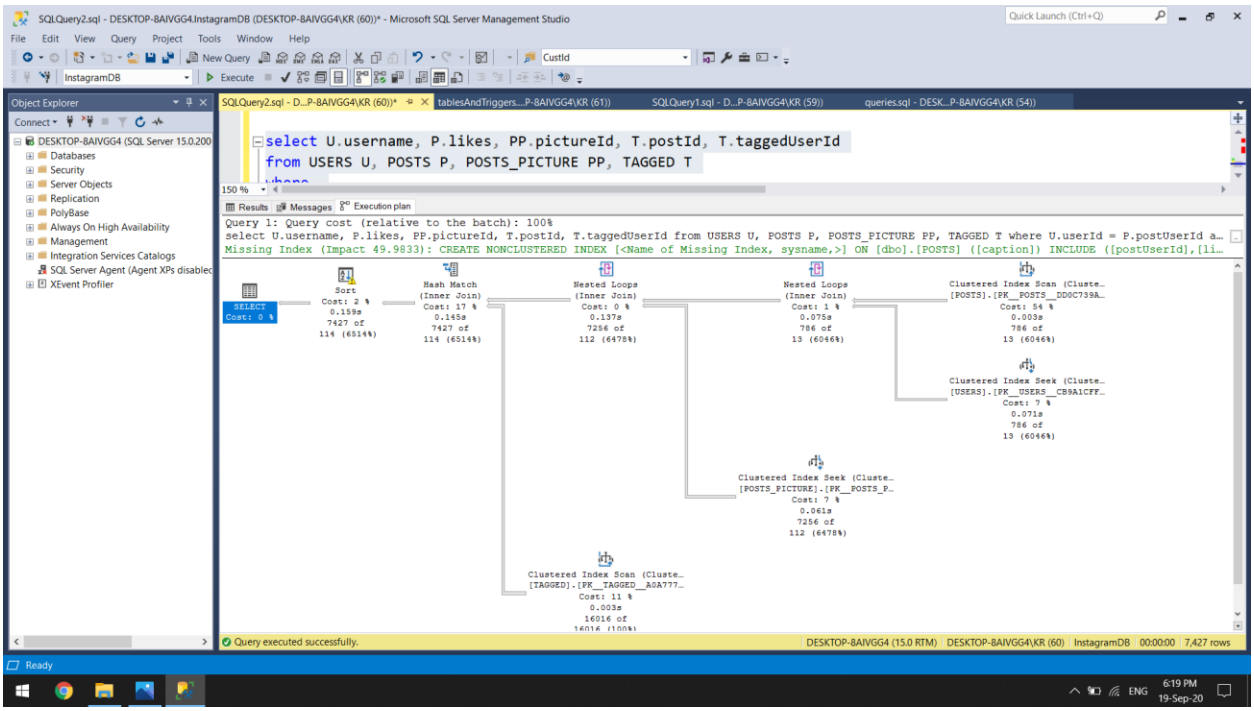
The query here has a condition on caption from POSTS table.



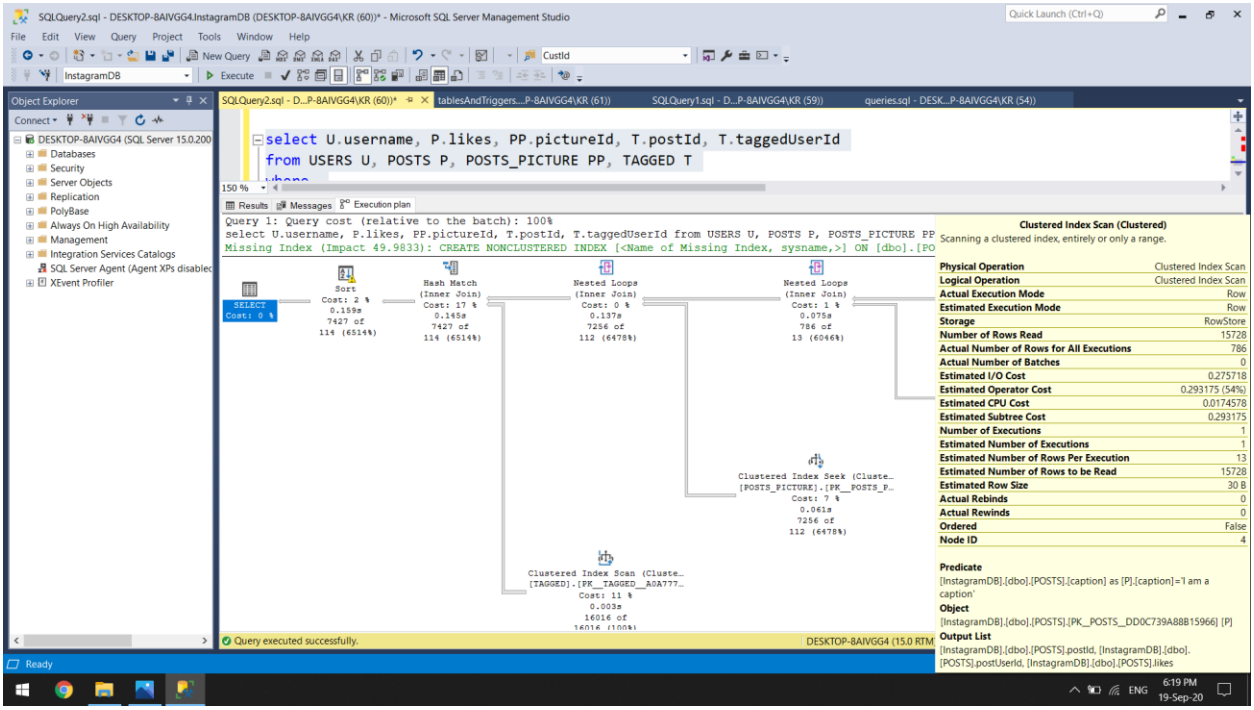
The message output-

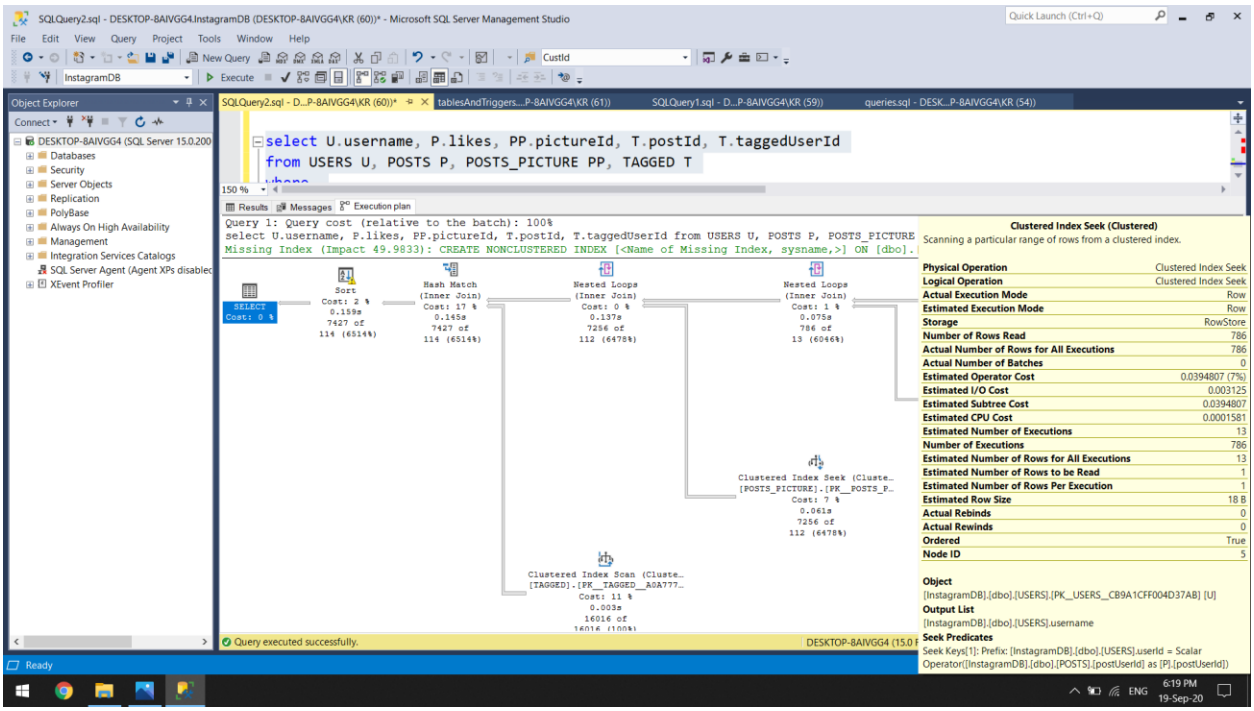


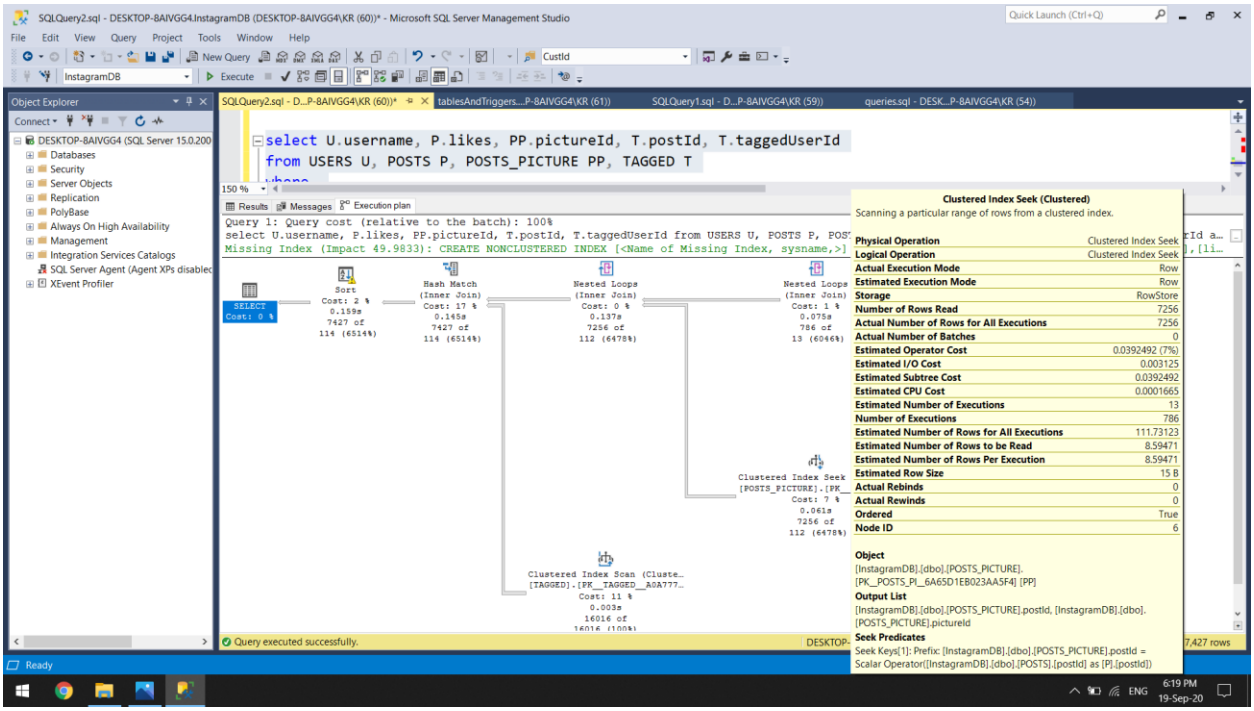
The execution plan shows that the tables are chosen in the order POSTS, USERS, POSTS_PICTURE, TAGGED. Notice the suggestion to create a non-clustered index on caption from POSTS table.

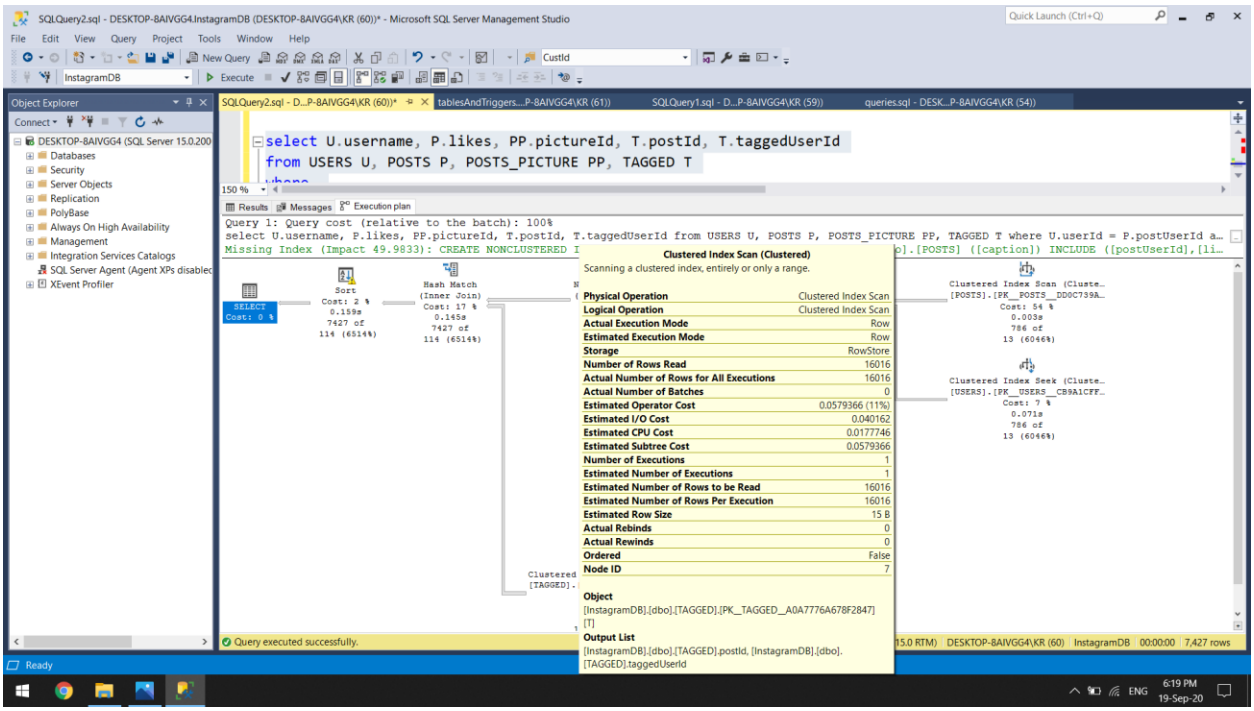


The detailed execution plan-

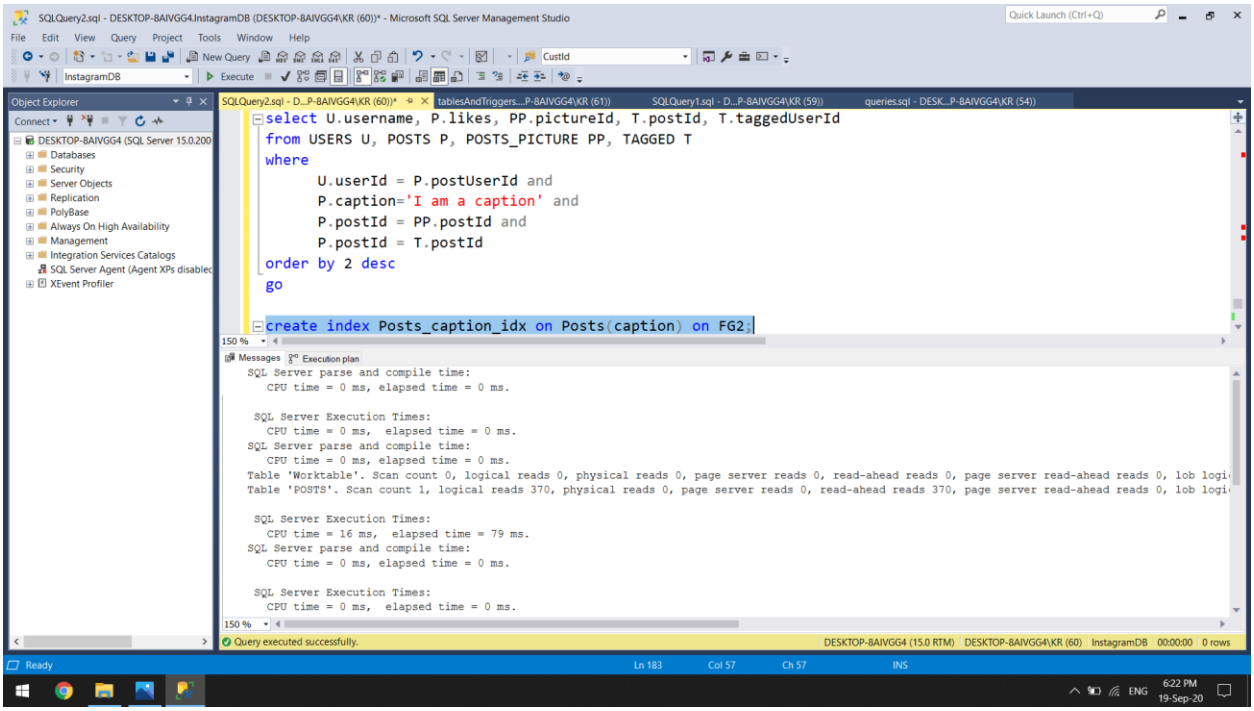



SQLQuery2.sql - DESKTOP-8AIVGG4.InstagramDB (DESKTOP-8AIVGG4\KR (60))* - Microsoft SQL Server Management Studio
select U.username, P.likes, PP.pictureId, T.postId, T.taggedUserId
from USERS U, POSTS P, POSTS_PICTURE PP, TAGGED T
Query 1: Query cost (relative to the batch): 100%
select U.username, P.likes, PP.pictureId, T.postId, T.taggedUserId from USERS U, POSTS P, POSTS_PICTURE
Missing Index (Impact 49.9833): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [dbo].
Clustered Index Seek (Clustered)
Scanning a particular range of rows from a clustered index.
Physical Operation Clustered Index Seek
Logical Operation Clustered Index Seek
Actual Execution Mode Row
Estimated Execution Mode Row
Storage RowStore
Number of Rows Read 786
Actual Number of Rows for All Executions 786
Actual Number of Batches 0
Estimated Operator Cost 0.0394807 (7%)
Estimated I/O Cost 0.003125
Estimated Subtree Cost 0.0394807
Estimated CPU Cost 0.0001581
Estimated Number of Executions 13
Number of Executions 786
Estimated Number of Rows for All Executions 13
Estimated Number of Rows to be Read 1
Estimated Number of Rows Per Execution 1
Estimated Row Size 18 B
Actual Rebinds 0
Actual Rewinds 0
Ordered True
Node ID 5
Object
[InstagramDB].[dbo].[USERS].[PK__USERS__CB9A1CFF004D37AB] [U]
Output List
[InstagramDB].[dbo].[USERS].username
Seek Predicates
Seek Keys[1]: Prefix: [InstagramDB].[dbo].[USERS].userId = Scalar Operator([InstagramDB].[dbo].[POSTS].[postUserId] as [P].[postUserId])
Query executed successfully.
Ready

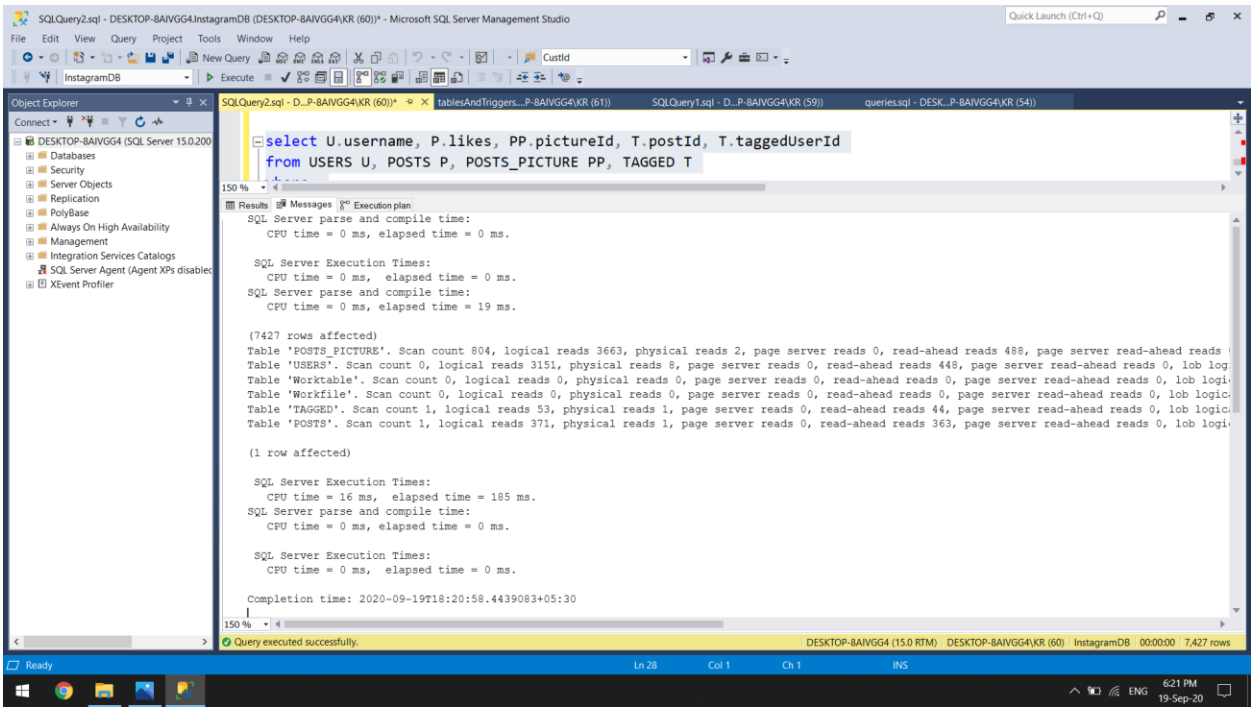

SQLQuery2.sql - DESKTOP-8AIVGG4.InstagramDB (DESKTOP-8AIVGG4\KR (60))* - Microsoft SQL Server Management Studio
select U.username, P.likes, PP.pictureId, T.postId, T.taggedUserId
from USERS U, POSTS P, POSTS_PICTURE PP, TAGGED T
Query 1: Query cost (relative to the batch): 100%
Missing Index (Impact 49.9833): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>]
Clustered Index Seek (Clustered)
Scanning a particular range of rows from a clustered index.
Physical Operation Clustered Index Seek
Logical Operation Clustered Index Seek
Actual Execution Mode Row
Estimated Execution Mode Row
Storage RowStore
Number of Rows Read 7256
Actual Number of Rows for All Executions 7256
Actual Number of Batches 0
Estimated Operator Cost 0.0392492 (7%)
Estimated I/O Cost 0.003125
Estimated Subtree Cost 0.0392492
Estimated CPU Cost 0.0001665
Estimated Number of Executions 13
Number of Executions 786
Estimated Number of Rows for All Executions 111.73123
Estimated Number of Rows to be Read 8.59471
Estimated Number of Rows Per Execution 8.59471
Estimated Row Size 15 B
Actual Rebinds 0
Actual Rewinds 0
Ordered True
Node ID 6
Object
[InstagramDB].[dbo].[POSTS_PICTURE].
[PK__POSTS_PI__6A65D1EB023AA5F4] [PP]
Output List
[InstagramDB].[dbo].[POSTS_PICTURE].postId, [InstagramDB].[dbo].
[POSTS_PICTURE].pictureId
Seek Predicates
Seek Keys[1]: Prefix: [InstagramDB].[dbo].[POSTS_PICTURE].postId =
Scalar Operator([InstagramDB].[dbo].[POSTS].[postId] as [P].[postId])
Query executed successfully.
Ready



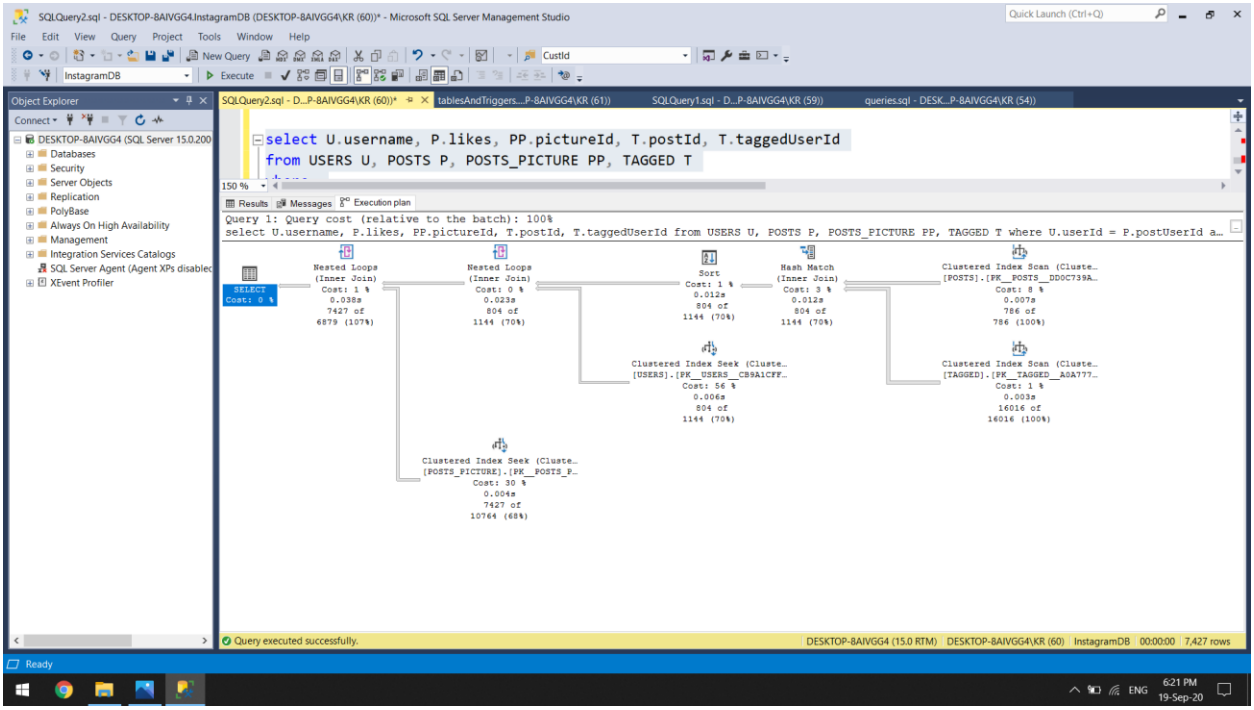
Now we create a non-clustered index on P.caption and check the execution plan.

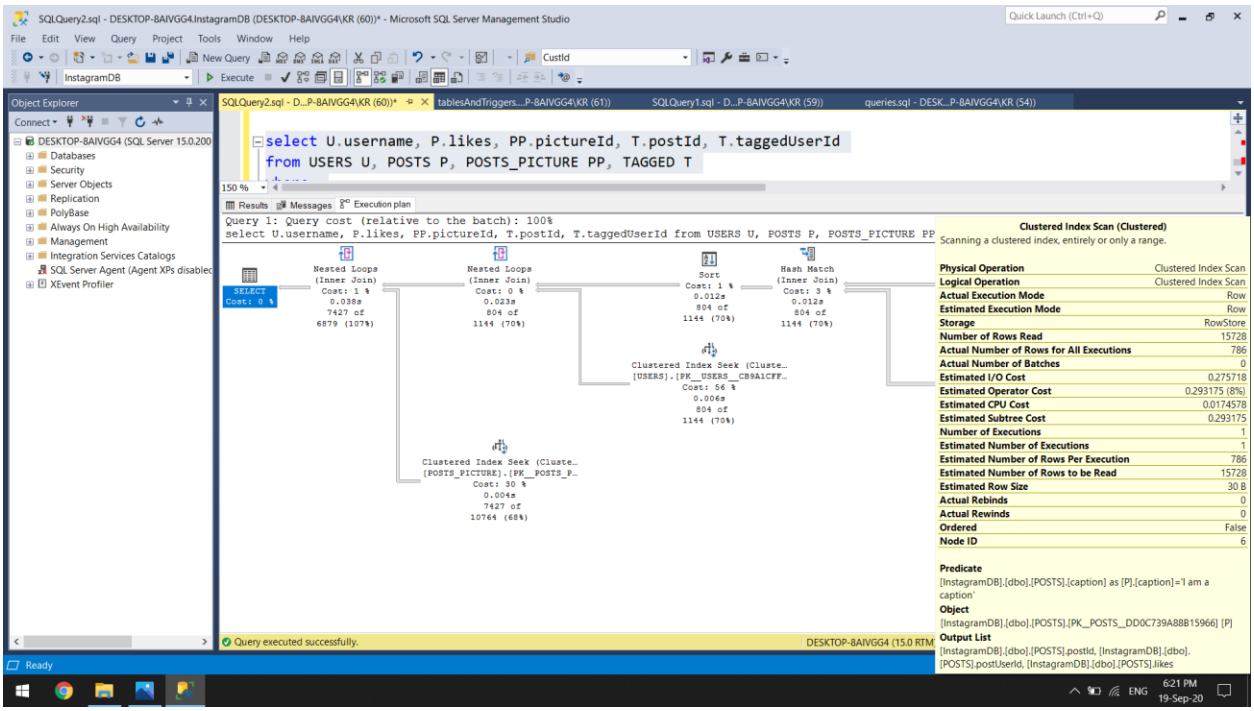


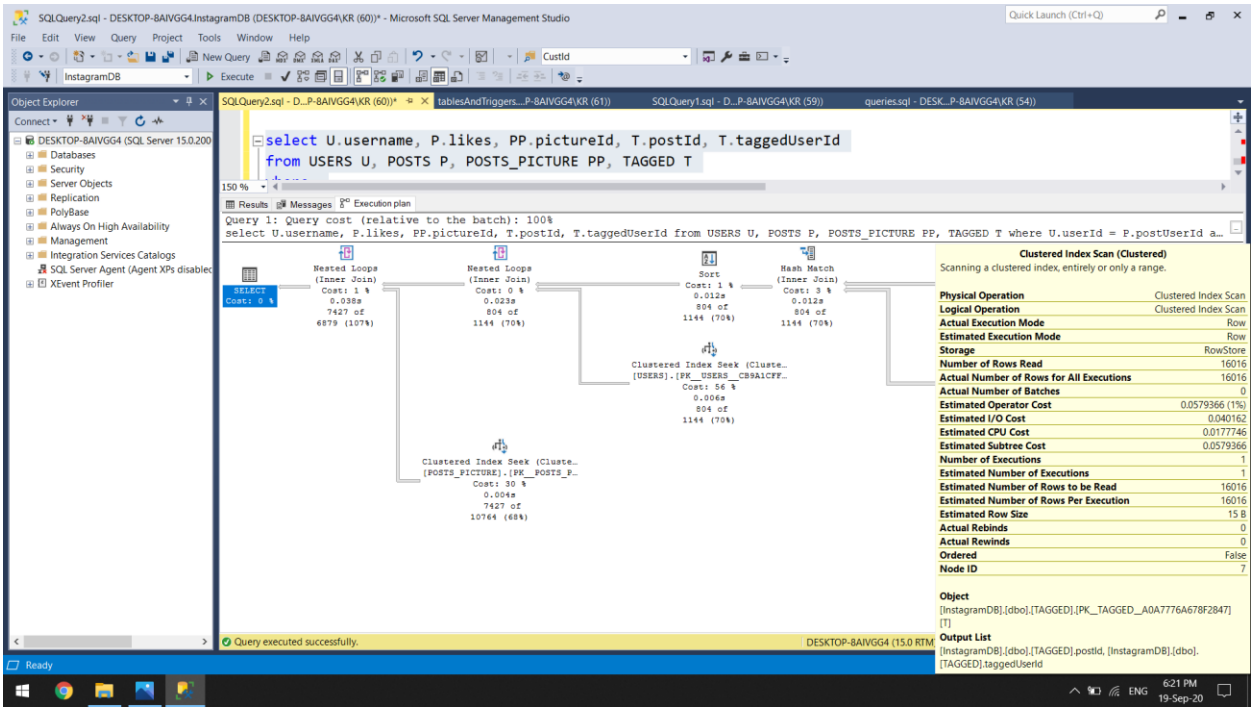
Notice the changes in the execution time- elapsed time was previously 315 ms, it is now 185 ms.

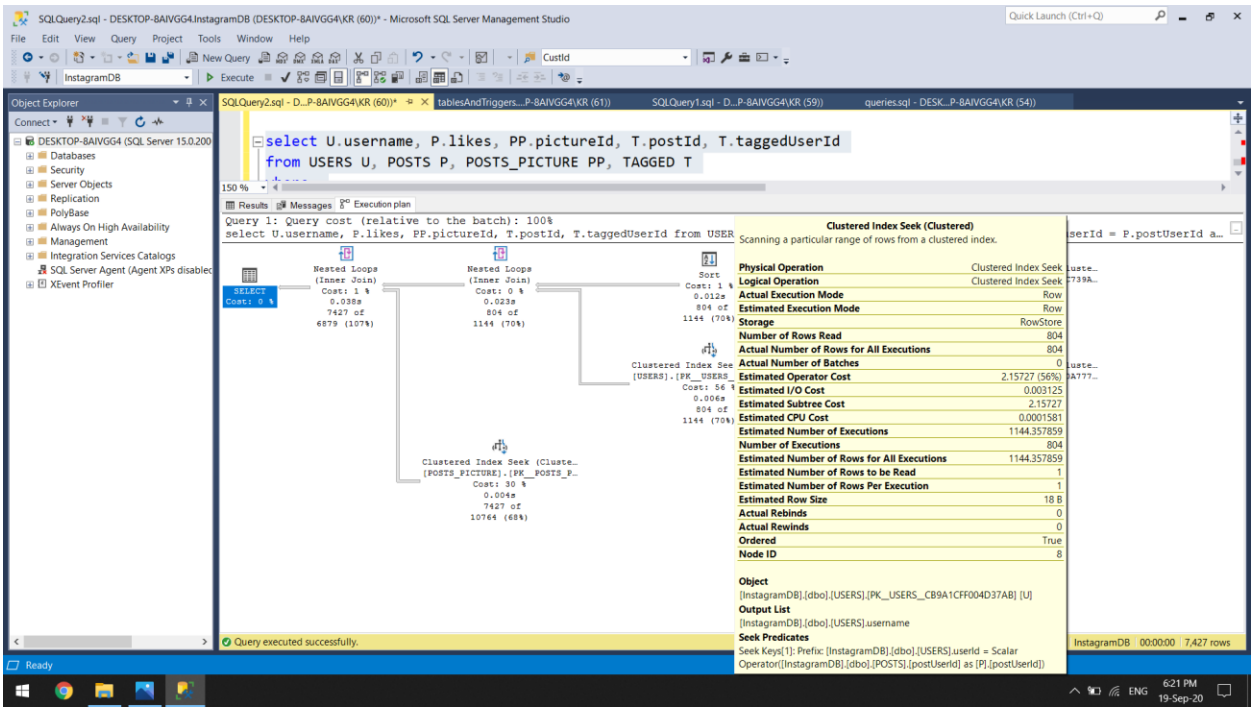


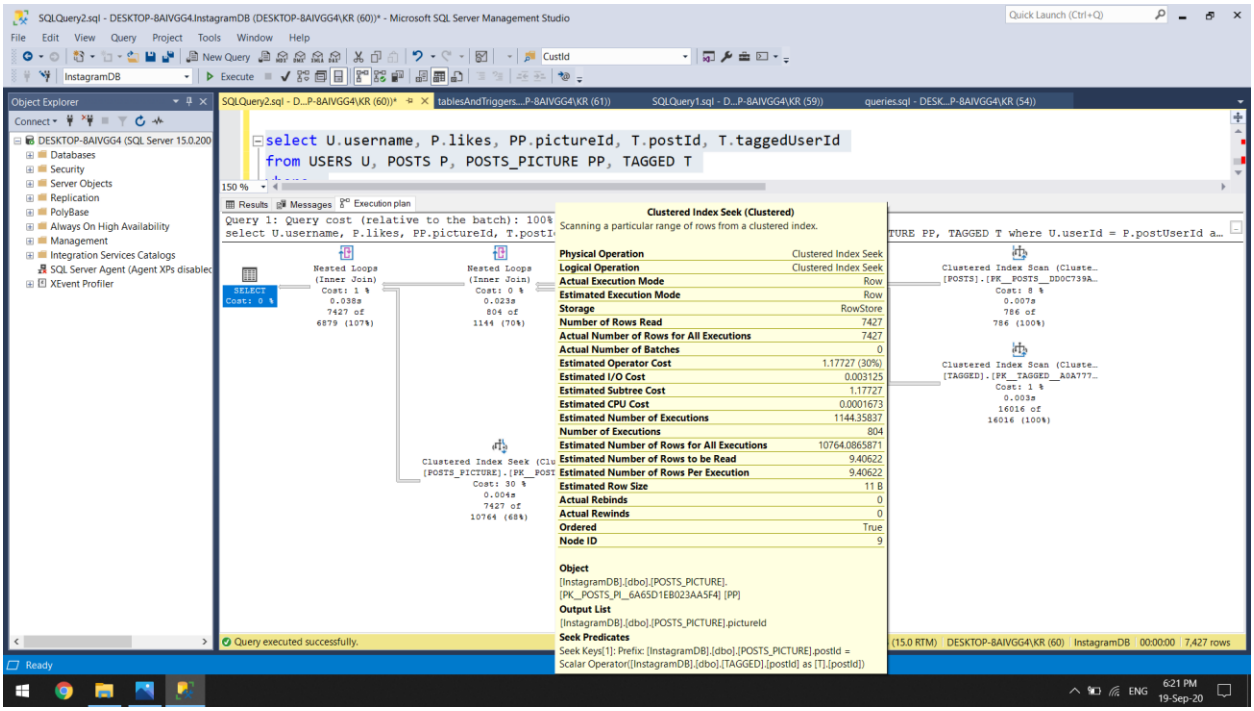
Previously, the cost of clustered index scan on POSTS was 54%. It has now reduced to 0%.




SQLQuery2.sql - DESKTOP-8AIVGG4.InstagramDB (DESKTOP-8AIVGG4\KR (60))* - Microsoft SQL Server Management Studio
select U.username, P.likes, PP.pictureId, T.postId, T.taggedUserId
from USERS U, POSTS P, POSTS_PICTURE PP, TAGGED T
Query 1: Query cost (relative to the batch): 100%
select U.username, P.likes, PP.pictureId, T.postId, T.taggedUserId from USERS U, POSTS P, POSTS_PICTURE PP
Clustered Index Scan (Clustered)
Scanning a clustered index, entirely or only a range.
Physical Operation Clustered Index Scan
Logical Operation Clustered Index Scan
Actual Execution Mode Row
Estimated Execution Mode Row
Storage RowStore
Number of Rows Read 15728
Actual Number of Rows for All Executions 786
Actual Number of Batches 0
Estimated I/O Cost 0.275718
Estimated Operator Cost 0.293175 (8%)
Estimated CPU Cost 0.0174578
Estimated Subtree Cost 0.293175
Number of Executions 1
Estimated Number of Executions 1
Estimated Number of Rows Per Execution 786
Estimated Number of Rows to be Read 15728
Estimated Row Size 30 B
Actual Rebinds 0
Actual Rewinds 0
Ordered False
Node ID 6
Predicate
[InstagramDB].[dbo].[POSTS].[caption] as [P].[caption]='I am a caption'
Object
[InstagramDB].[dbo].[POSTS].[PK__POSTS__DD0C739A88B15966] [P]
Output List
[InstagramDB].[dbo].[POSTS].postId, [InstagramDB].[dbo].[POSTS].postUserId, [InstagramDB].[dbo].[POSTS].likes
Query executed successfully.
Ready


SQLQuery2.sql - DESKTOP-8AIVGG4.InstagramDB (DESKTOP-8AIVGG4\KR (60))* - Microsoft SQL Server Management Studio
select U.username, P.likes, PP.pictureId, T.postId, T.taggedUserId
from USERS U, POSTS P, POSTS_PICTURE PP, TAGGED T
Query 1: Query cost (relative to the batch): 100%
select U.username, P.likes, PP.pictureId, T.postId, T.taggedUserId from USERS U, POSTS P, POSTS_PICTURE PP, TAGGED T where U.userId = P.postUserId a...
Clustered Index Scan (Clustered)
Scanning a clustered index, entirely or only a range.
Physical Operation Clustered Index Scan
Logical Operation Clustered Index Scan
Actual Execution Mode Row
Estimated Execution Mode Row
Storage RowStore
Number of Rows Read 16016
Actual Number of Rows for All Executions 16016
Actual Number of Batches 0
Estimated Operator Cost 0.0579366 (1%)
Estimated I/O Cost 0.040162
Estimated CPU Cost 0.0177746
Estimated Subtree Cost 0.0579366
Number of Executions 1
Estimated Number of Executions 1
Estimated Number of Rows to be Read 16016
Estimated Number of Rows Per Execution 16016
Estimated Row Size 15 B
Actual Rebinds 0
Actual Rewinds 0
Ordered False
Node ID 7
Object
[InstagramDB].[dbo].[TAGGED].[PK__TAGGED__A0A7776A678F2847] [T]
Output List
[InstagramDB].[dbo].[TAGGED].postId, [InstagramDB].[dbo].[TAGGED].taggedUserId
Query executed successfully.
Ready


SQLQuery2.sql - DESKTOP-8AIVGG4.InstagramDB (DESKTOP-8AIVGG4\KR (60))* - Microsoft SQL Server Management Studio
select U.username, P.likes, PP.pictureId, T.postId, T.taggedUserId
from USERS U, POSTS P, POSTS_PICTURE PP, TAGGED T
Clustered Index Seek (Clustered)
Scanning a particular range of rows from a clustered index.
Physical Operation Clustered Index Seek
Logical Operation Clustered Index Seek
Actual Execution Mode Row
Estimated Execution Mode Row
Storage RowStore
Number of Rows Read 804
Actual Number of Rows for All Executions 804
Actual Number of Batches 0
Estimated Operator Cost 2.15727 (56%)
Estimated I/O Cost 0.003125
Estimated Subtree Cost 2.15727
Estimated CPU Cost 0.0001581
Estimated Number of Executions 1144.357859
Number of Executions 804
Estimated Number of Rows for All Executions 1144.357859
Estimated Number of Rows to be Read 1
Estimated Number of Rows Per Execution 1
Estimated Row Size 18 B
Actual Rebinds 0
Actual Rewinds 0
Ordered True
Node ID 8
Object
[InstagramDB].[dbo].[USERS].[PK__USERS__CB9A1CFF004D37AB] [U]
Output List
[InstagramDB].[dbo].[USERS].username
Seek Predicates
Seek Keys[1]: Prefix: [InstagramDB].[dbo].[USERS].userId = Scalar Operator([InstagramDB].[dbo].[POSTS].[postUserId] as [P].[postUserId])
Query executed successfully.
Ready


SQLQuery2.sql - DESKTOP-8AIVGG4.InstagramDB (DESKTOP-8AIVGG4\KR (60))* - Microsoft SQL Server Management Studio
select U.username, P.likes, PP.pictureId, T.postId, T.taggedUserId
from USERS U, POSTS P, POSTS_PICTURE PP, TAGGED T
Clustered Index Seek (Clustered)
Scanning a particular range of rows from a clustered index.
Physical Operation Clustered Index Seek
Logical Operation Clustered Index Seek
Actual Execution Mode Row
Estimated Execution Mode Row
Storage RowStore
Number of Rows Read 7427
Actual Number of Rows for All Executions 7427
Actual Number of Batches 0
Estimated Operator Cost 1.17727 (30%)
Estimated I/O Cost 0.003125
Estimated Subtree Cost 1.17727
Estimated CPU Cost 0.0001673
Estimated Number of Executions 1144.35837
Number of Executions 804
Estimated Number of Rows for All Executions 10764.0865871
Estimated Number of Rows to be Read 9.40622
Estimated Number of Rows Per Execution 9.40622
Estimated Row Size 11 B
Actual Rebinds 0
Actual Rewinds 0
Ordered True
Node ID 9
Object
[InstagramDB].[dbo].[POSTS_PICTURE].
[PK__POSTS_PI__6A65D1EB023AA5F4] [PP]
Output List
[InstagramDB].[dbo].[POSTS_PICTURE].pictureId
Seek Predicates
Seek Keys[1]: Prefix: [InstagramDB].[dbo].[POSTS_PICTURE].postId = Scalar Operator([InstagramDB].[dbo].[TAGGED].[postId] as [T].[postId])
Query executed successfully.
Ready