Instagram Database Management

Problem Statement:

Instagram database needs to store information about its users(identified by username, name, email, password, bio and privacy levels as attributes), their followers and the users they follow, posts(identified by postId, caption, location, time, comment disabler as attributes, number of likes, and number of comments), likes on each post, comments(identified by the photo, user who commented, time and the comment as attributes) and finally, data of users that tagged in a post.

Database Schema:

Tables from DBMS project:

1. **USERS**(userId(Integer) P.K, username(string_unique), firstName(string), lastName(string), emailId(string), bio(string), password(string), privacyLevel(Bit))
2. **FOLLOWER_FOLLOWING**(followerUserId(Integer), followingUserId(Integer))
3. **REQUEST**(receivedUserId(Integer), sentUserId(Integer), status(Bit))
4. **BLOCKED**(blockingUserId(Integer), blockedUserId(Integer))
5. **POSTS**(postId(Integer) P.K, postUserId(Integer), caption(string), latitude(Integer), longitude(Integer), timePost(Timestamp), disableComment(Bit), likes(Integer), comments(Int))
6. **POSTS_PICTURE**(postId(Integer), pictureId(Integer))
7. **LIKES**(postId(Integer), likedUserId(Integer))
8. **COMMENTS**((commentId(Integer), postId(Integer)), commentUserId(Integer), timeComment(timeStamp), comment(string))
9. **TAGGED**(postId(Integer), taggedUserId(Integer))

Tables implemented now:

1. **USERS**(userId(Integer) P.K, username(string_unique), firstName(string), lastName(string), emailId(string), bio(string), password(string), privacyLevel(Bit))
2. **POSTS**(postId(Integer) P.K, postUserId(Integer), caption(string), latitude(Integer), longitude(Integer), timePost(Timestamp), disableComment(Bit), likes(Integer), comments(Int))

Functional Dependencies:

1. userId → userId, username, firstName, lastName, emailId, bio, password, privacyLevel
2. postId → postId, postUserId, caption, latitude, longitude, disableComment, timePost, likes, comments
3. {postId, pictureId} → postId, pictureId
4. {postId, likedUserId} → postId, likedUserId

1NF: Cannot have multivalued attributes in a table
2NF: Cannot have partial functional dependencies
3NF: Cannot have transitive dependencies

Normalisation and testing for lossless join property:

- **POSTS** initially had the attributes *postId, pictureId caption, location, timestamp, likes, comments and disableComment.*
    - Location is a multivalued attribute containing latitude and longitude. Therefore, following **1NF**(Cannot have multivalued attributes in a table), the table was made into:
      POSTS*(postId, pictureId, caption, latitude, longitude, timestamp, disableComment)*
    - Since there were several *pictureIds* for each *postId* and the rest of the attributes did not depend on individual pictureId, to reduce redundancy, following **1NF**, the table was further split into:
      POSTS*(postId, caption, latitude, longitude, timestamp, likes, comments disableComment)*
      And ***POSTS_PICTURE****(postId, pictureId)*
- If the candidate key in **POSTS** consisted of *postId and userId,* then it would violate **2NF** because *postId → caption, latitude, longitude, timestamp, likes, comments and disableComment* but candidate key is *postId, userId.* Therefore, since *postId* is unique on its own, it is the only candidate key, avoiding violation of **2NF.**
- No violations of **3NF** were found as there are no transitive dependencies.
- **Testing for lossless join property:** On splitting ***POSTS****(R)* table into ***POSTS****(R1) and* ***POSTS_PICTURE****(R2),* all three conditions of the lossless join property were satisfied:
    - Att(R1) U Att(R2) = Att(R)
      Union of ***POSTS*** and ***POSTS_PICTURE*** gives back the initial table
    - Att(R1) ∩ Att(R2) ≠ Φ
      Att(R1) ∩ Att(R2) = *postId*
    - Att(R1) ∩ Att(R2) -> Att(R1) *or* Att(R1) ∩ Att(R2) -> Att(R2)
      Common attribute is key in both relations

[Test for lossless join attached in excel sheet]

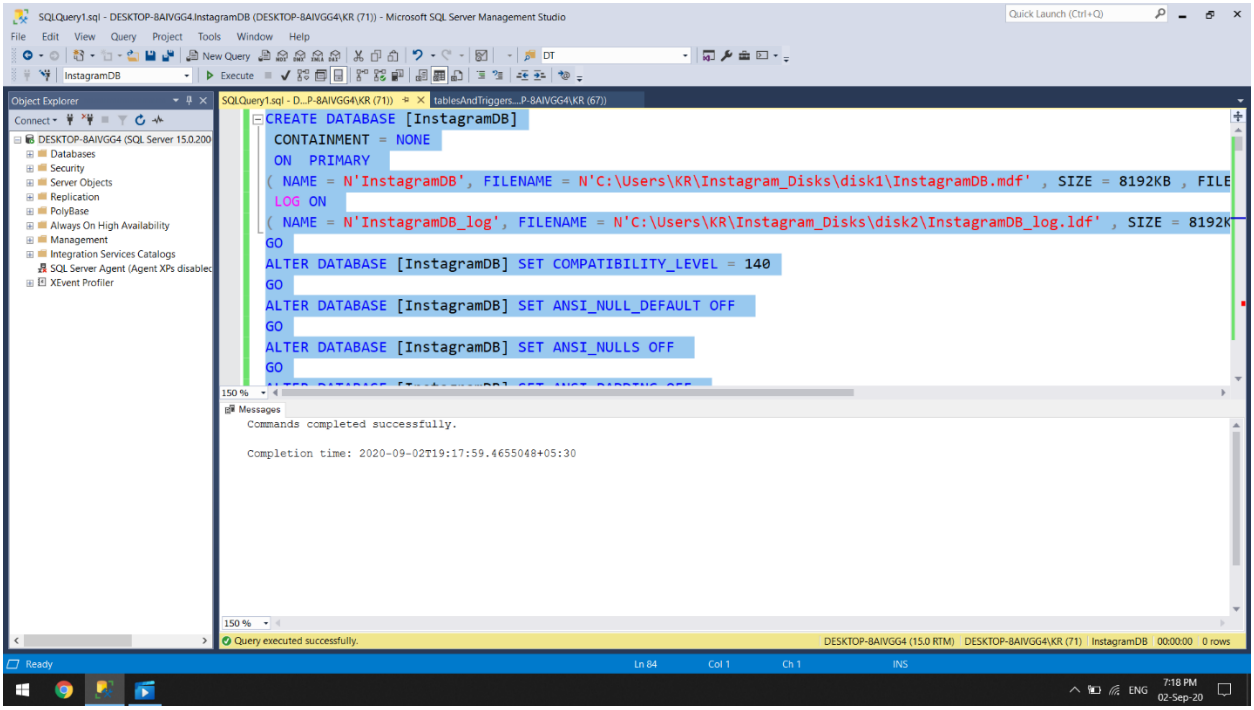
Case 1:

Single table without partition, created one non-clustered index:

1. Created two folders called disk1 and disk2 representing two raid volumes.

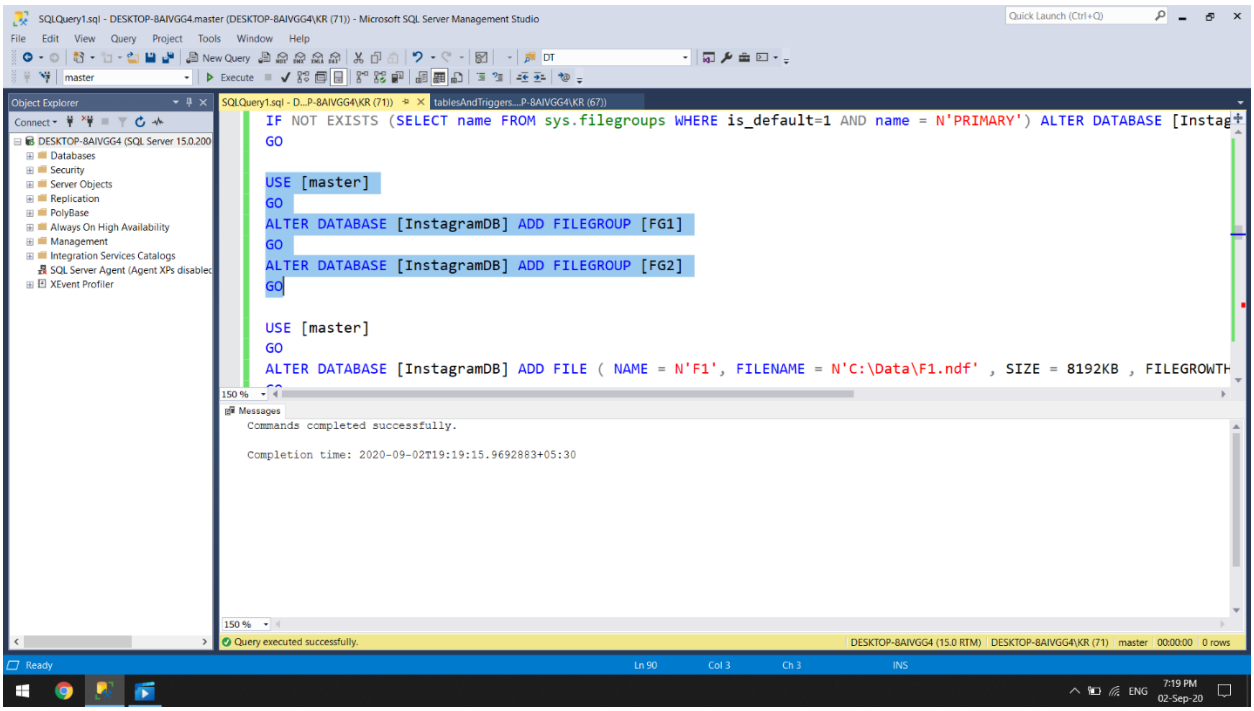
2. Create Database called InstagramDB.

SQL Server databases use two files: an MDF file(main data file), known as the primary database file, which contains the schema and data, and a LDF(log data file), which contains the logs.



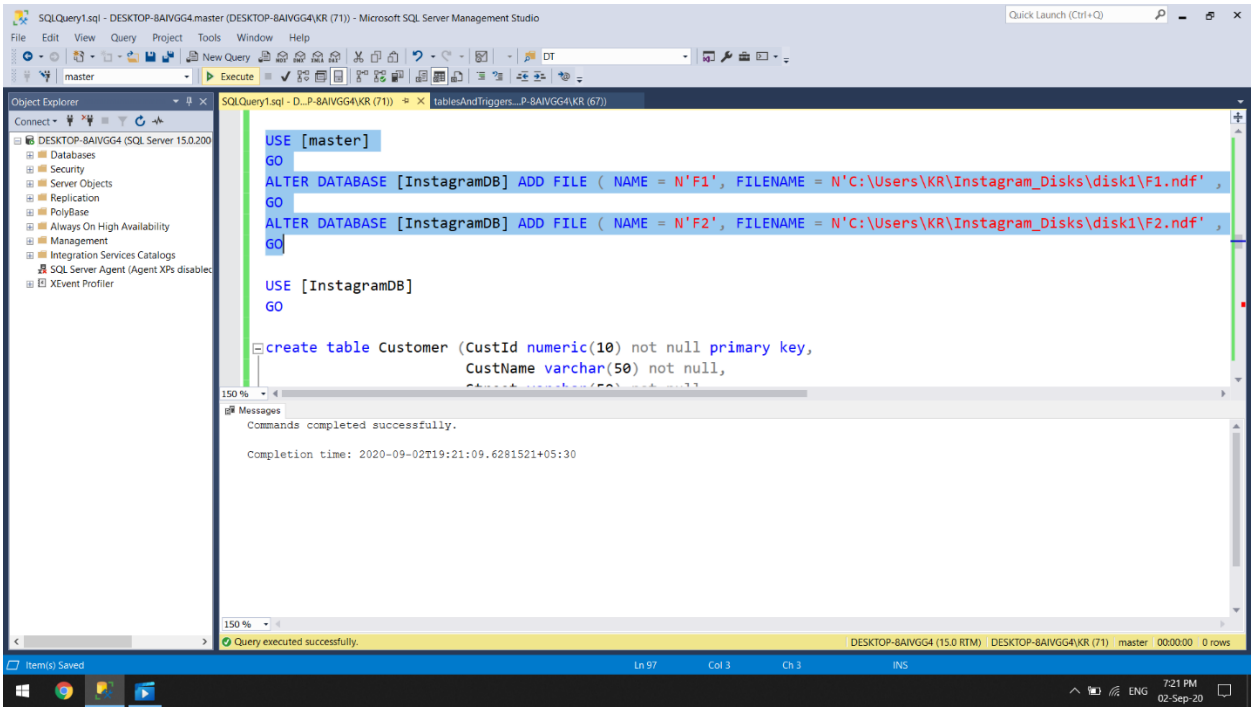
3. Create two filegroups FG1 and FG2

- A filegroup is a group of files
- We can add files to a filegroup across all the available logical volumes(disks) of different raid levels.

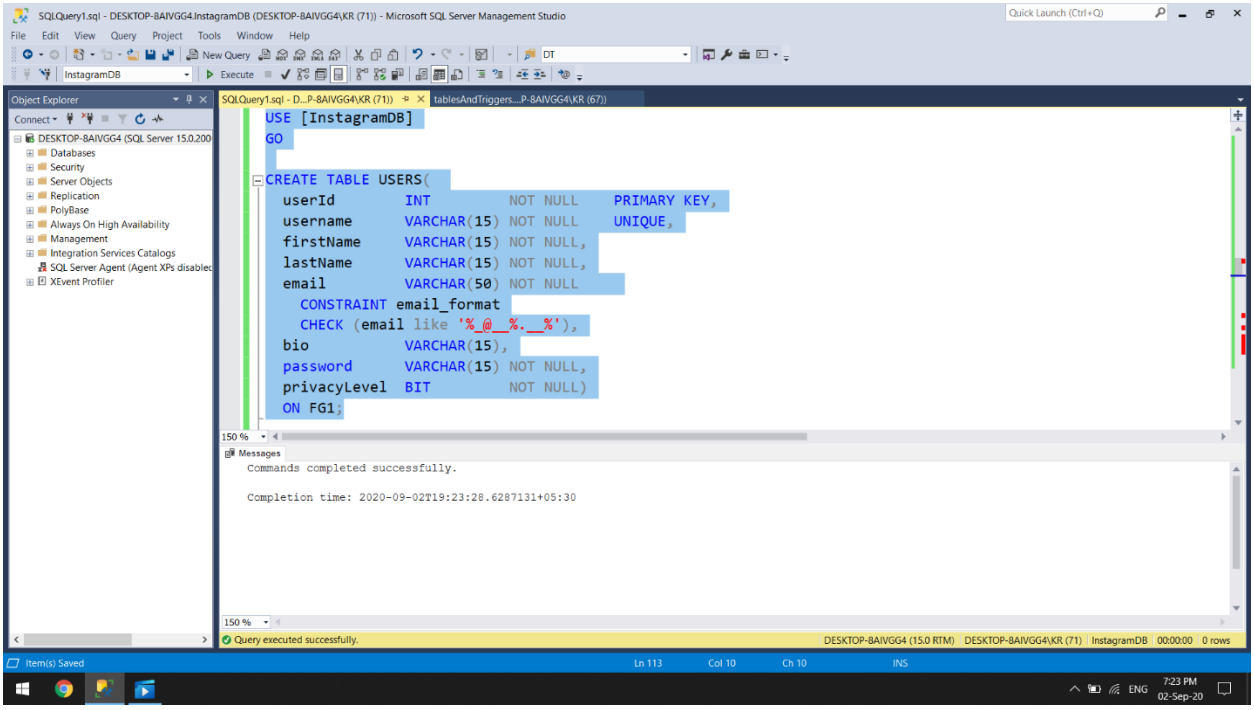


4. Add files F1 to filegroup FG1 and F2 to filegroup FG2, respectively.

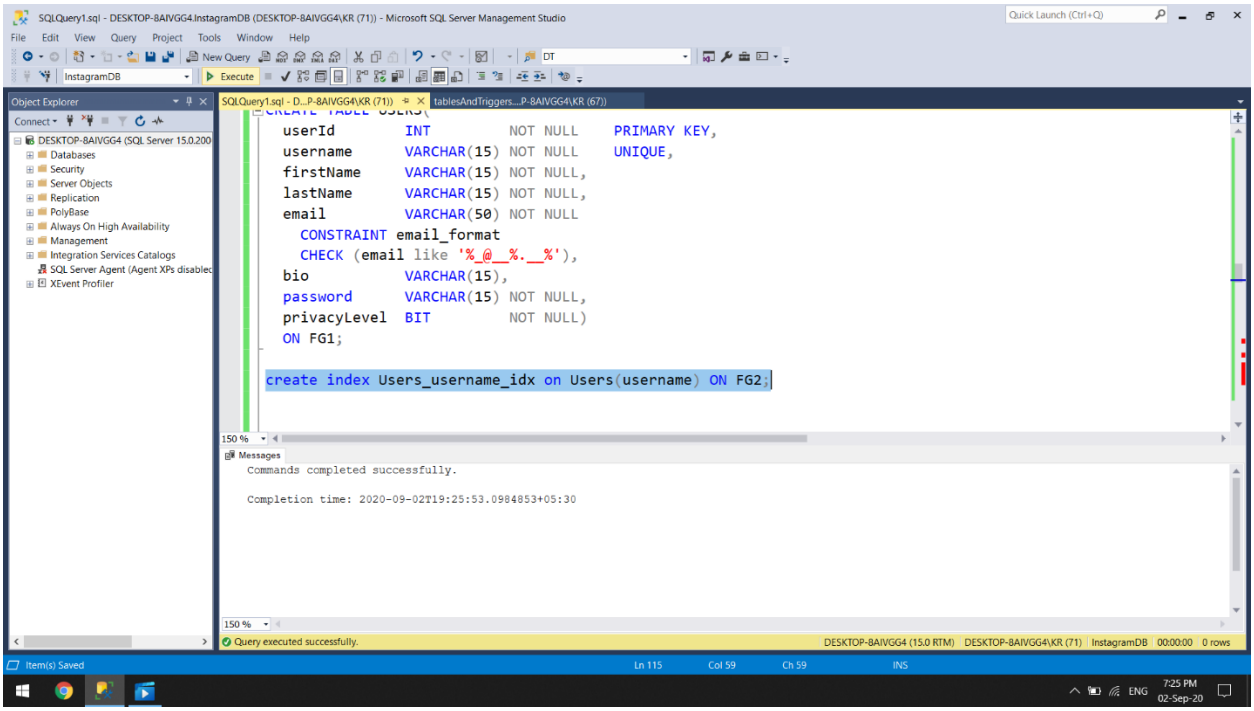
- A database may also use secondary database file(next database file), which normally uses a .ndf extension.



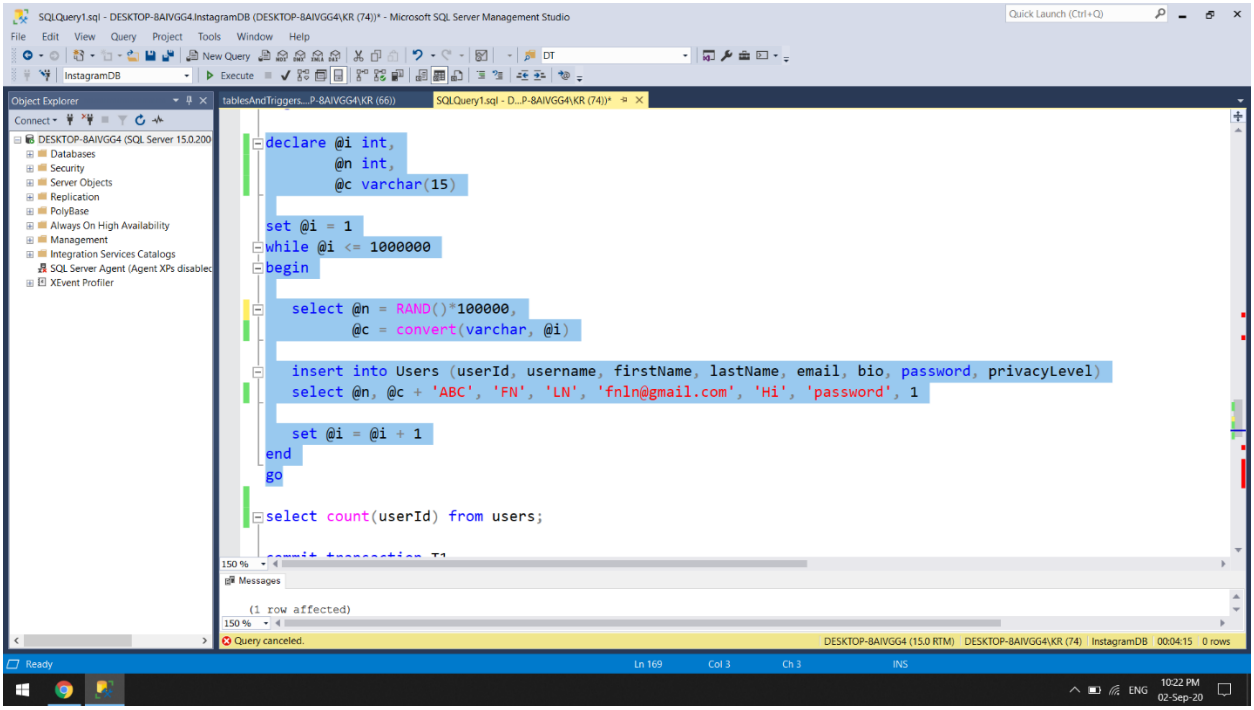
5. Create table Users on FG1



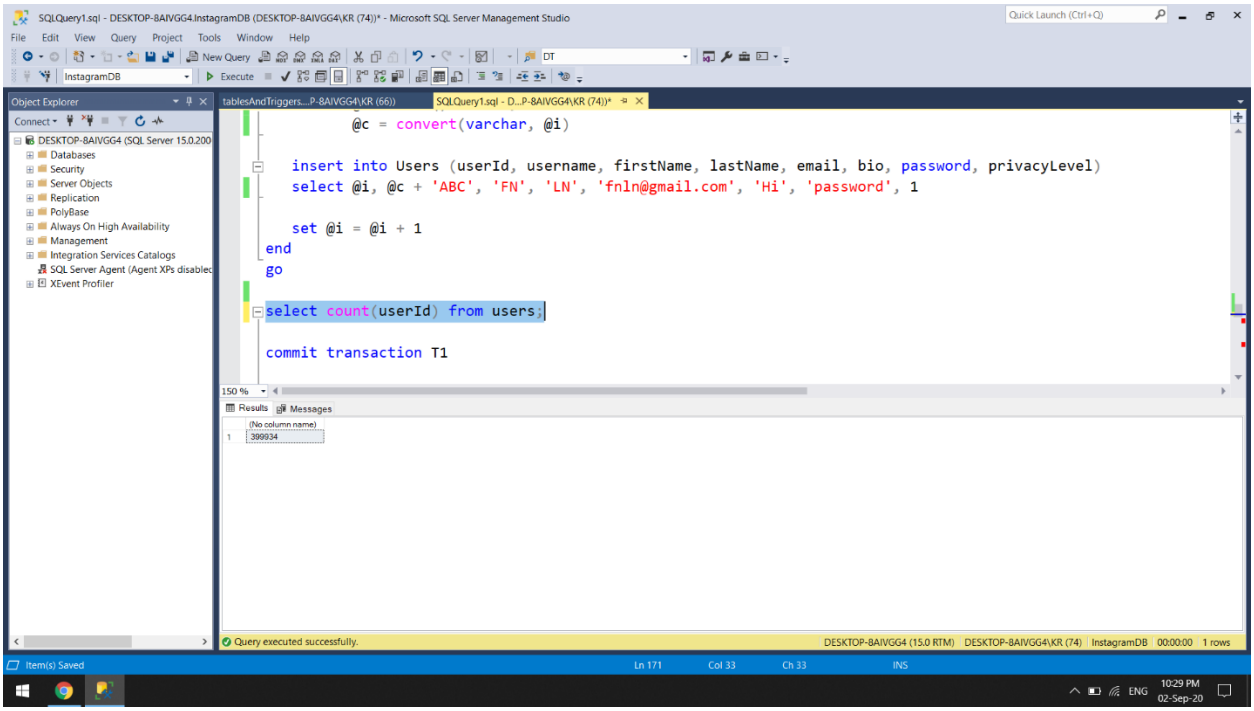
6. Create a nonclustered index called Users_username_idx on Users using attribute username; stored on FG2



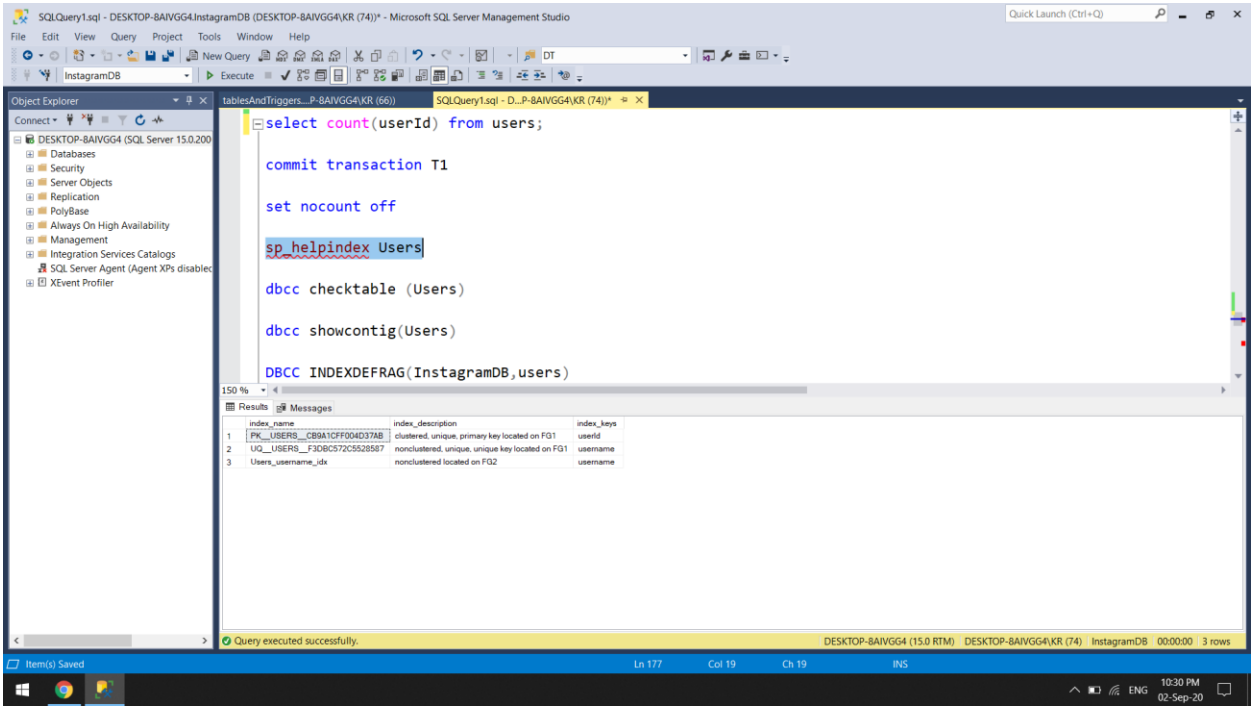
7. Populate the Users table with random userId and username. The rest of the attributes are hardcoded with the same value for convenience at this point. We load thousands of rows like this.



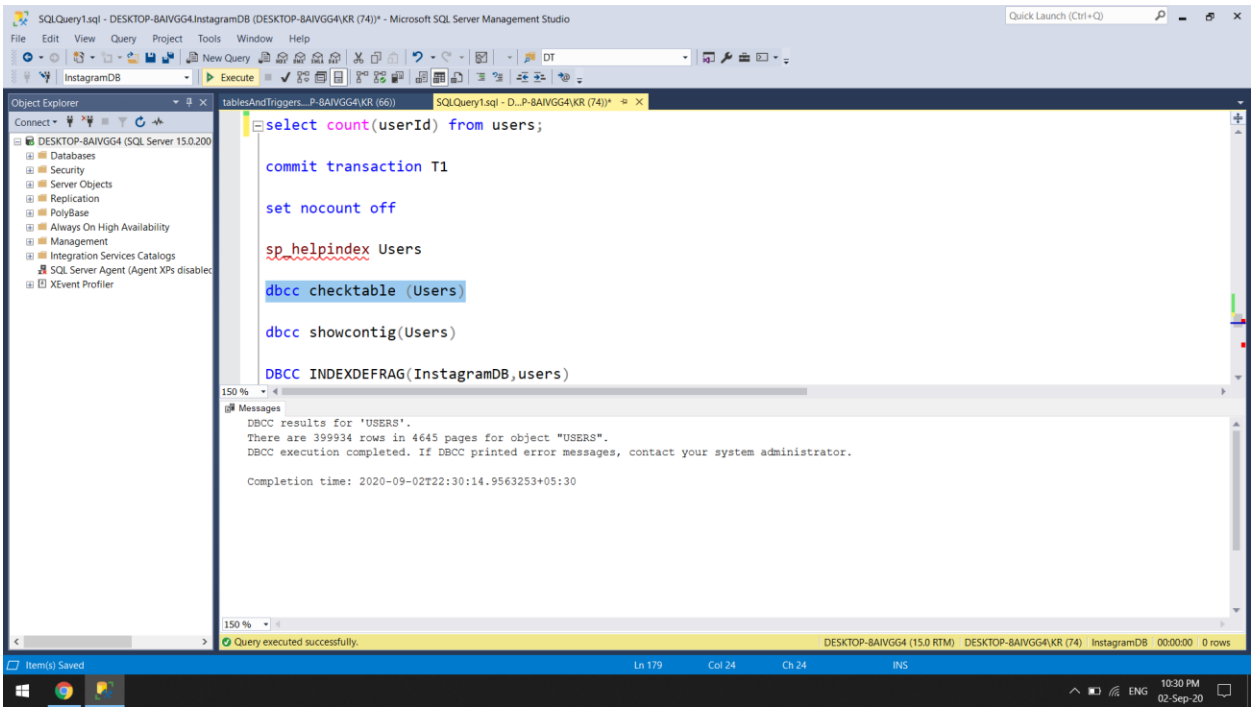
8. We have loaded 399934 rows.



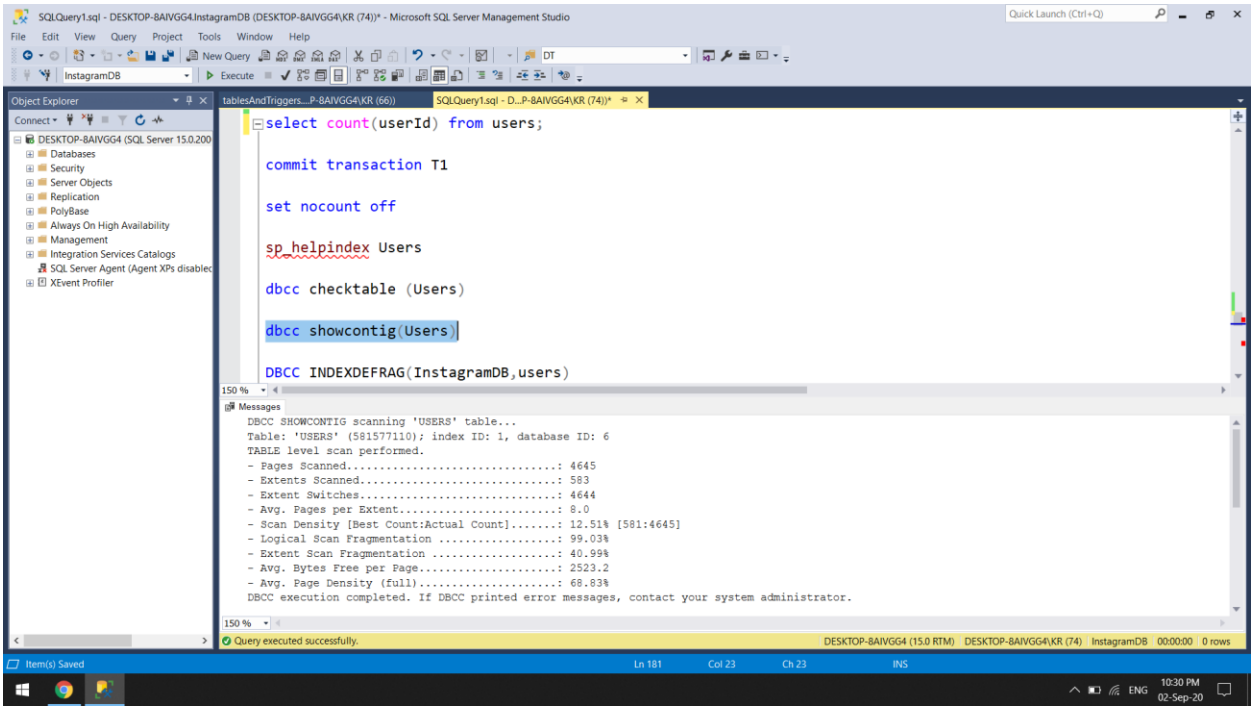
9. sp_helpindex command tells us the index on Users table. We currently have two indexes on FG1 that were created automatically with the creation of the table: a primary key clustered index for the userId key, a non-clustered index for the username key because we have set username to be unique in the schema of the table. The third index is the one that we have manually created now. It is a non-clustered index on FG2 for the username key.



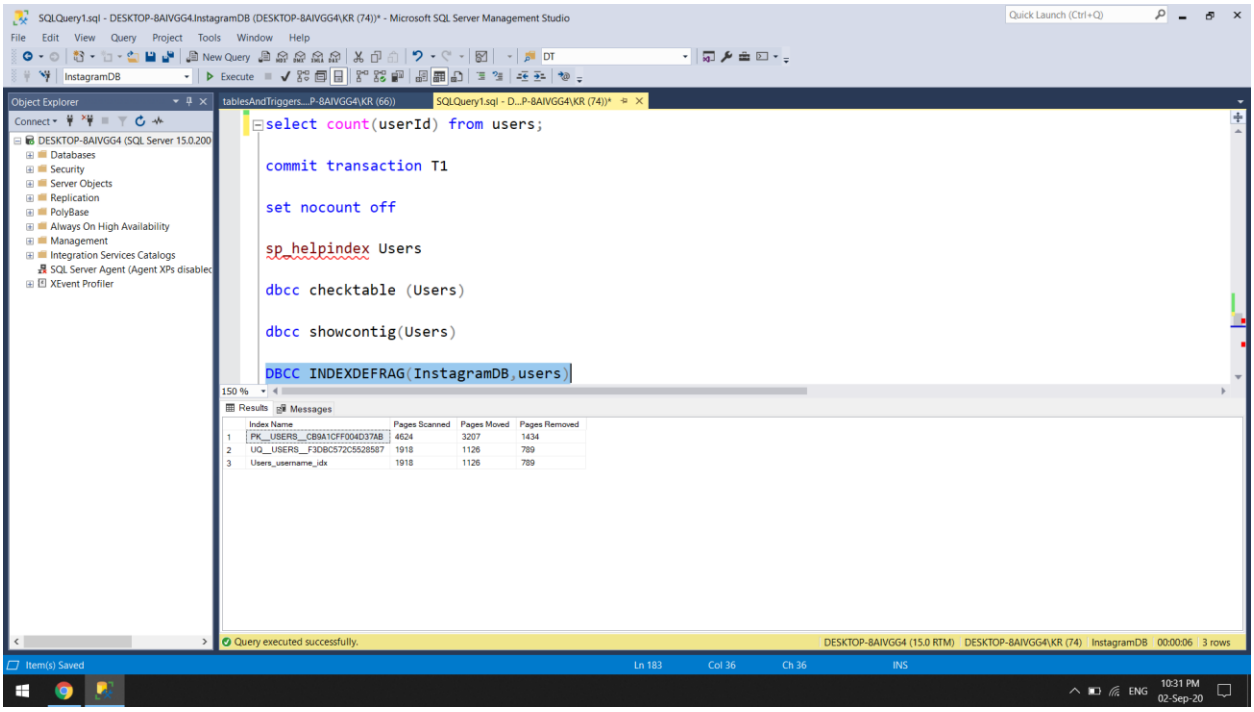
10. We check the number of pages for the rows that we have created. Note that 4645 pages have been allotted for the Users table currently.



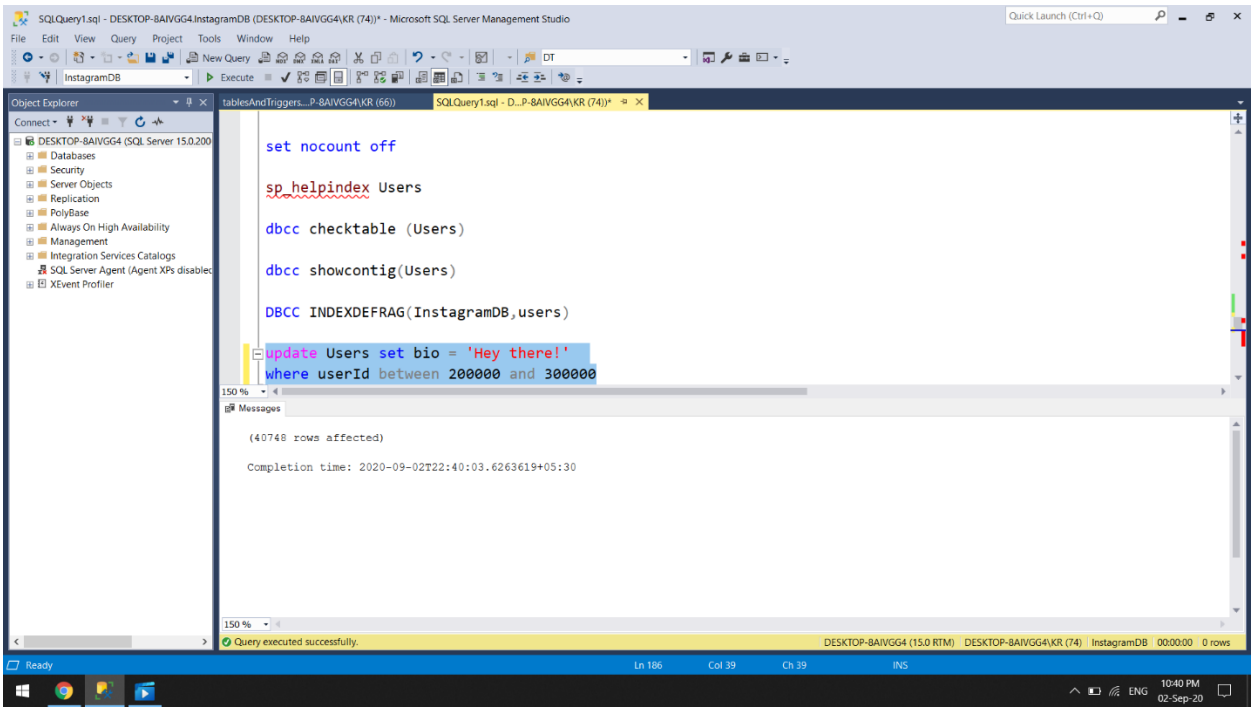
11. We check the average page density, which is currently 68.83%, which means that there is a chance for it to be improved.

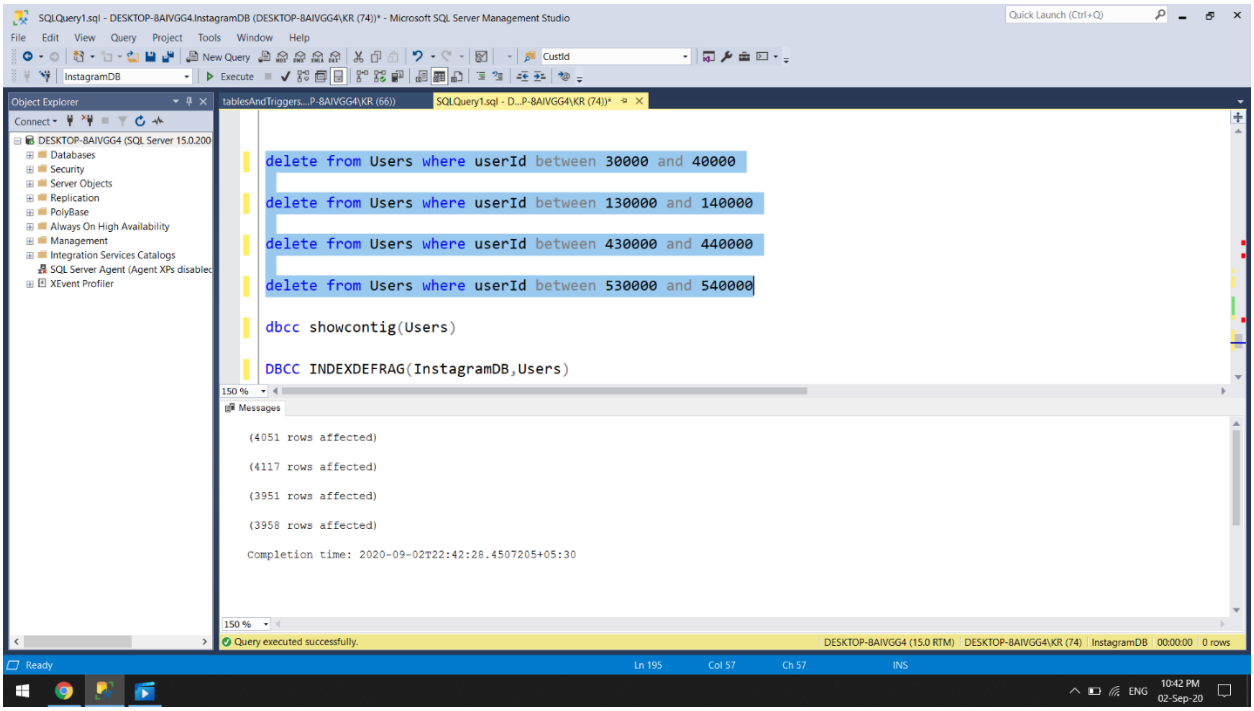


12. We perform index-based defragmentation to improve the average page density and reduce the overall number of pages used for the current number of rows. Notice that several pages are moved and removed in order to achieve a better storage arrangement.

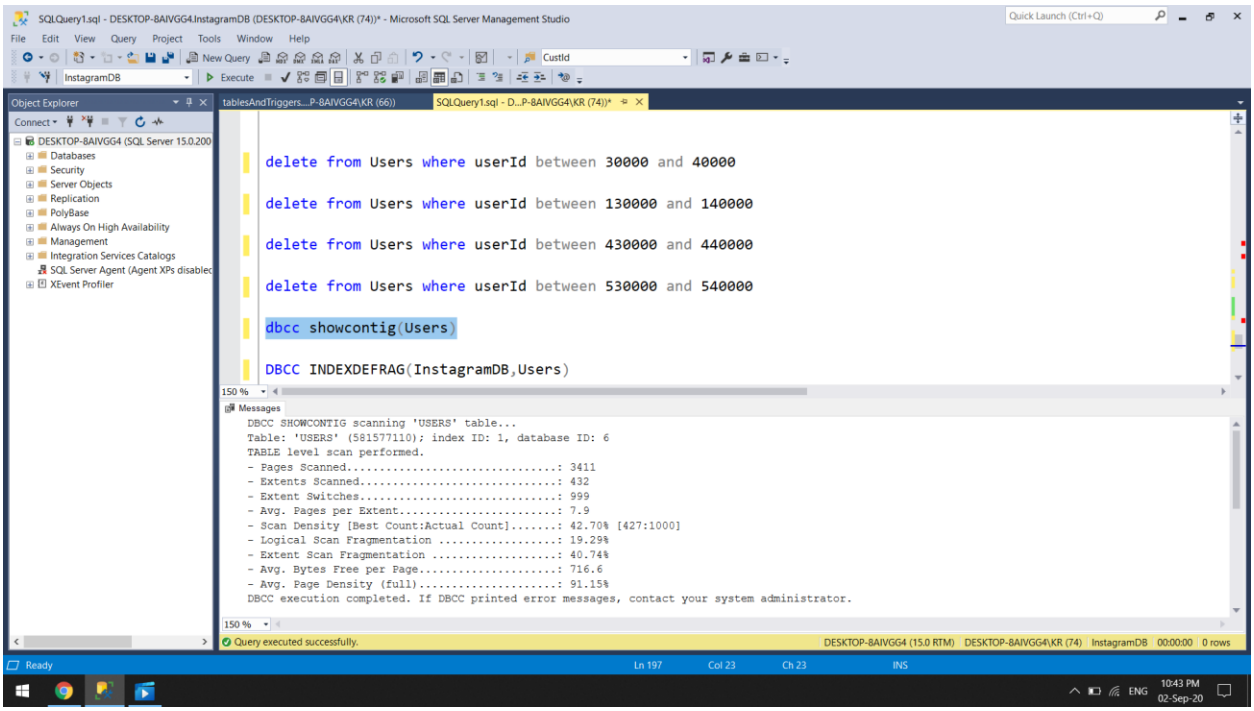


13. We update the bio attribute with a string of characters longer than what it was before. We do this for one lakh tuples. We then delete some tuples to see how the storage on the disk changes.

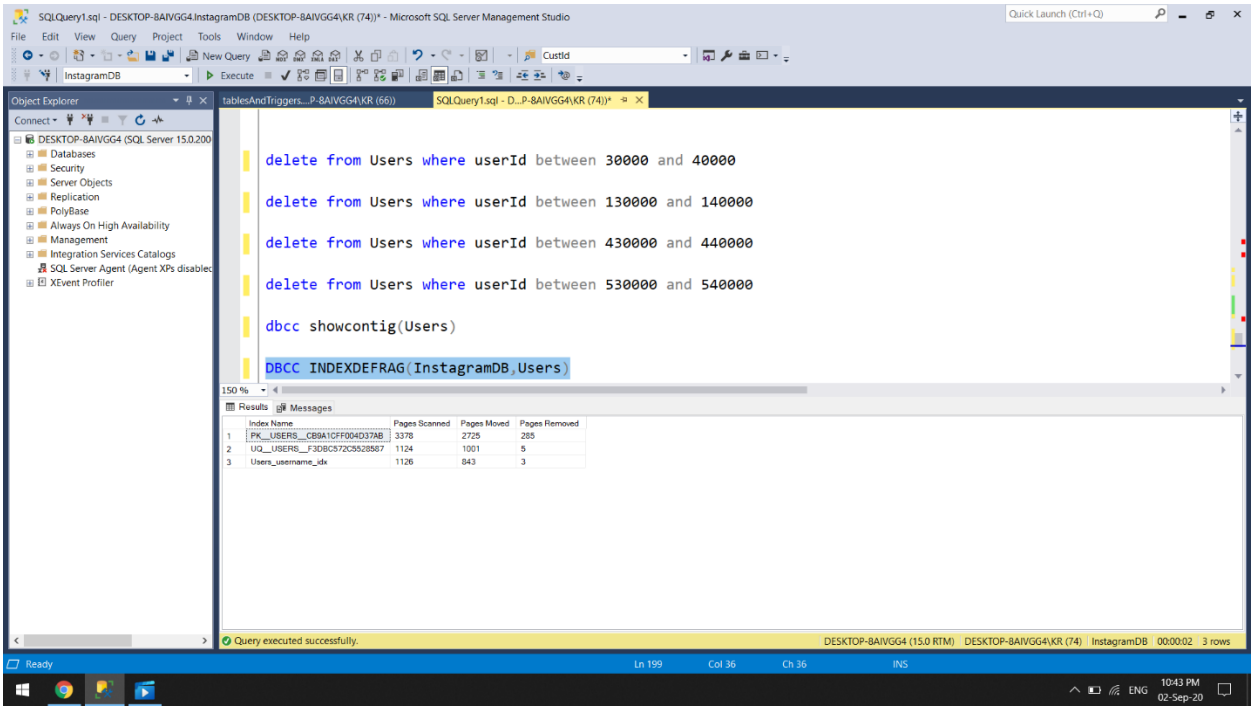




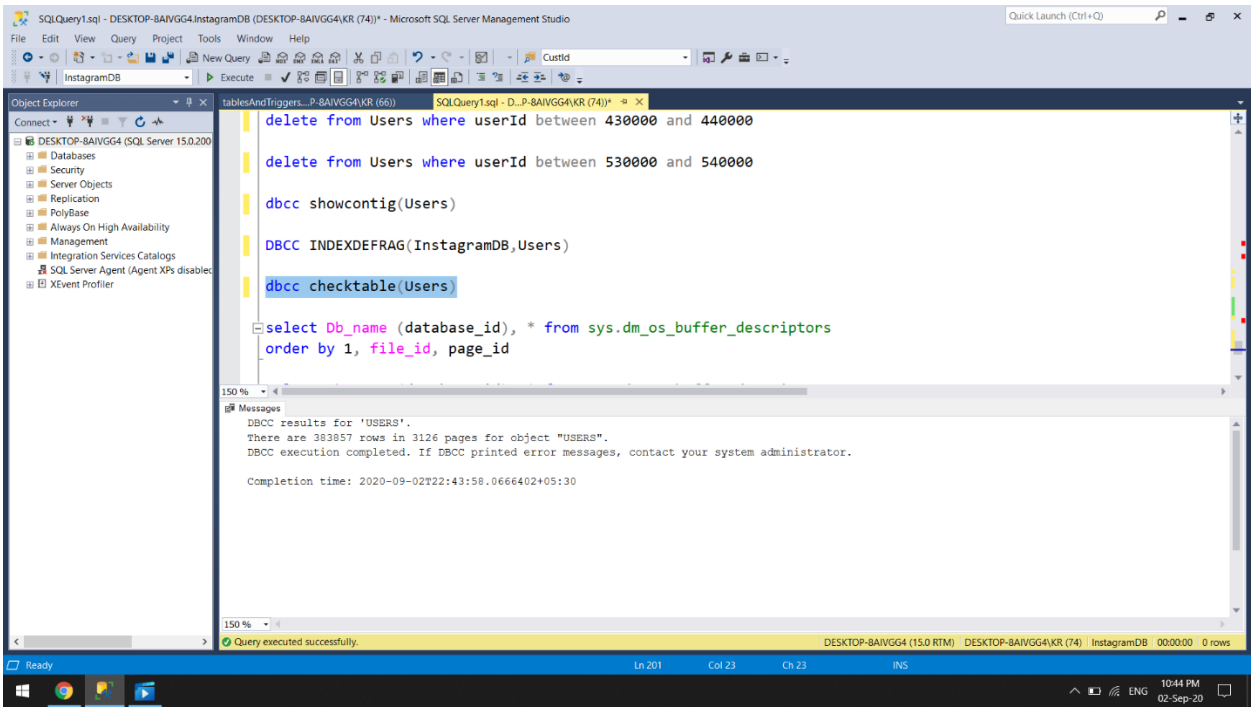
14. We see that the average page density has come down to 91.15%



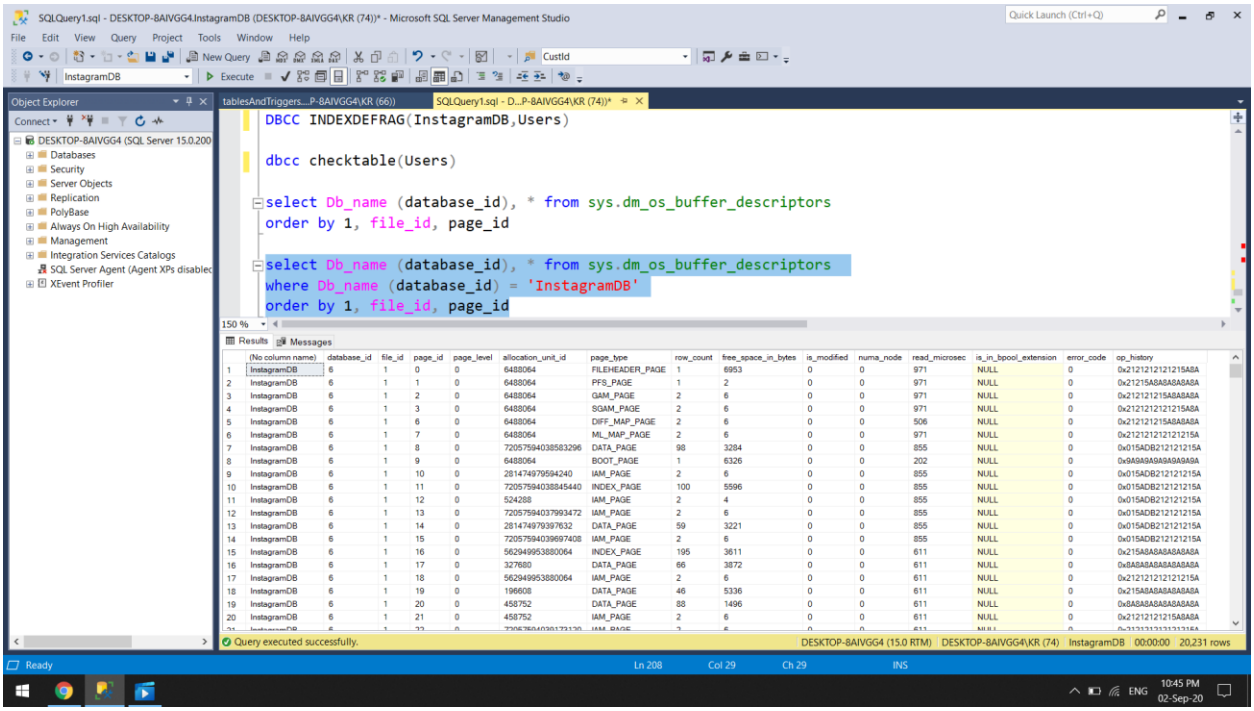
15. Again run INDEXDEFRAG on the Users table. Several pages are moved and removed to improve parameters like the average page density and reduce the number of pages used overall.



16. We see that the number of pages have reduced.



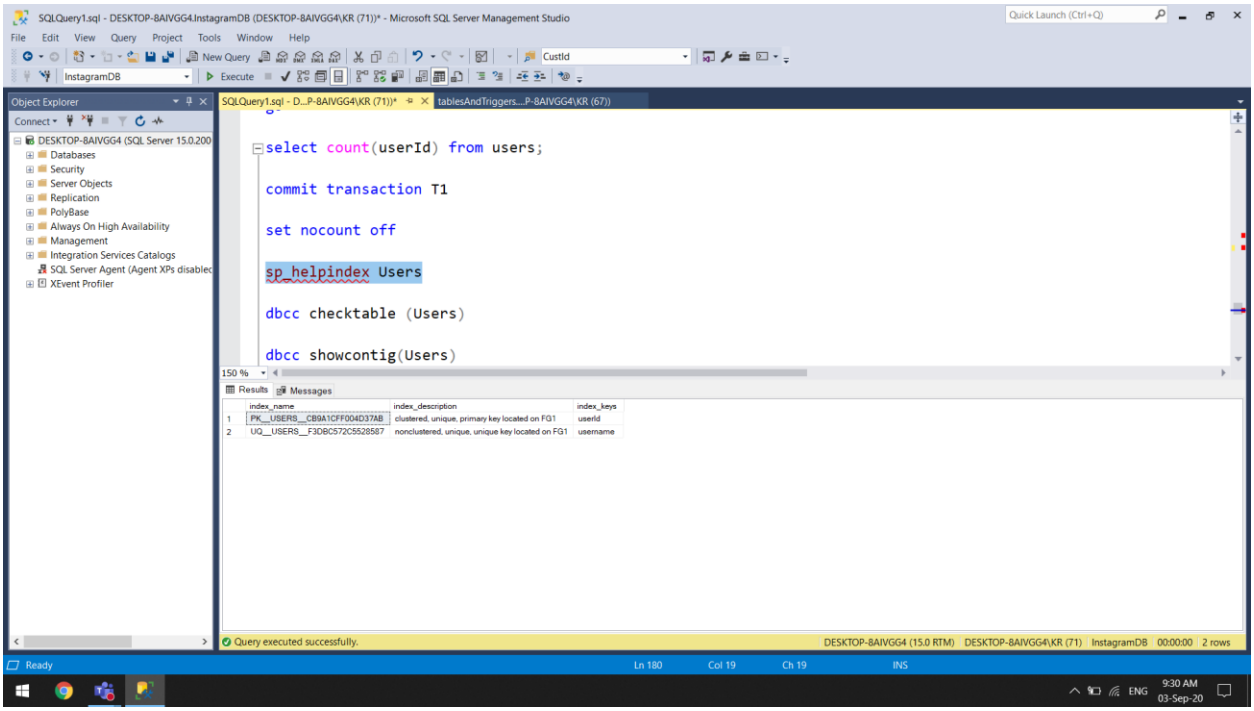
17. To see the pages used by the resource database



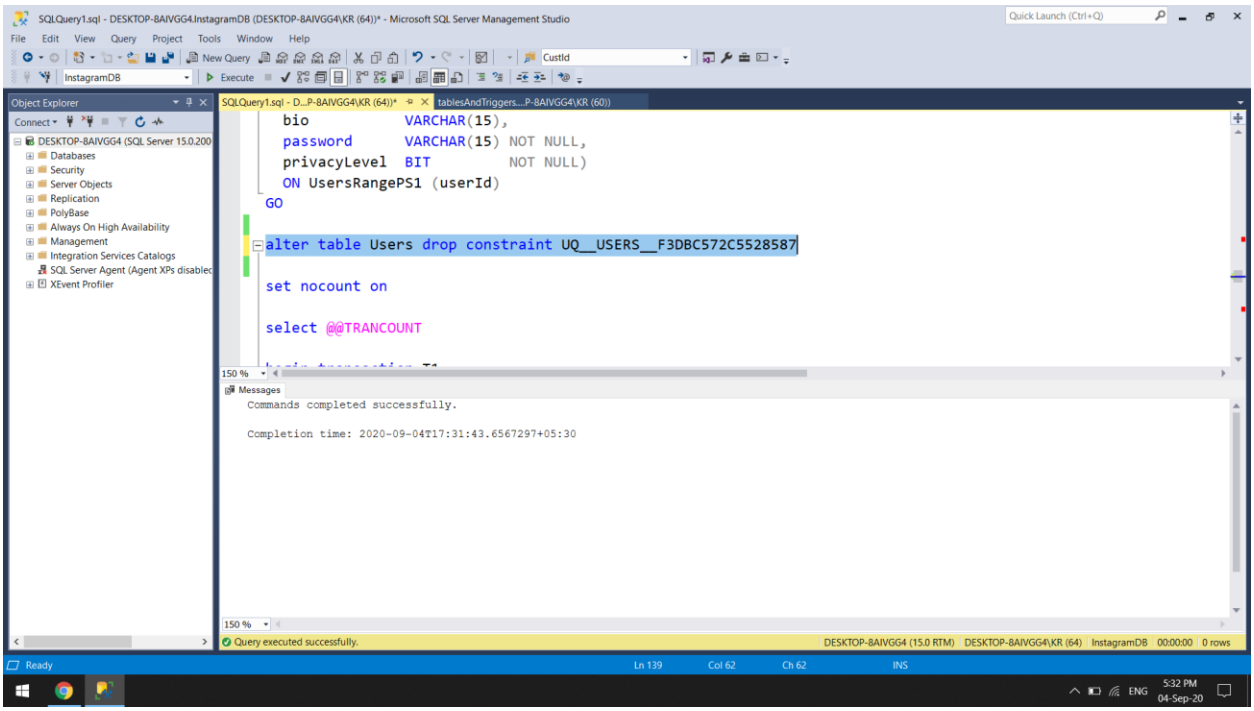
Case 2:

Single table with partition, created one non-clustered index:

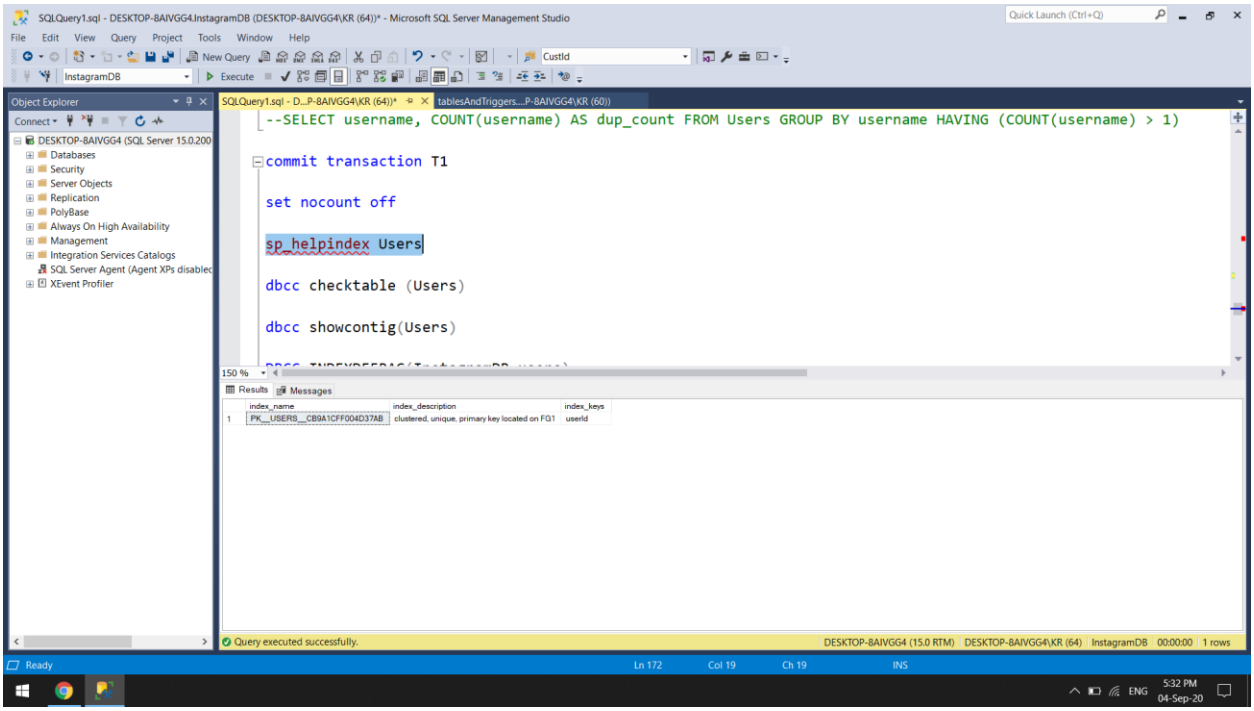
1. Dropped the index we had created in Case 1, i.e, dropped Users_username_idx. It will be created again after partitioning the table.



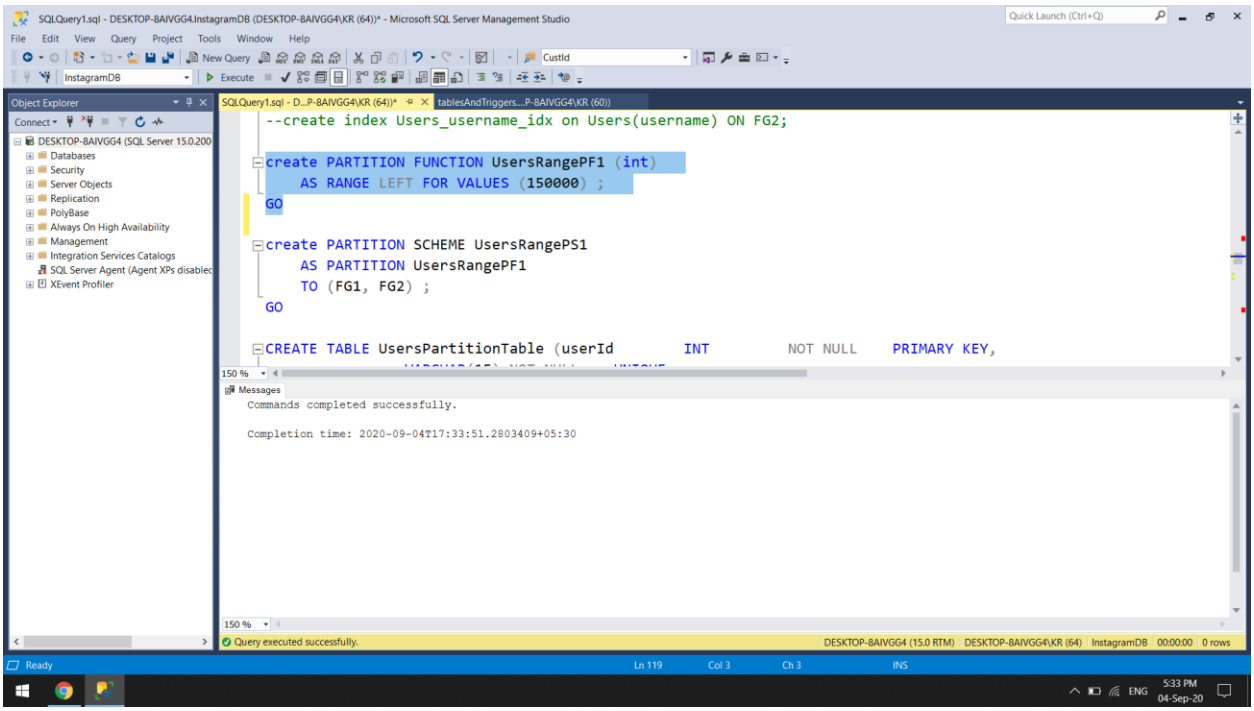
2. Dropped the Unique constraint on username attribute in Users, because when partitioning a unique index(clustered or nonclustered), the partitioning column must be chosen from among those used in the unique index key.

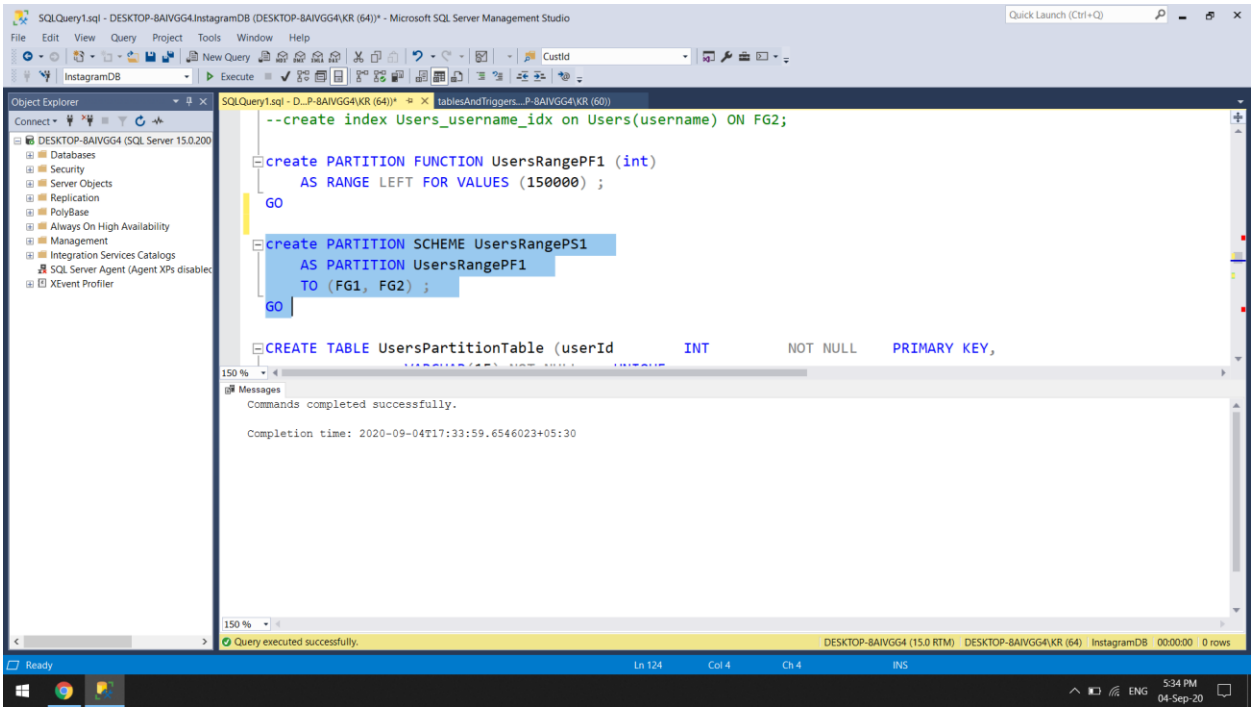


3. The current indexes on Users table

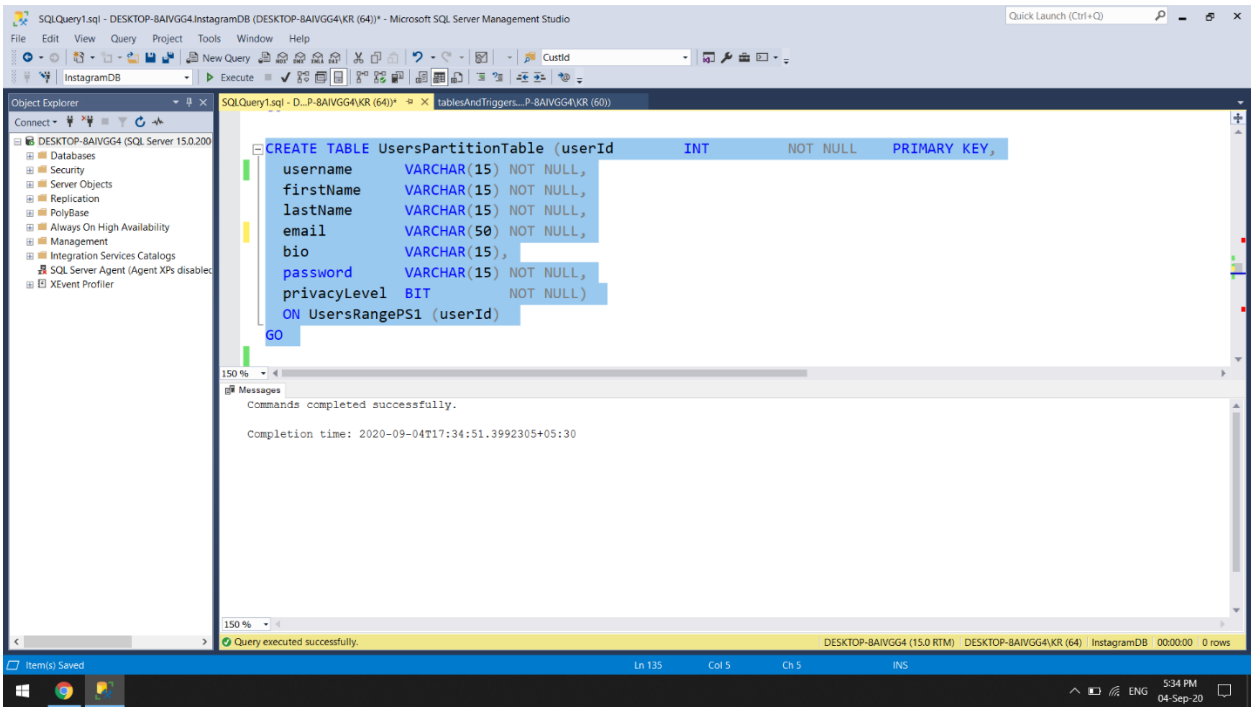


4. Create a partition function and partition scheme. We are partitioning the table into two- tuples that go into FG1 for userId values lesser than 150000 and into FG2 for userId values greater than 150000

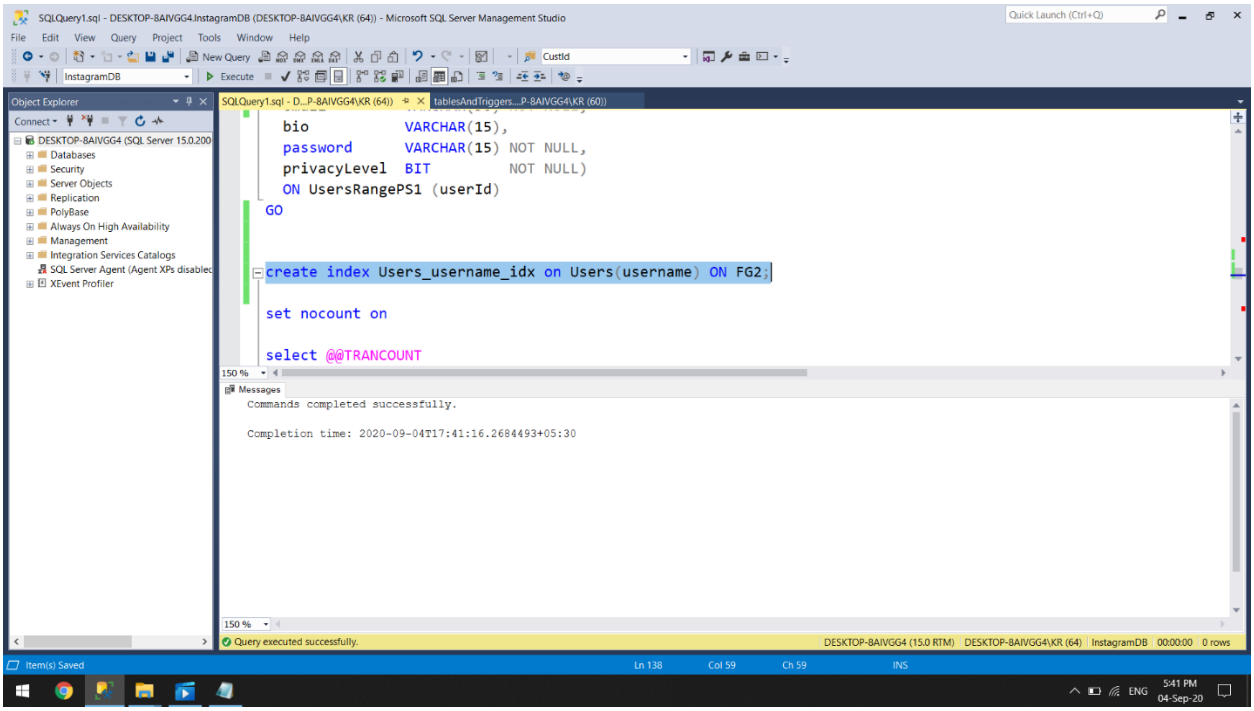




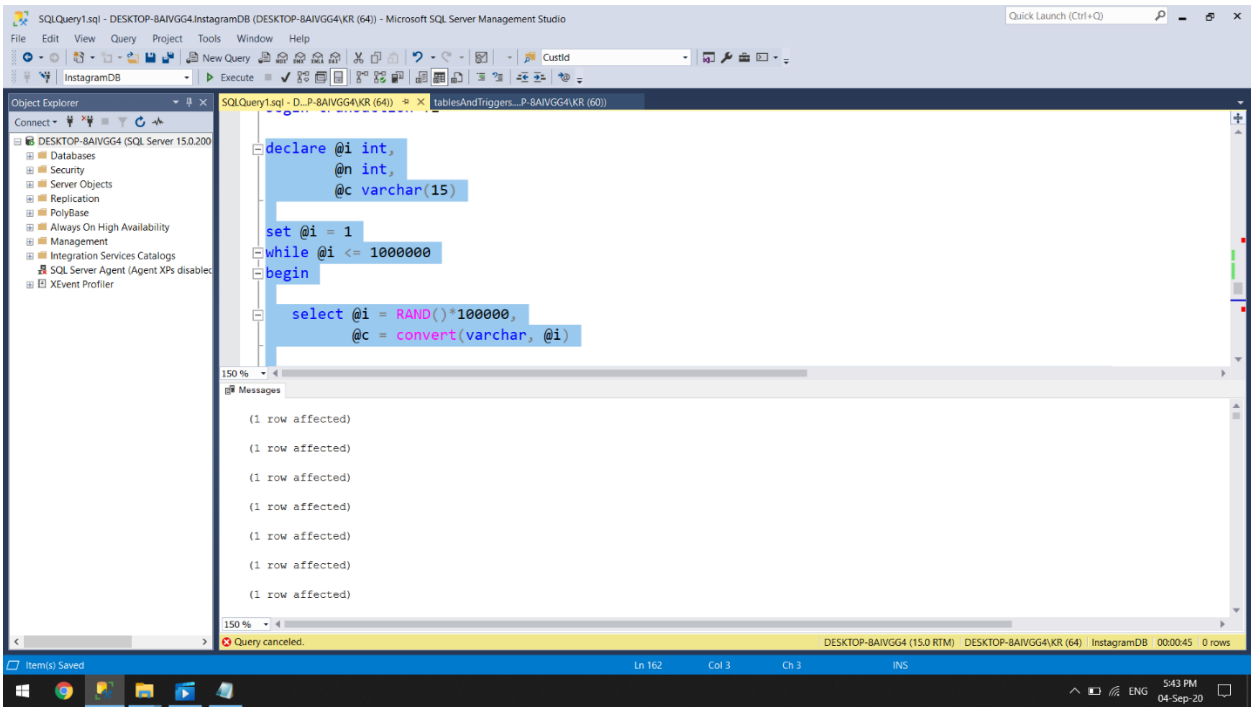
5. Create the partition table schema.



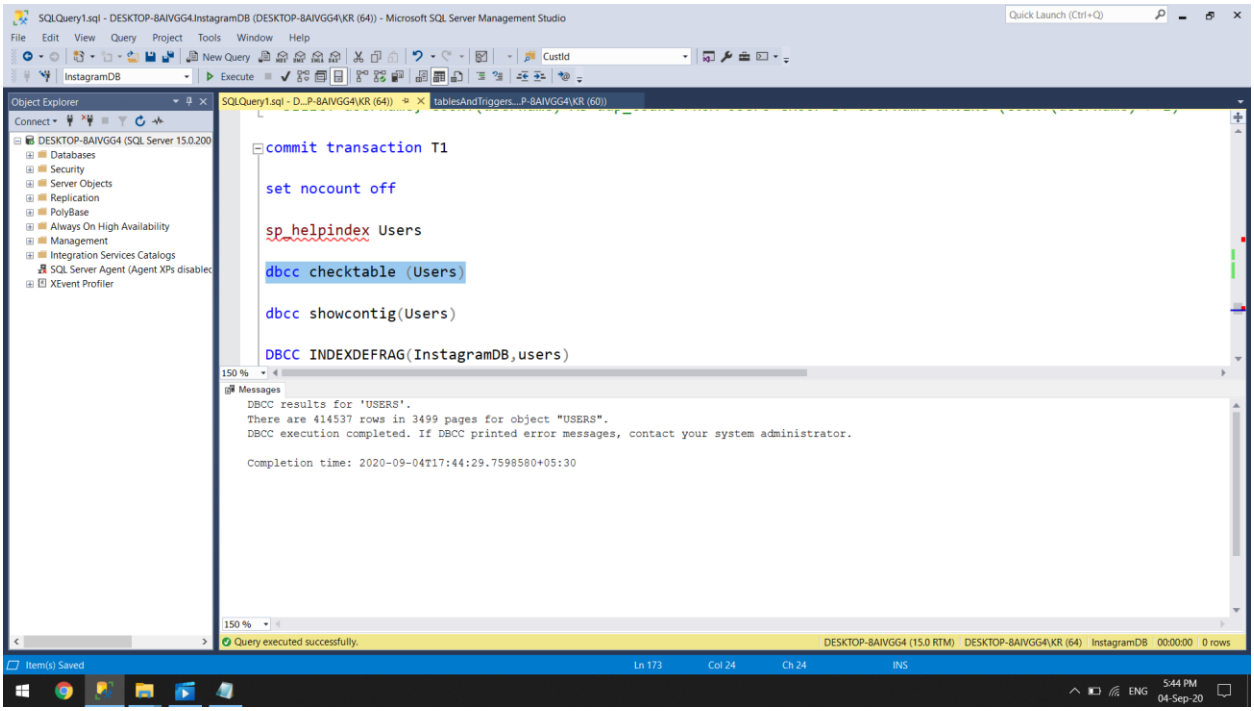
6. Now we create the Users_username_idx on Users table and store it in FG2



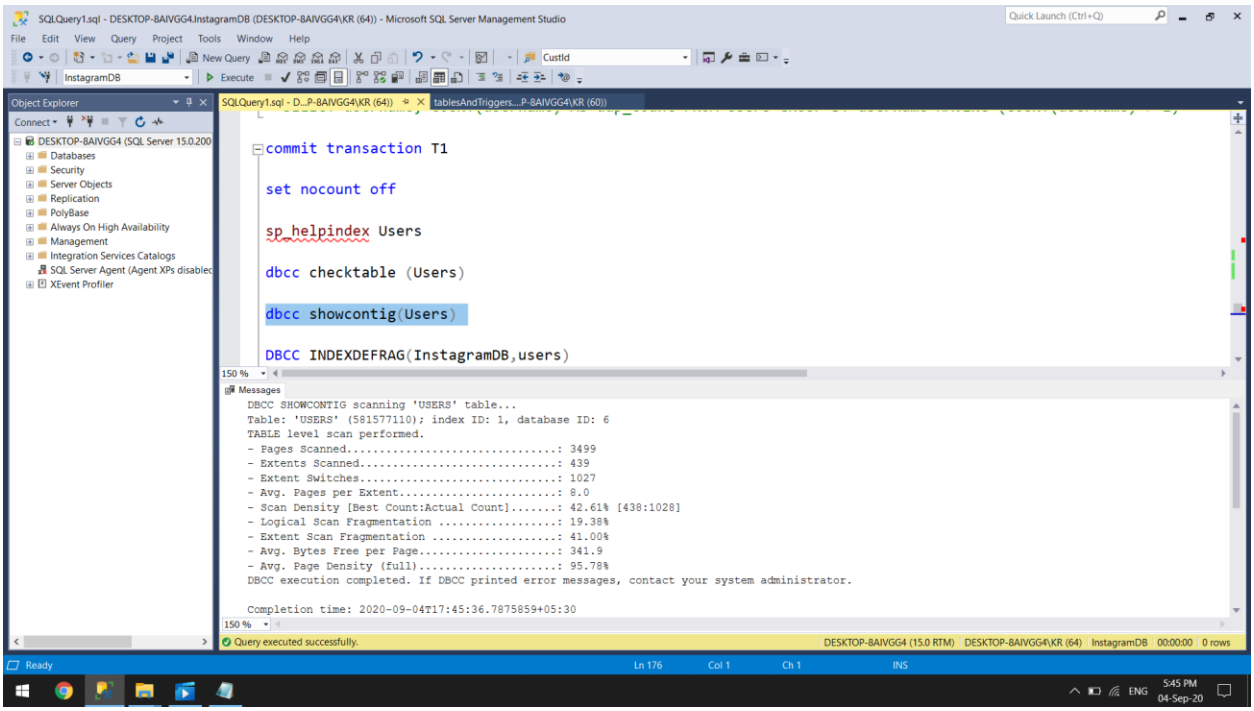
7. Add random tuples into the Users table



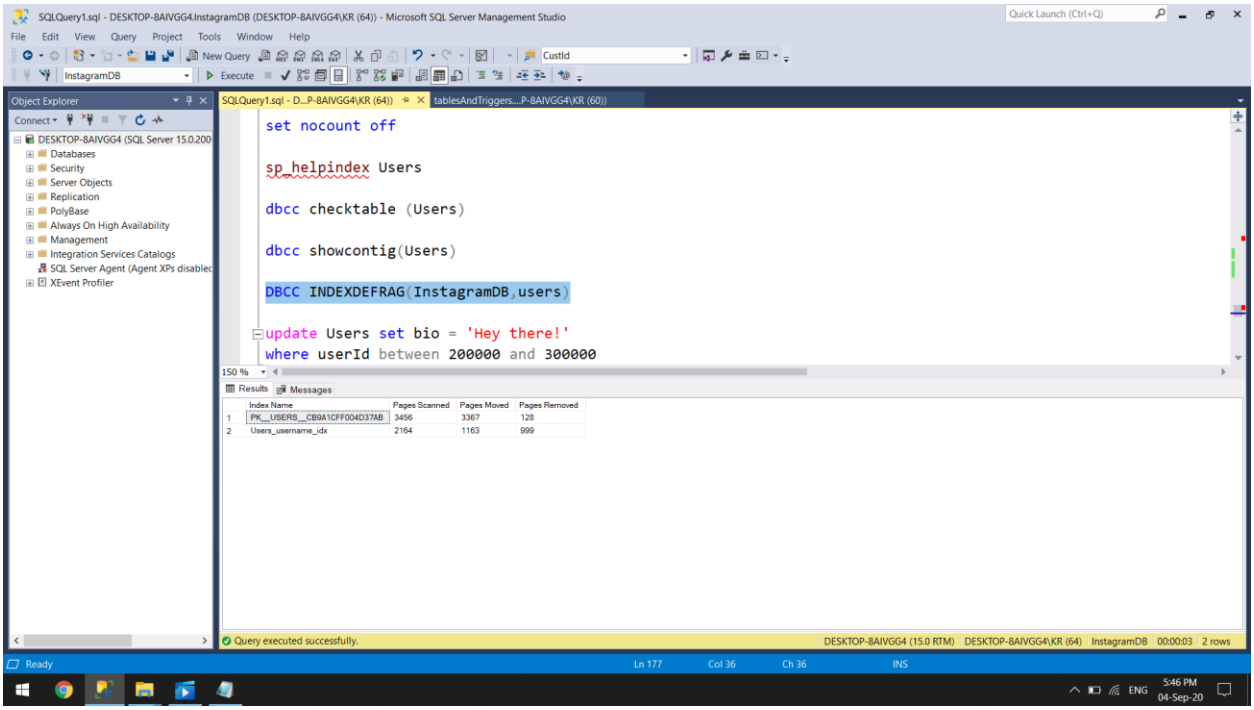
8. The 414537 tuples have been stored on 3499 pages.



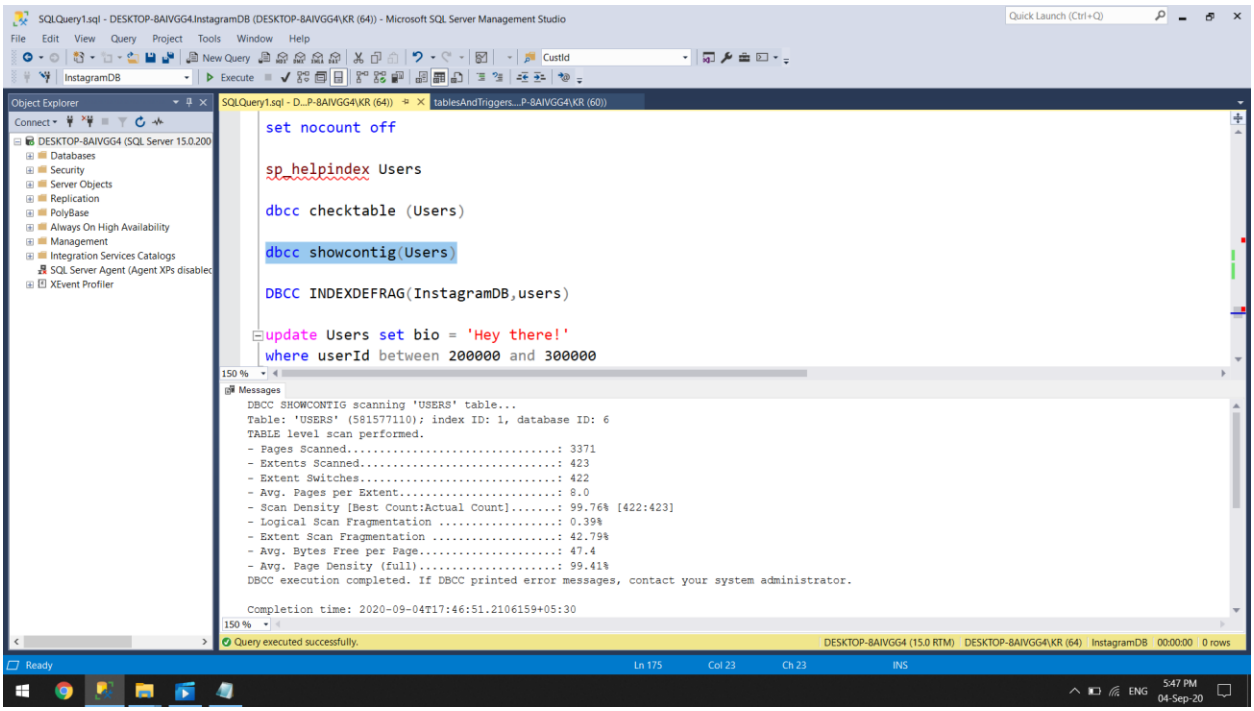
9. Checking the average page density, we find it to be 95.78% which is pretty good.



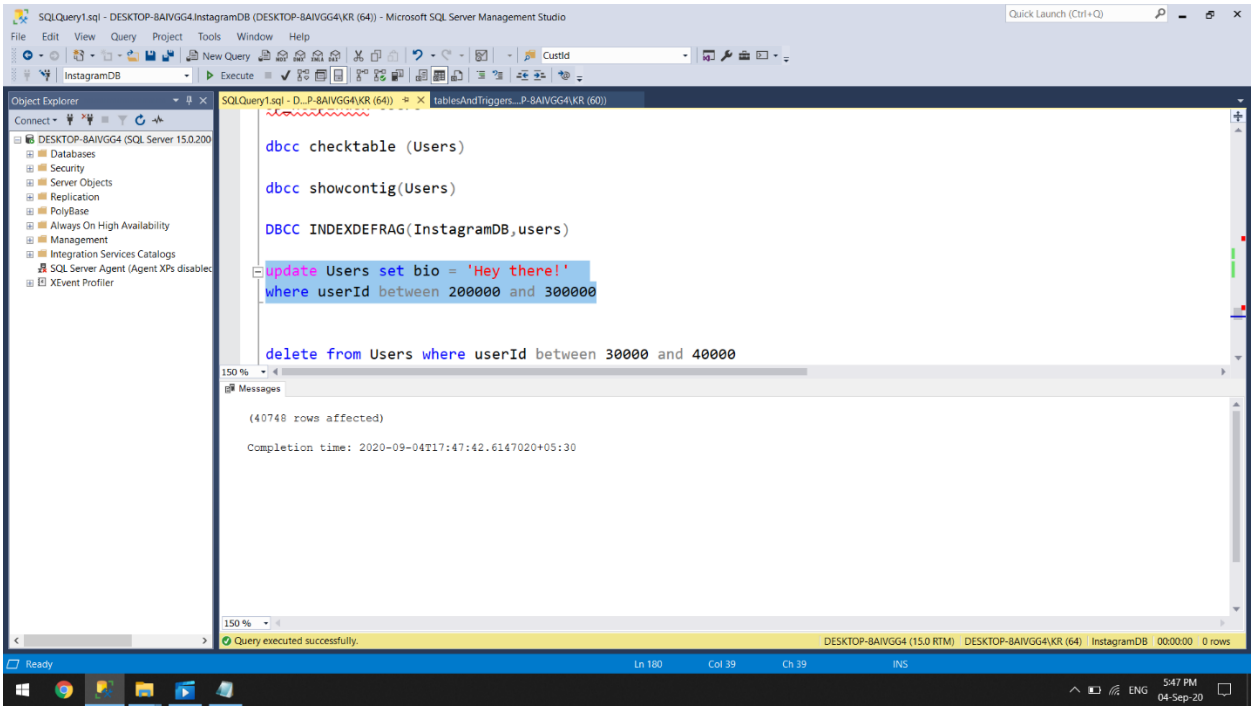
10. We do an index based defragmentation to improve the average page density further (if possible) and reduce the overall number of pages used.

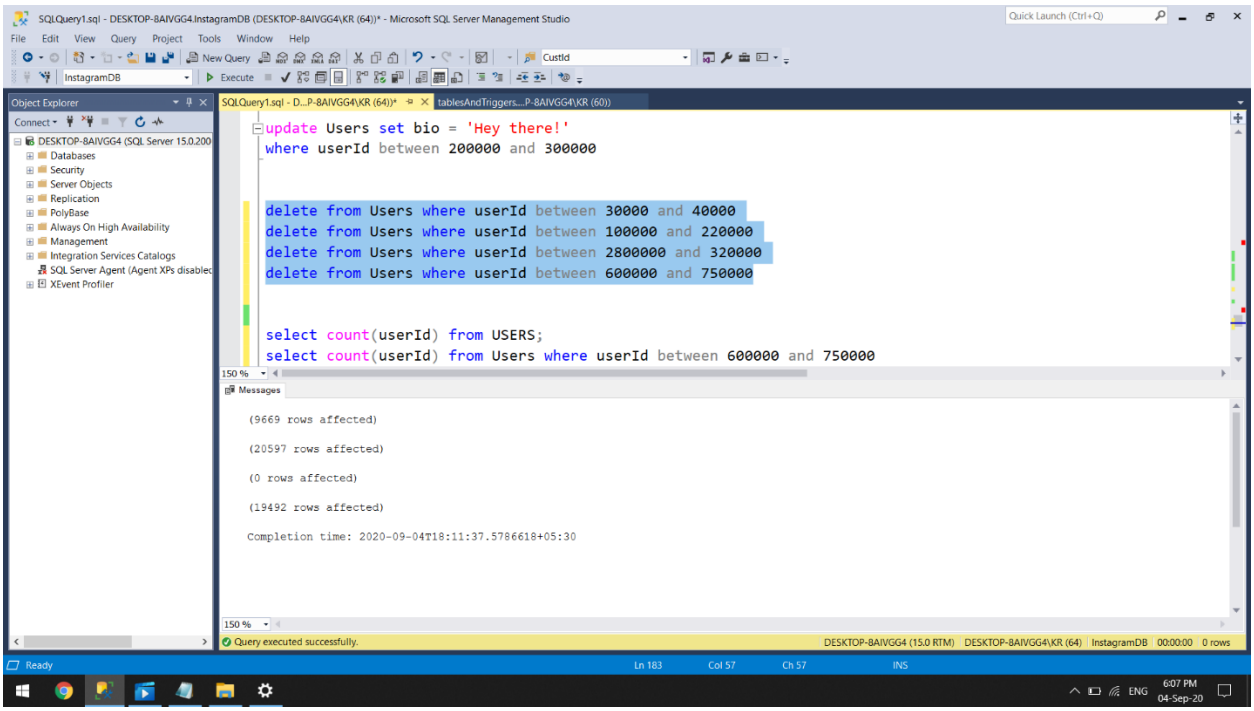


11. The average page density has now improved to 99.41%.

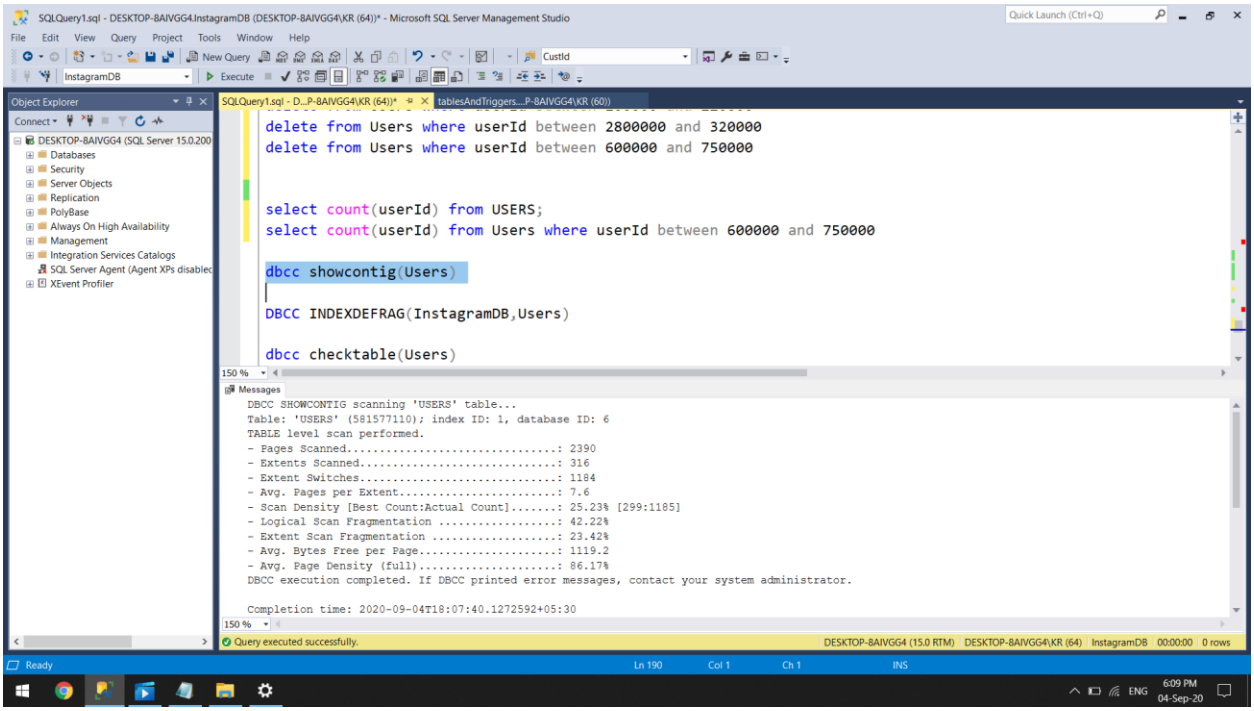


12. Similar to case 1, we update the bio attribute, and delete some tuples to see how the storage on the disk changes.

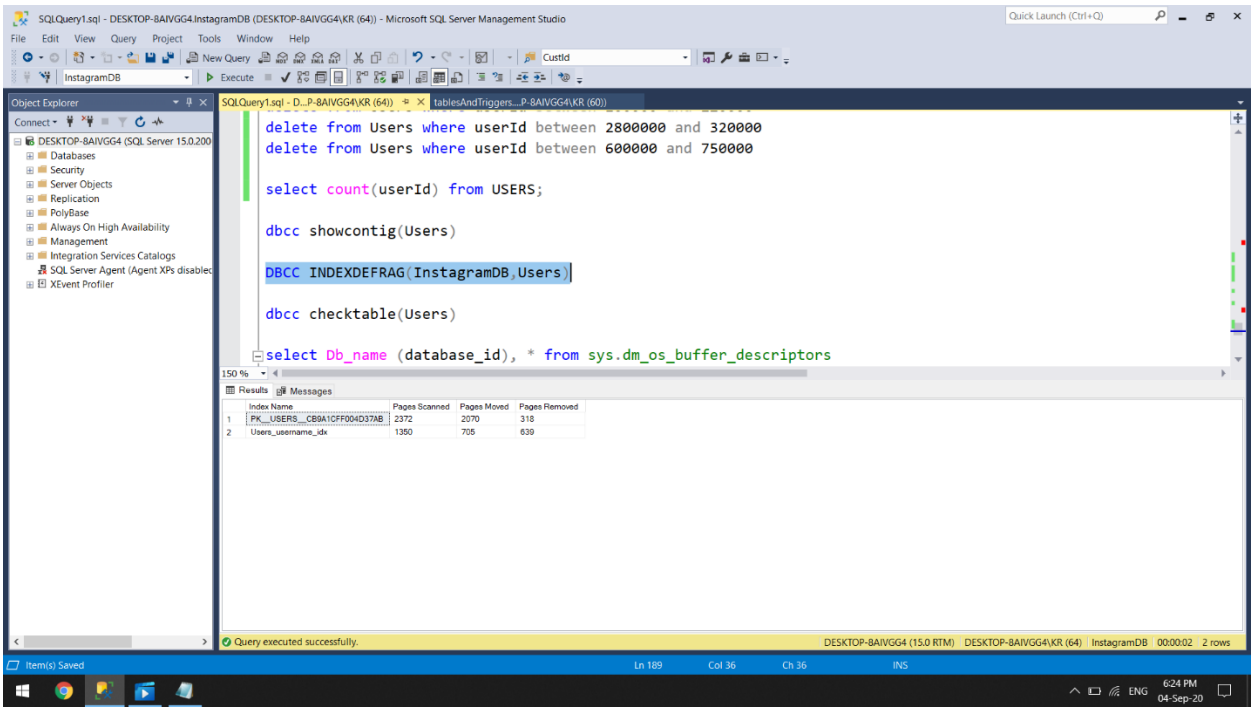




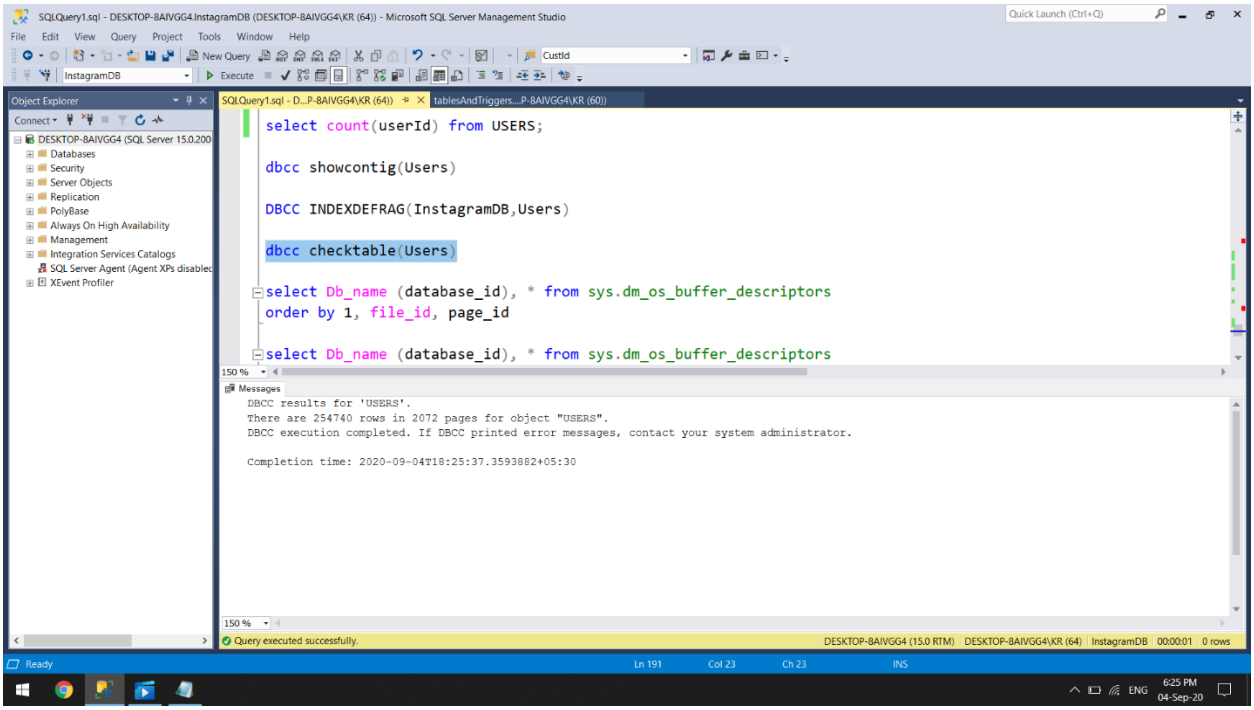
13. We see that the average page density has come down to 86.17%



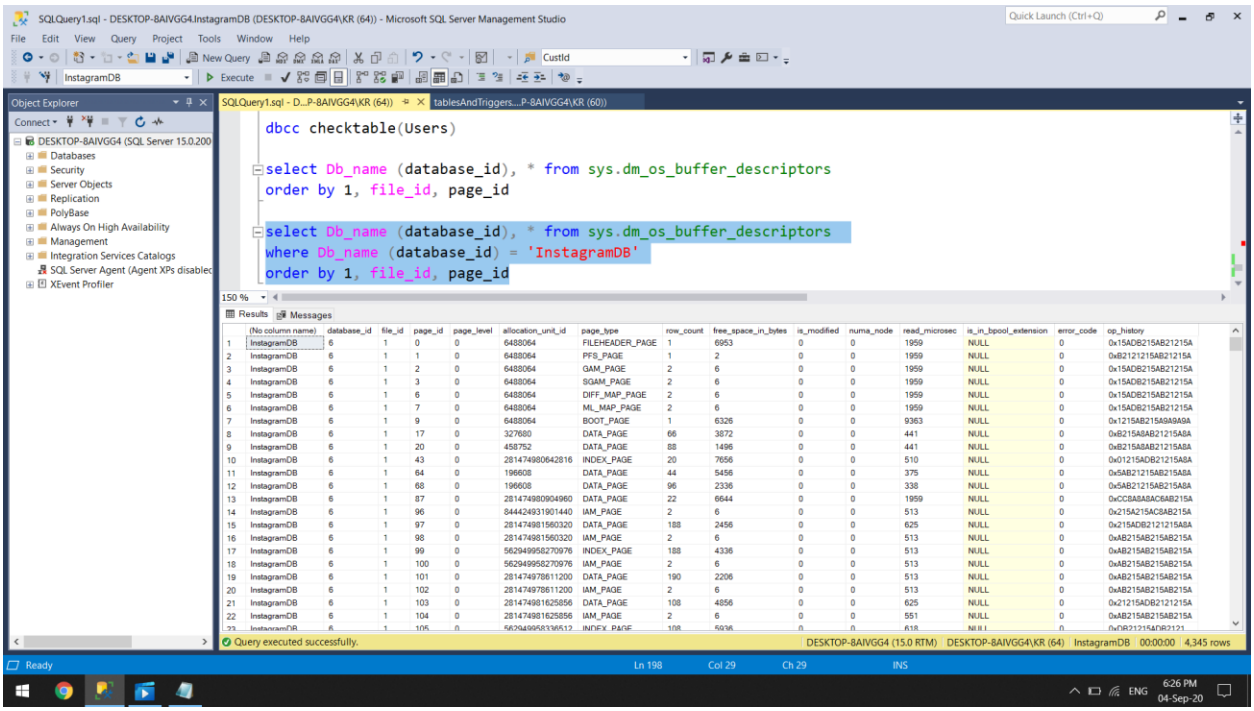
14. Perform the defragmentation process again.



15. We check that the number of pages used have reduced, so there is an optimization in the storage space used.



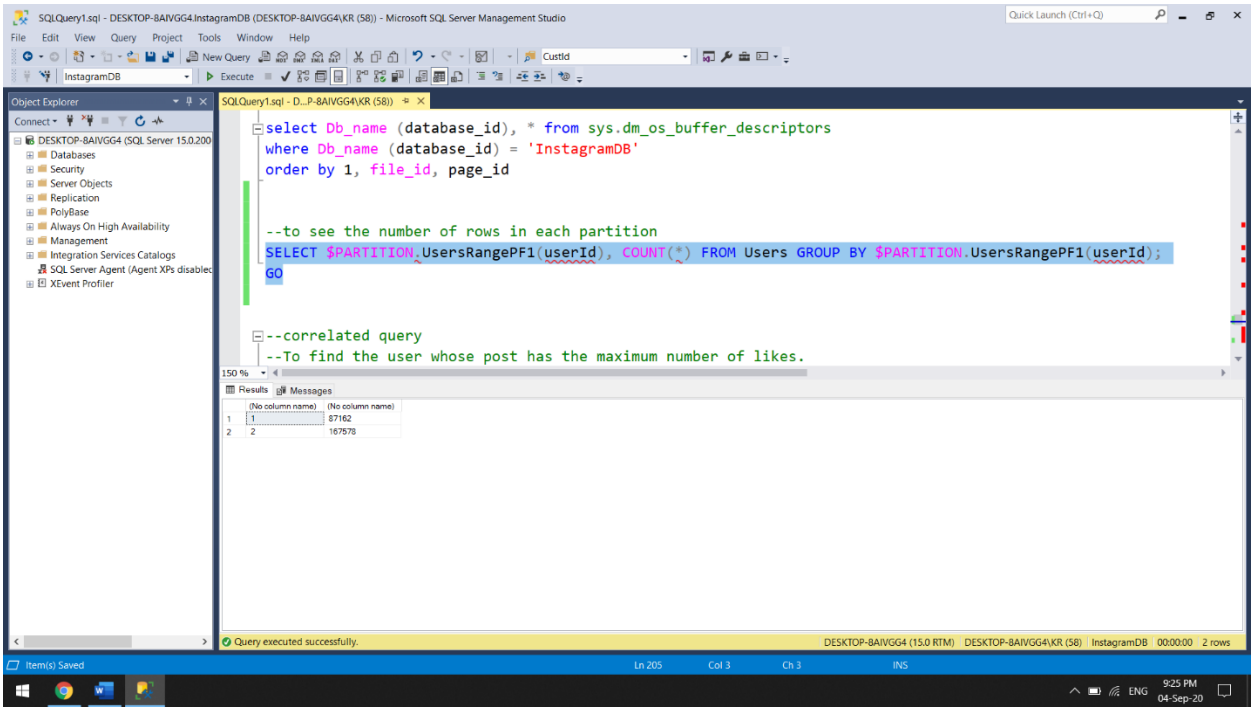
16. To see the pages used by the resource database


SQLQuery1.sql - DESKTOP-8AIVGG4.InstagramDB (DESKTOP-8AIVGG4\KR (64)) - Microsoft SQL Server Management Studio
Object Explorer
SQLQuery1.sql - D...P-8AIVGG4\KR (64))
tablesAndTriggers...P-8AIVGG4\KR (60))
dbcc checktable(Users)
select Db_name (database_id), * from sys.dm_os_buffer_descriptors
order by 1, file_id, page_id
select Db_name (database_id), * from sys.dm_os_buffer_descriptors
where Db_name (database_id) = 'InstagramDB'
order by 1, file_id, page_id
Results
Messages
Query executed successfully.
Ready

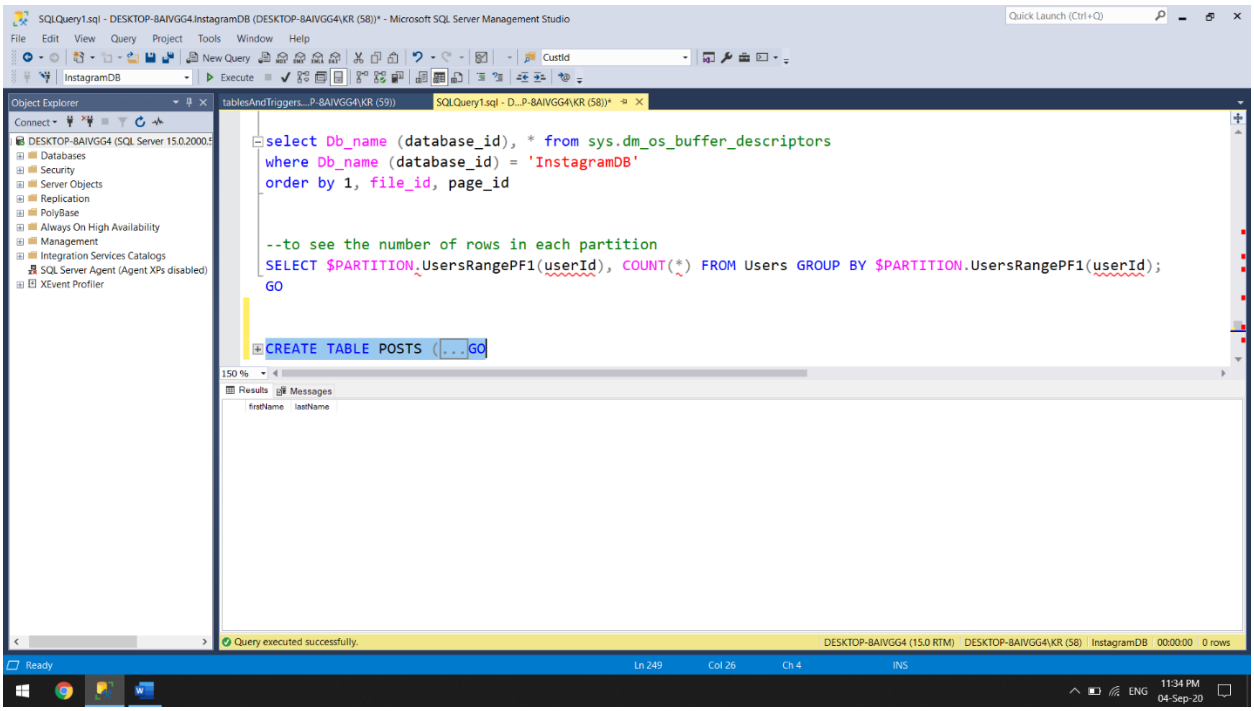
Case 3:

Two tables, with indexes created to optimize query execution:

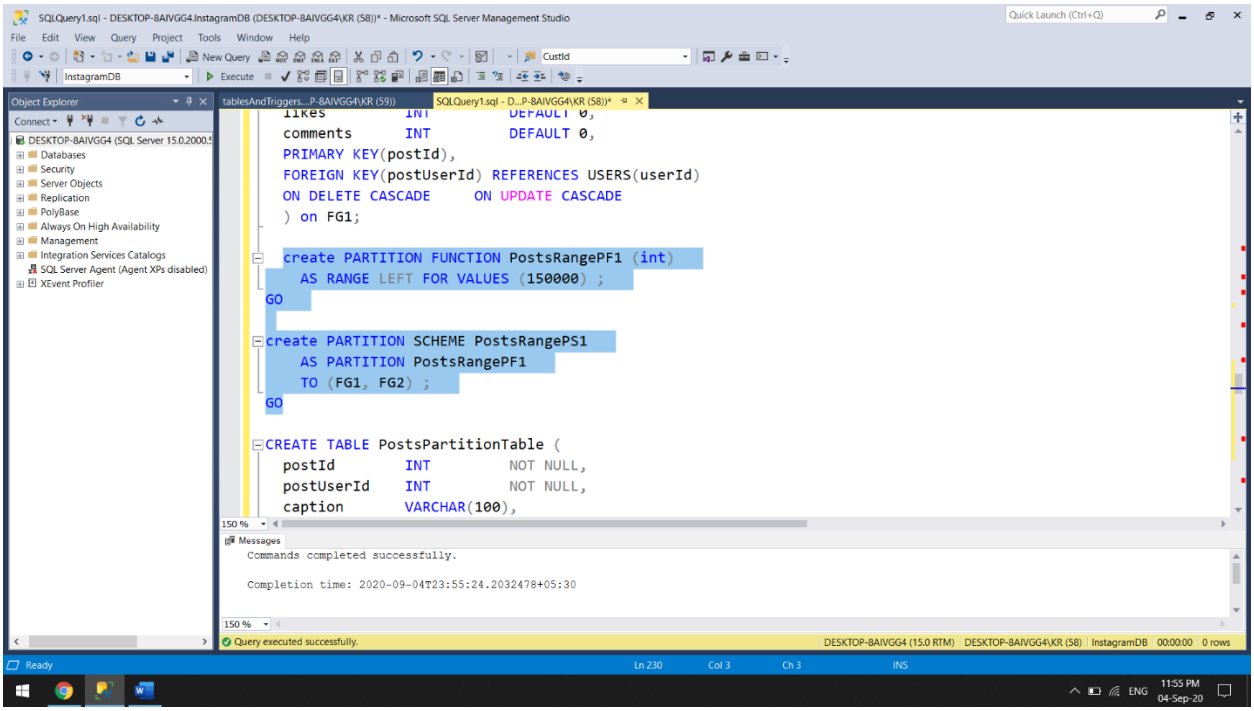
1. Currently we have only the table Users which is partitioned between FG1 and FG2. The following command shows the number of tuples on each filegroup.

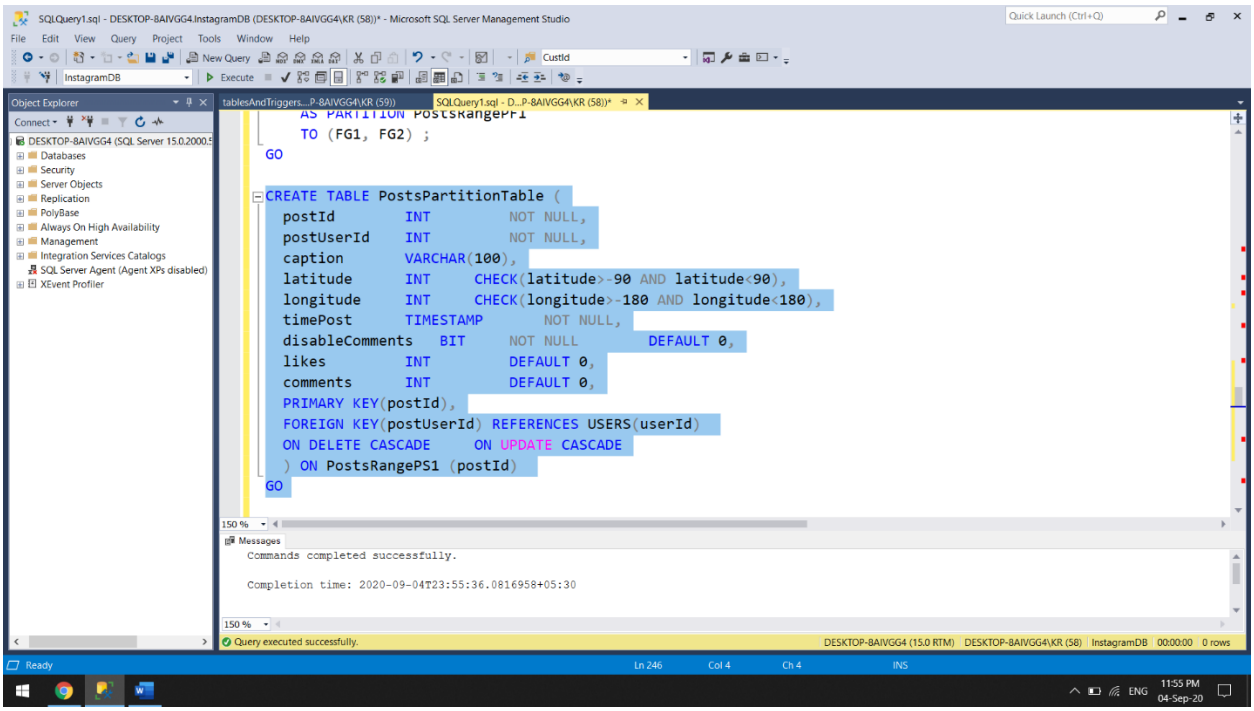


2. We create the table Posts on the same filegroup as Users, i.e, on FG1



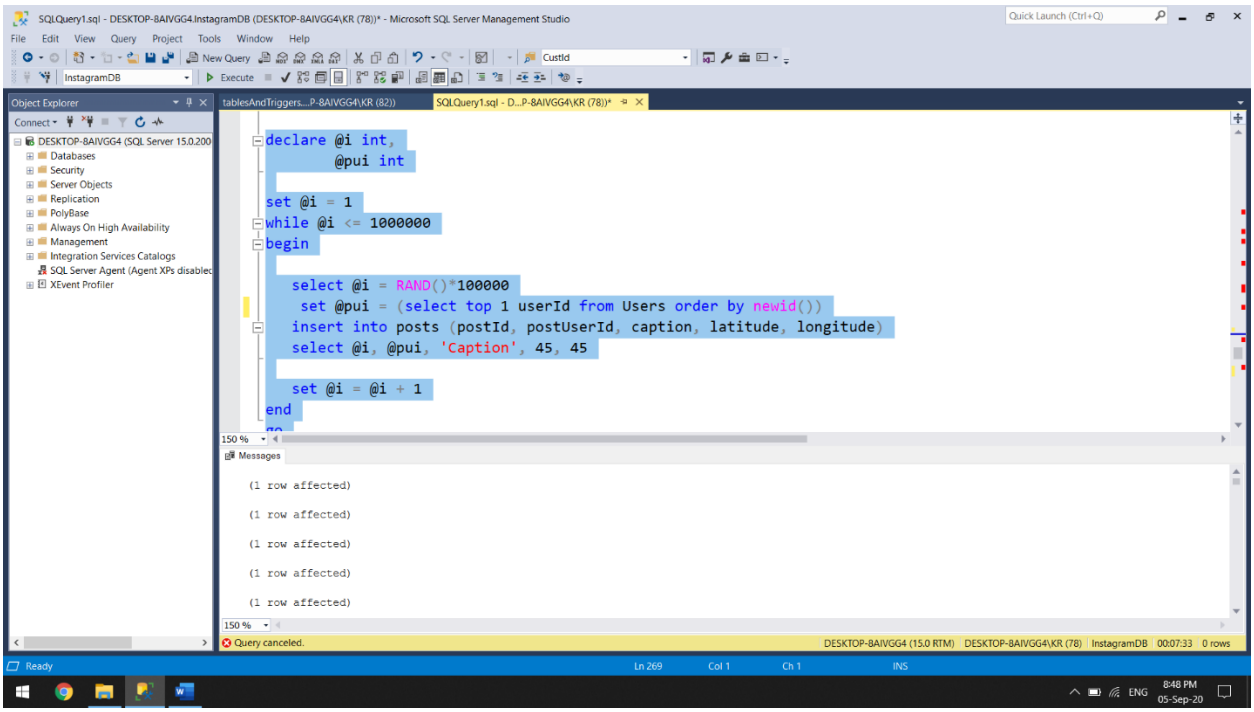
3. Partition the table Posts based on postId, similar to partitioning table Users based on userId.



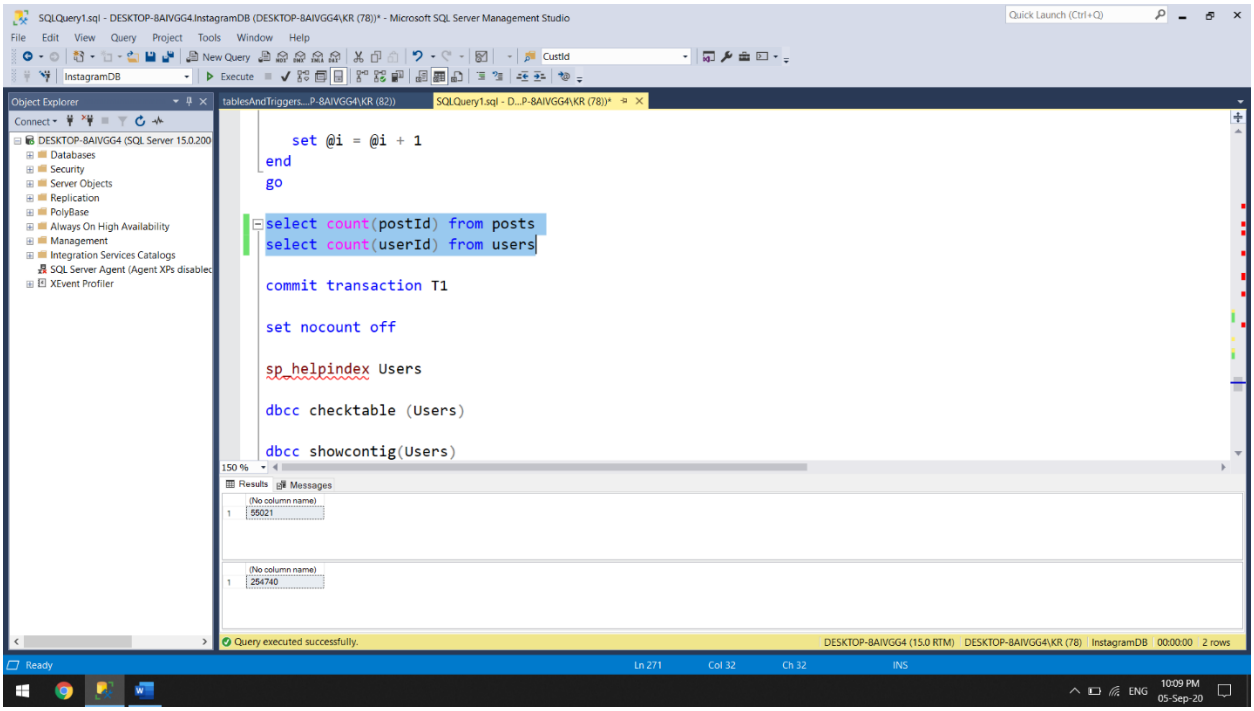


4. Insert a couple of thousand of tuples into the Posts table. Note that this table has an attribute called postUserId which is a foreign key referencing userId of the user table. So, to get this attribute's values from the Users table in random order we use the statement

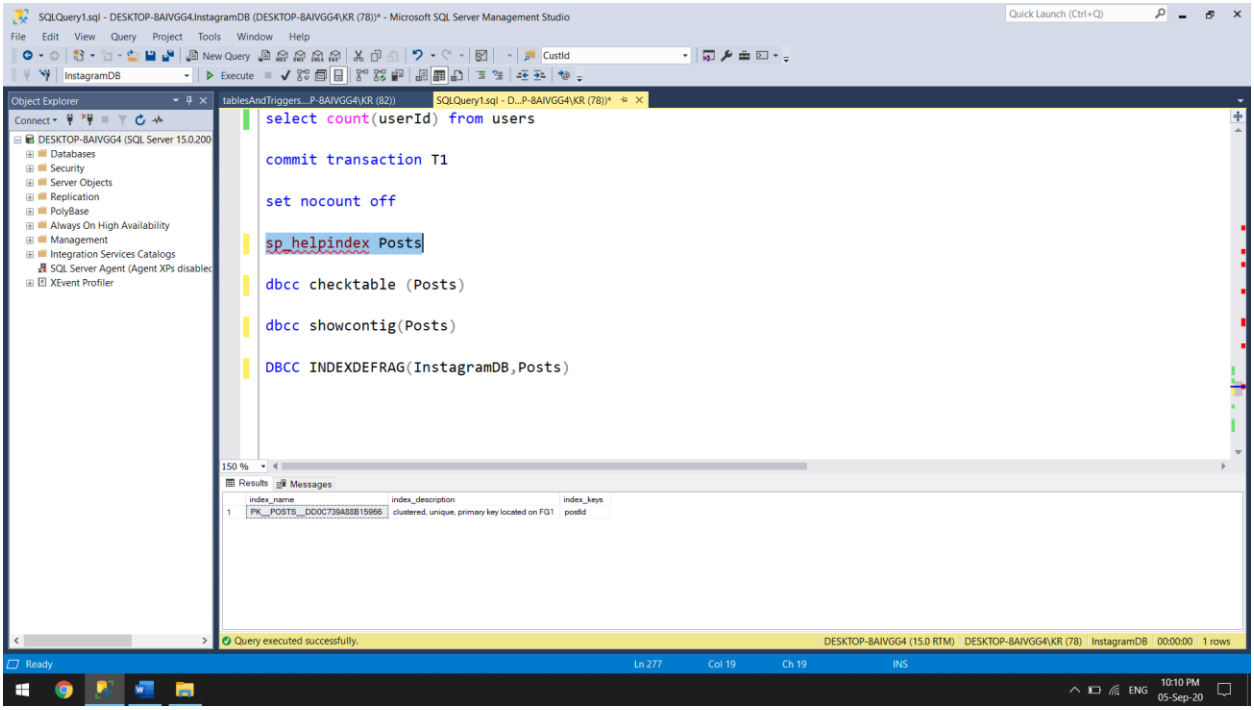
Set @pui = (select top 1 userId from Users order by newid())



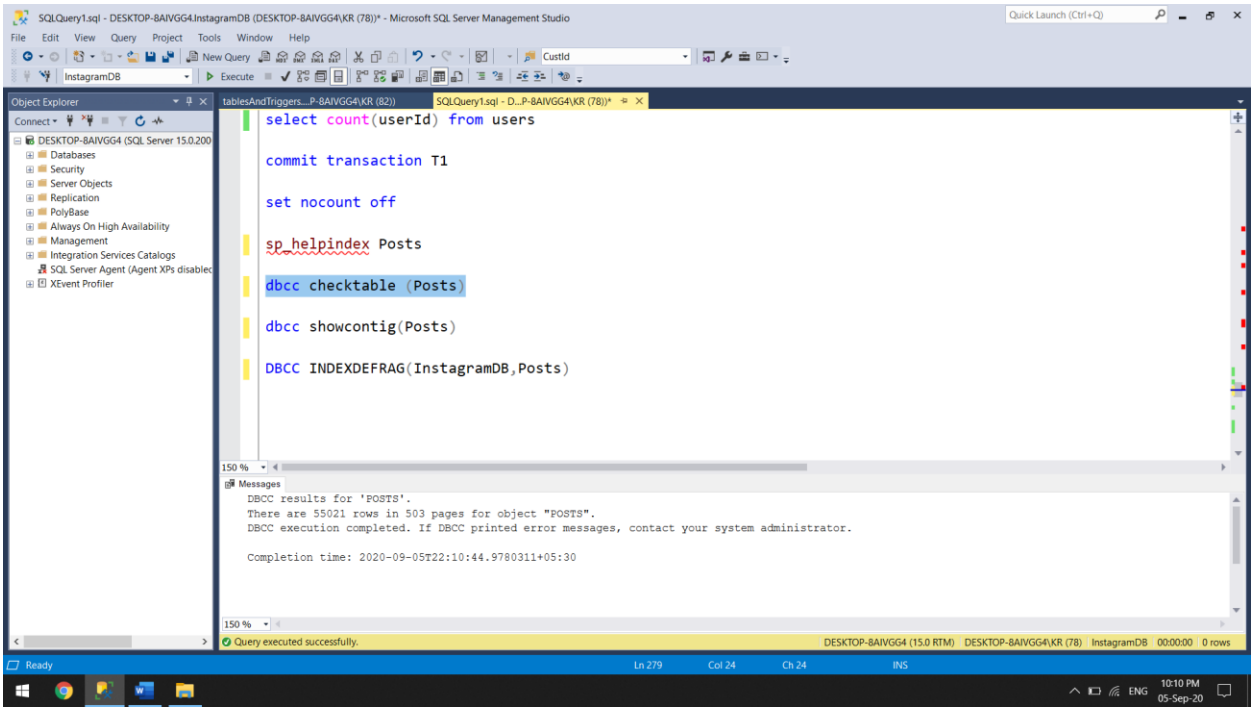
5. We have populated the Posts table with 55021 values. (didn't add more rows because it was taking very long to do so based on the above function)

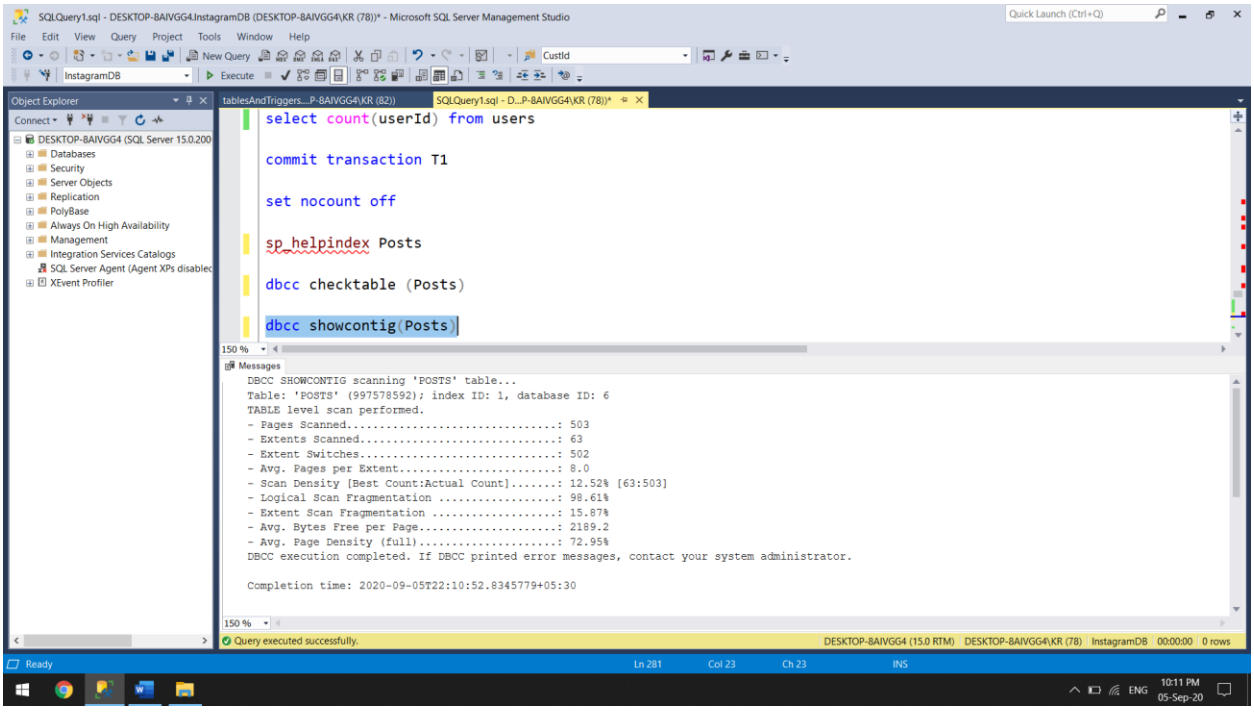


6. The current indexex on Posts consists of only the clustered, unique primary key index located on FG1.

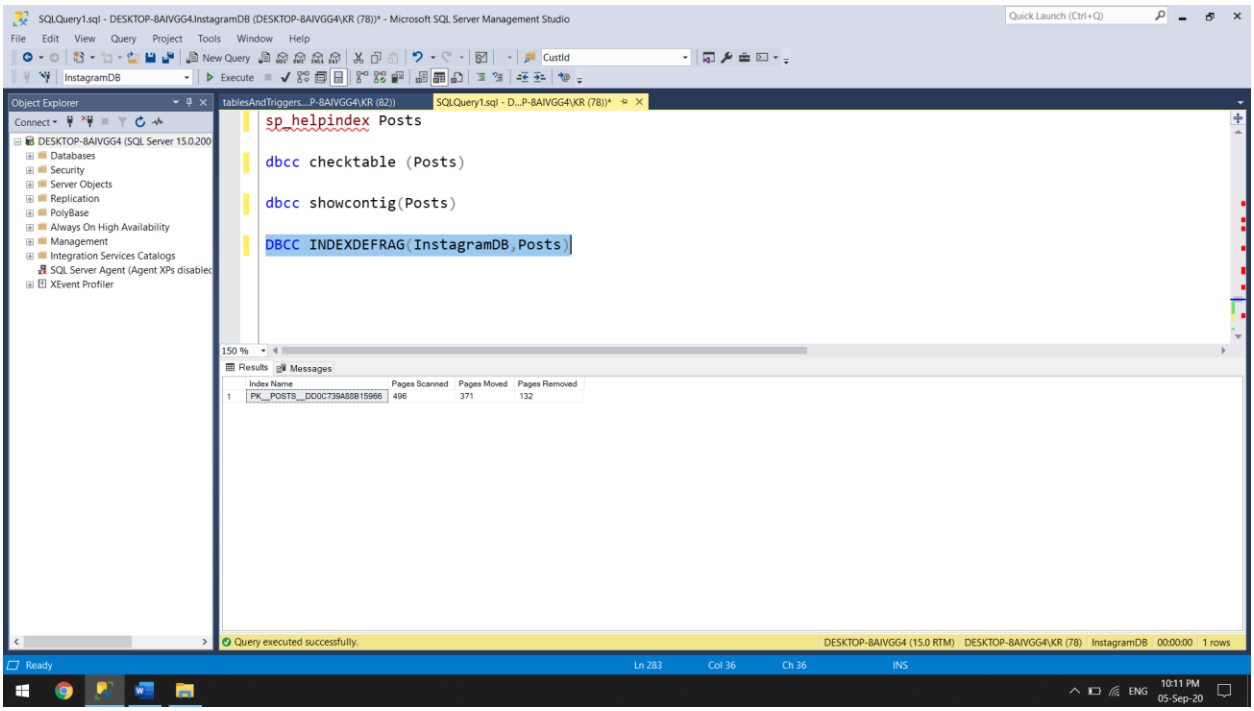


7. The 55021 tuples use 503 pages at this moment and the average page density is 72.95%, which could be improved.

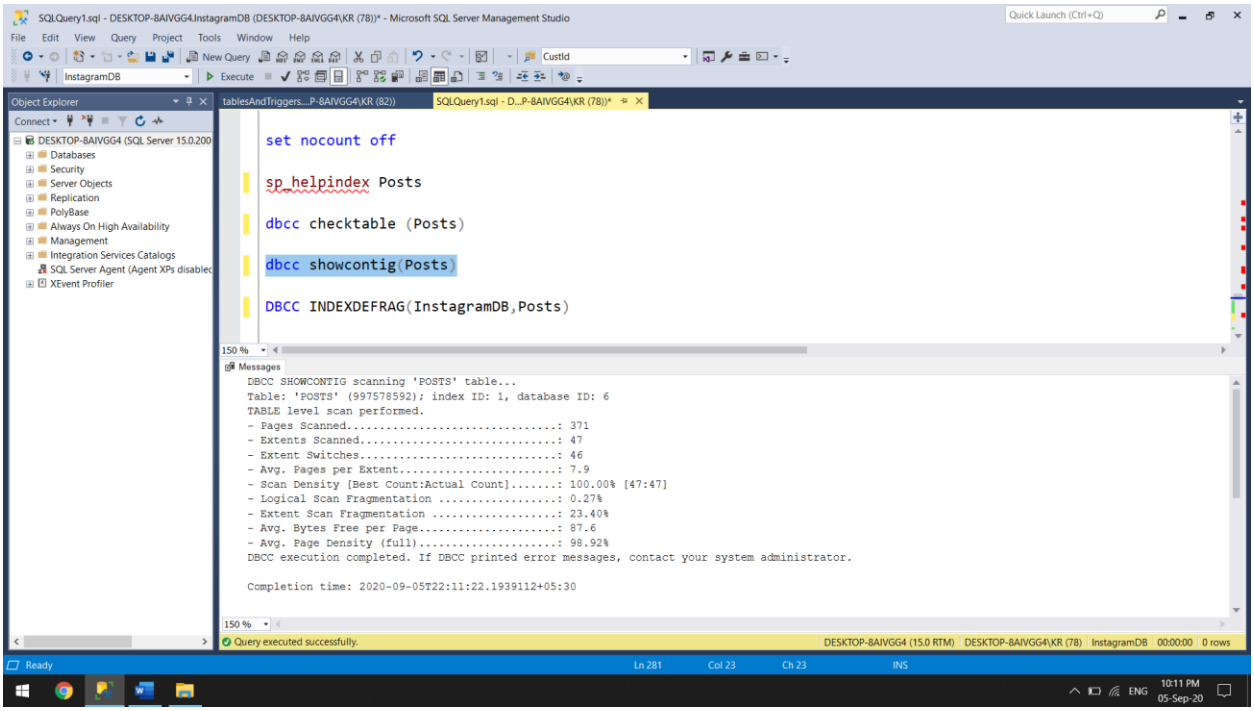




8. We perform an index based defragmentation. 371 pages are moved and 132 are removed.

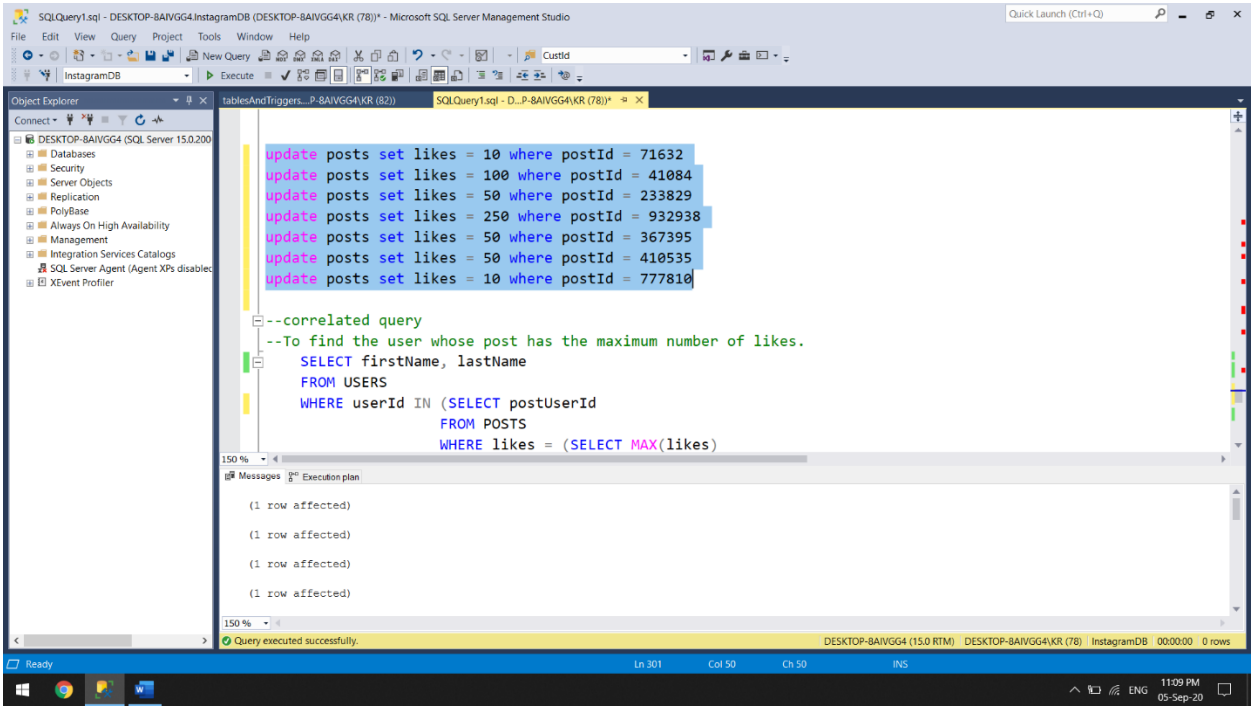


9. The average page density has now improved to 98.92%.

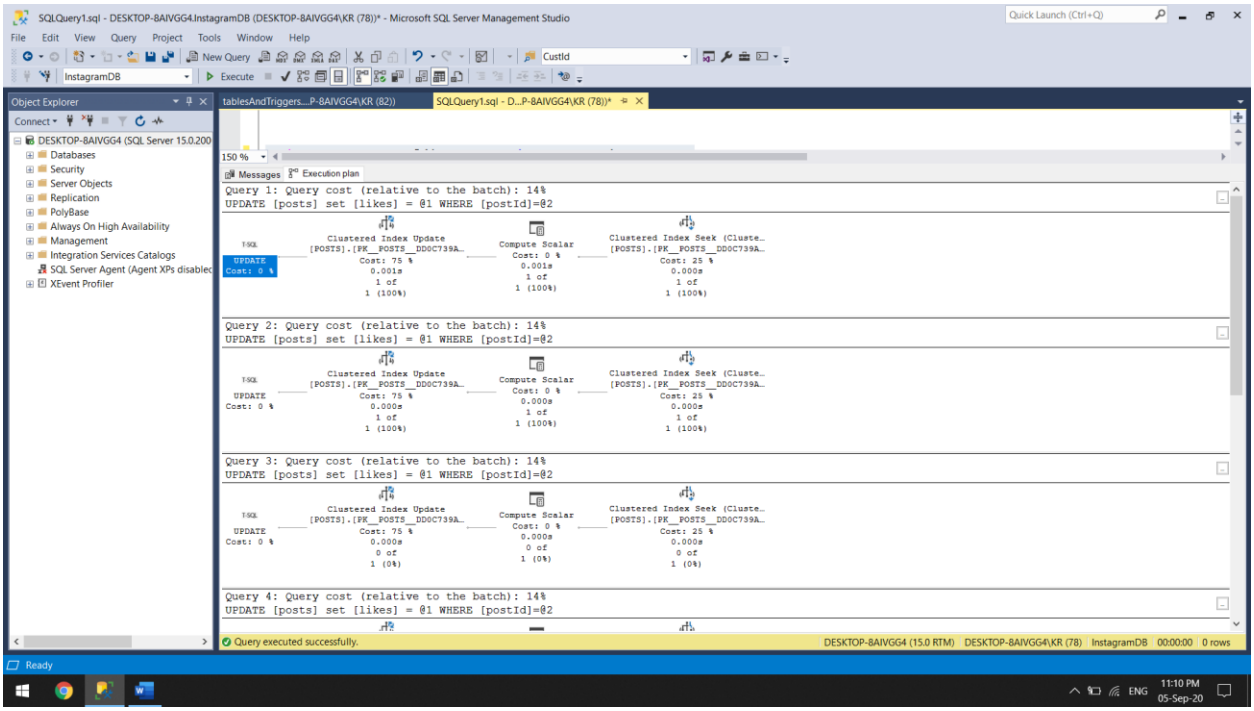


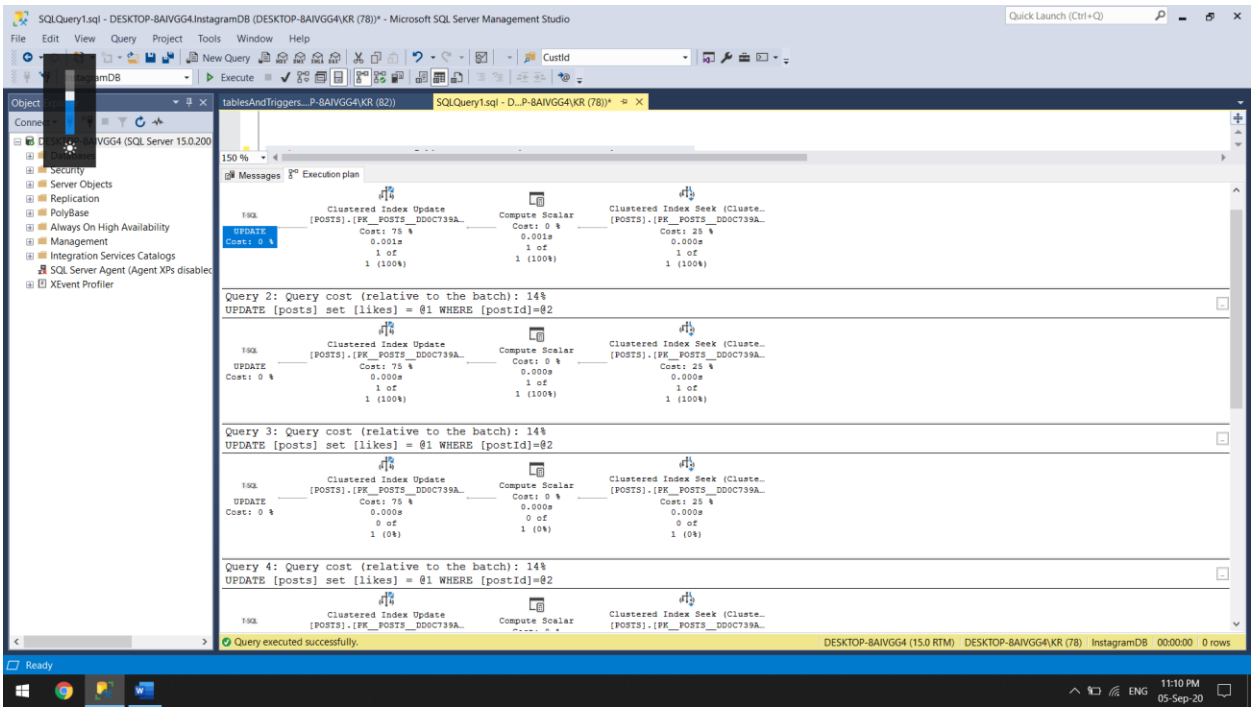
10. From this point onwards, we will be looking at queries and revising indexes on the tables for optimized execution of the queries.

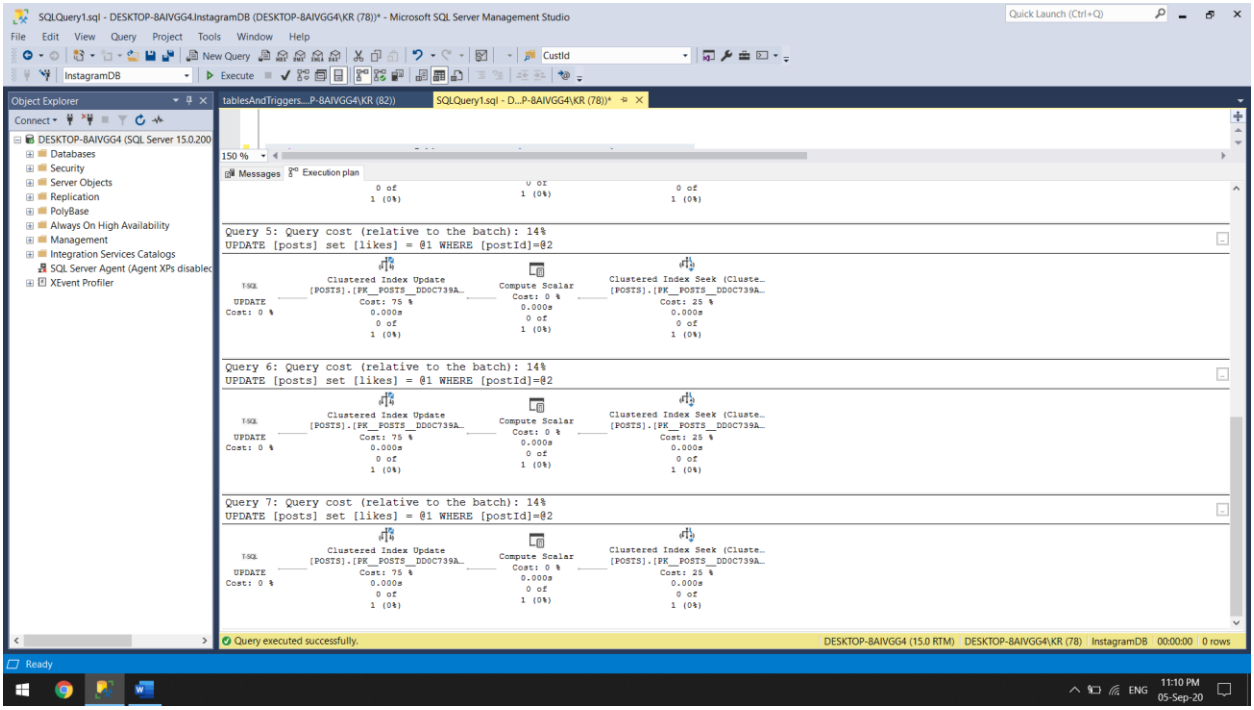
Here we update the likes attribute for some tuples before we search for the user whose post has the maximum number of likes. Before doing this, all the values of likes were 0.



11. This is the execution plan for the above 7 update commands

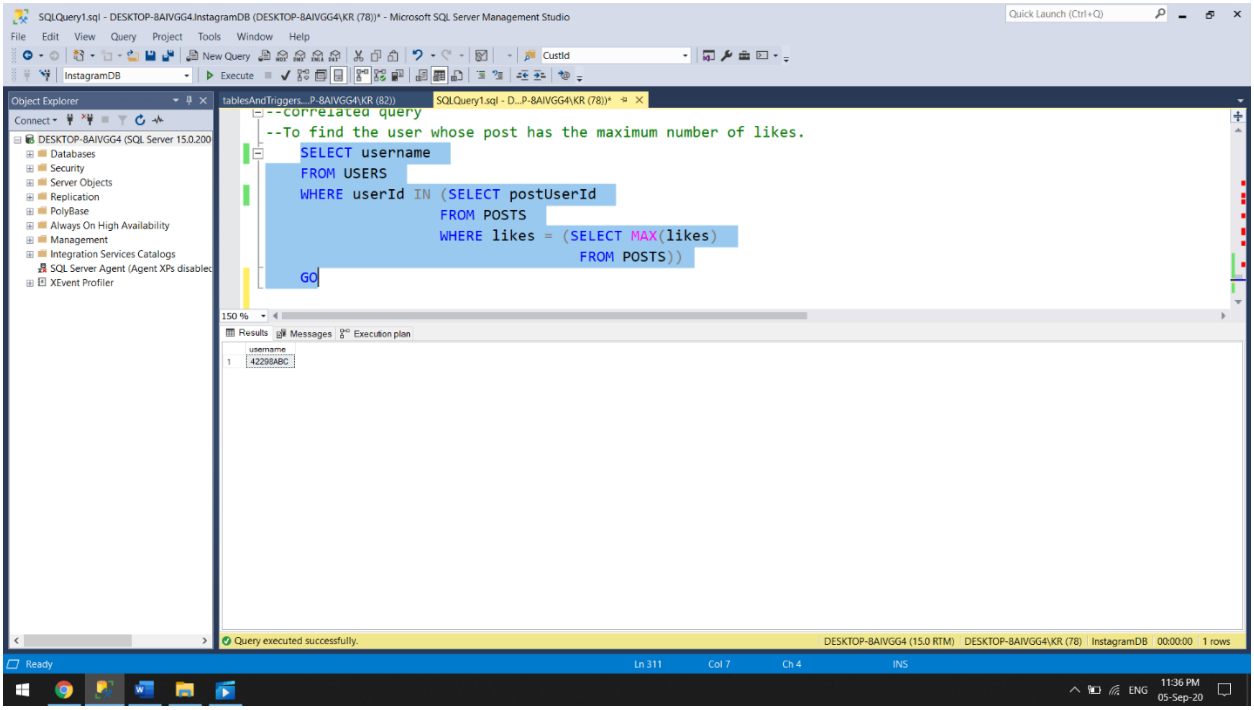



SQLQuery1.sql - DESKTOP-8AIVGG4.InstagramDB (DESKTOP-8AIVGG4\KR (78))* - Microsoft SQL Server Management Studio
Quick Launch (Ctrl+Q)
File Edit View Query Project Tools Window Help
New Query
Execute
InstagramDB
CustId
Object Explorer
Connect
DESKTOP-8AIVGG4 (SQL Server 15.0.200
Security
Server Objects
Replication
PolyBase
Always On High Availability
Management
Integration Services Catalogs
SQL Server Agent (Agent XPs disablec
XEvent Profiler
tablesAndTriggers....P-8AIVGG4\KR (82))
SQLQuery1.sql - D...P-8AIVGG4\KR (78))*
150 %
Messages
Execution plan
Query 2: Query cost (relative to the batch): 14%
UPDATE [posts] set [likes] = @1 WHERE [postId]=@2
Query 3: Query cost (relative to the batch): 14%
UPDATE [posts] set [likes] = @1 WHERE [postId]=@2
Query 4: Query cost (relative to the batch): 14%
UPDATE [posts] set [likes] = @1 WHERE [postId]=@2
Query executed successfully.
DESKTOP-8AIVGG4 (15.0 RTM) DESKTOP-8AIVGG4\KR (78) InstagramDB 00:00:00 0 rows
Ready
ENG
11:10 PM
05-Sep-20


SQLQuery1.sql - DESKTOP-8AIVGG4.InstagramDB (DESKTOP-8AIVGG4\KR (78))* - Microsoft SQL Server Management Studio
Quick Launch (Ctrl+Q)
File Edit View Query Project Tools Window Help
New Query
Execute
InstagramDB
CustId
Object Explorer
Connect
DESKTOP-8AIVGG4 (SQL Server 15.0.200
Databases
Security
Server Objects
Replication
PolyBase
Always On High Availability
Management
Integration Services Catalogs
SQL Server Agent (Agent XPs disablec
XEvent Profiler
tablesAndTriggers....P-8AIVGG4\KR (82))
SQLQuery1.sql - D...P-8AIVGG4\KR (78))*
150 %
Messages
Execution plan
Query 5: Query cost (relative to the batch): 14%
UPDATE [posts] set [likes] = @1 WHERE [postId]=@2
Query 6: Query cost (relative to the batch): 14%
UPDATE [posts] set [likes] = @1 WHERE [postId]=@2
Query 7: Query cost (relative to the batch): 14%
UPDATE [posts] set [likes] = @1 WHERE [postId]=@2
Query executed successfully.
DESKTOP-8AIVGG4 (15.0 RTM) DESKTOP-8AIVGG4\KR (78) InstagramDB 00:00:00 0 rows
Ready
ENG
11:10 PM
05-Sep-20

12. **CORRELATED QUERY**

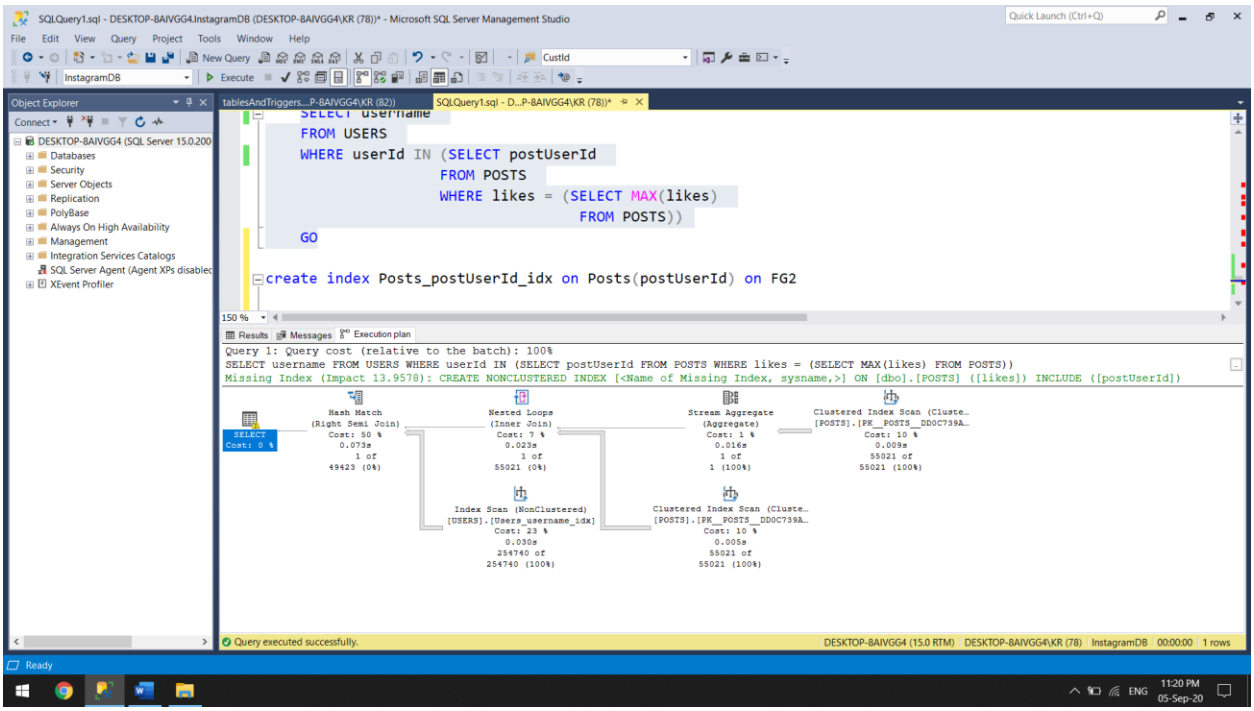
This query is to find the user whose post has the maximum number of likes.



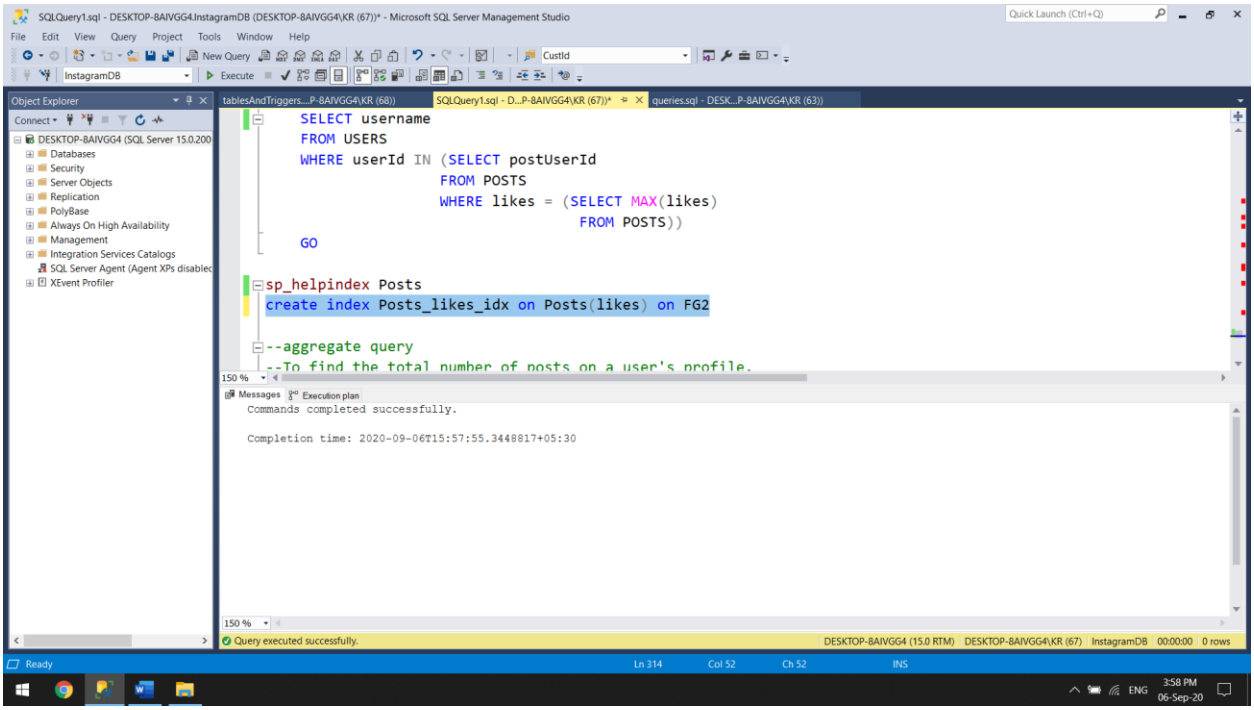
This is the execution plan of the query when the indexes on:

- Users is PK__USERS__CB9A1CFF004D37AB (clustered, unique, primary key located on FG1) and Users_username_idx (nonclustered located on FG2)
- Posts is PK__POSTS__DD0C739A88B15966 (clustered, unique, primary key located on FG1)

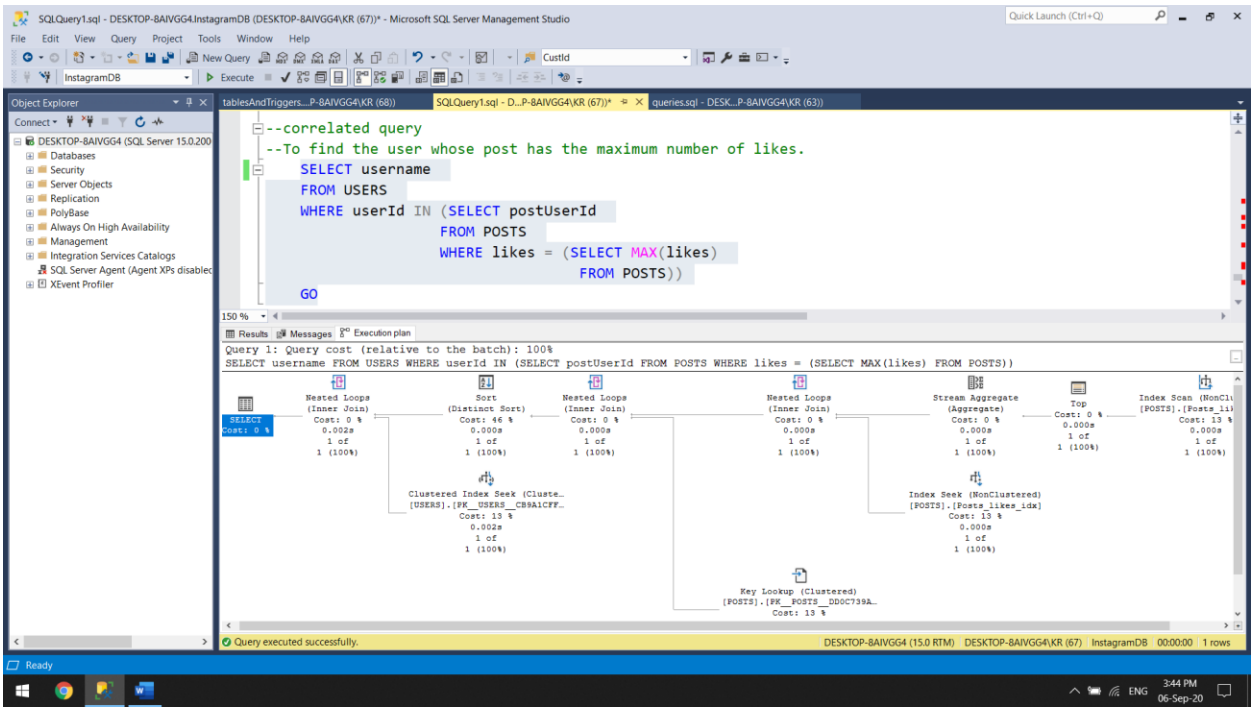
See that the execution plan shows (in green text) that since there is no non-clustered index on Posts based on the likes key, the impact on the execution of this query is 13.9578%.



As **suggested**, we create a non-clustered index Posts_likes_idx on Posts table with key as likes. It is stored in filegroup FG2.



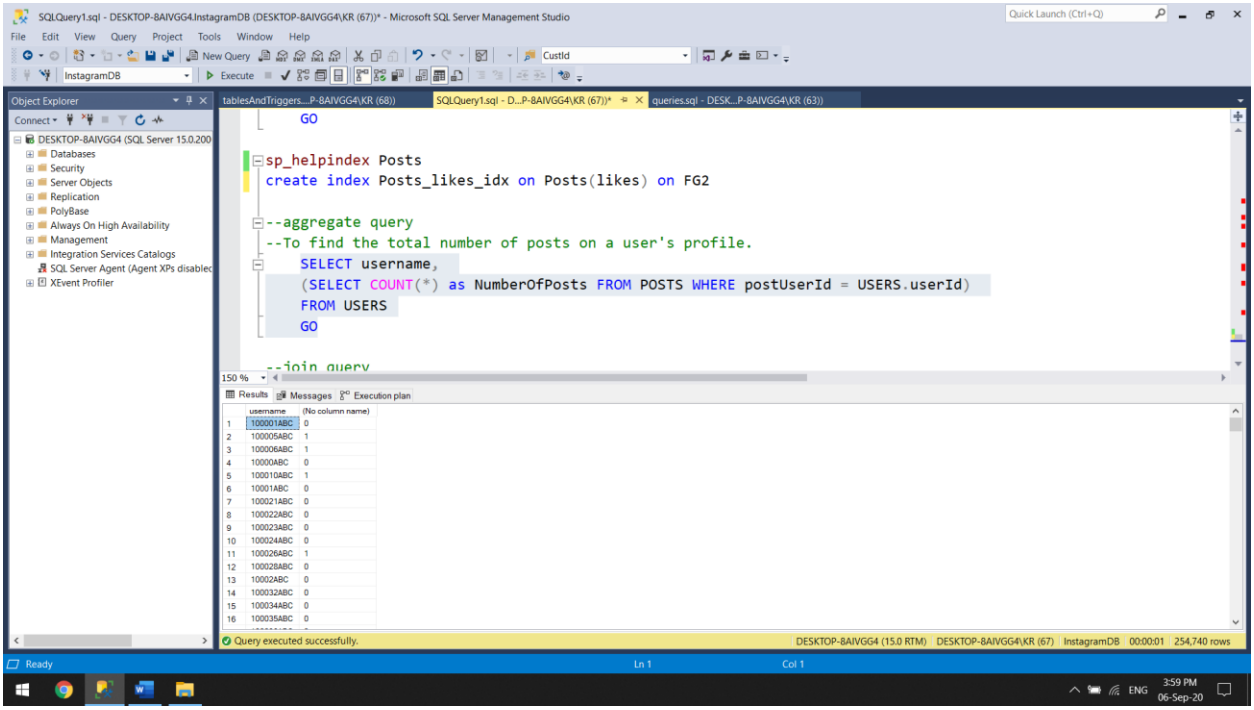
Upon re-executing the query we see that the **execution plan has improved**.



It is now seeking based clustered index of the primary key of Users table, i.e, PK__USERS__CB9A1CFF004D37AB rather than the non-clustered index Users_username_idx. Then a key lookup is done based on the primary key of Posts table, i.e, PK__USERS__CB9A1CFF004D37AB and then another index seek and scan which use the newly created non-clustered index Posts_likes_idx.

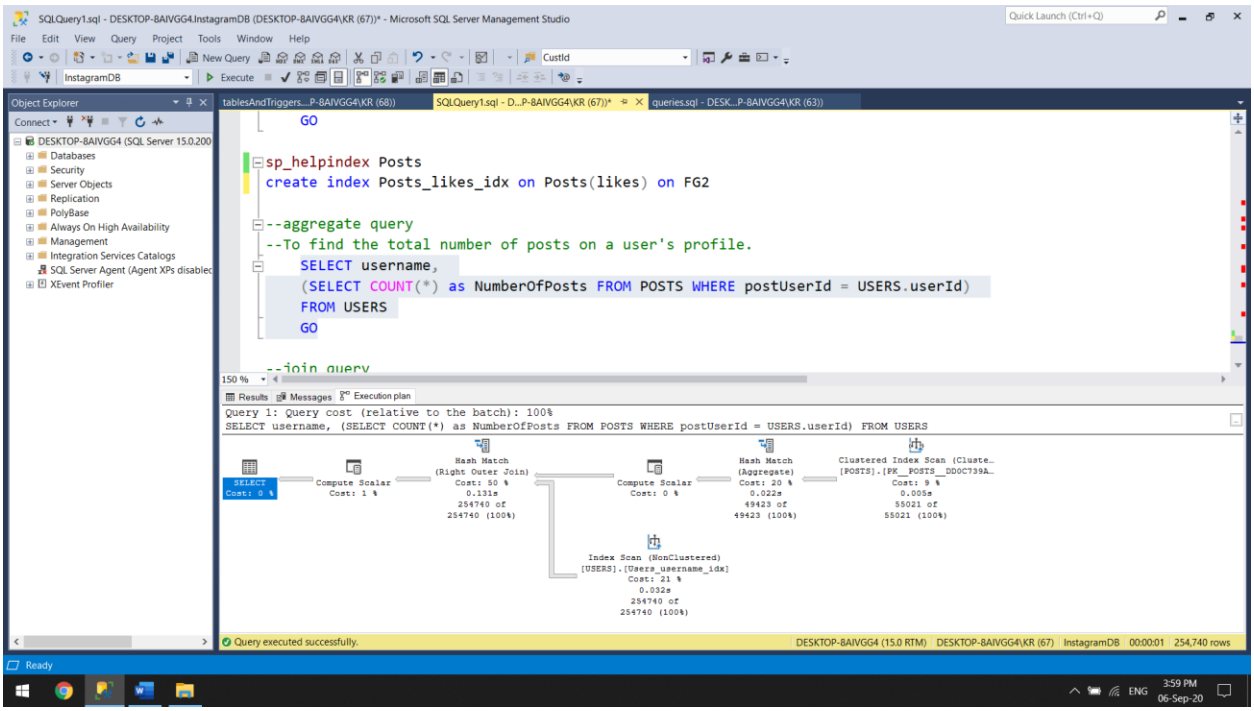
13. **AGGREGATE QUERY WITH EQUI-JOIN CONDITION**

This query is to find the total number of posts on a user's profile.

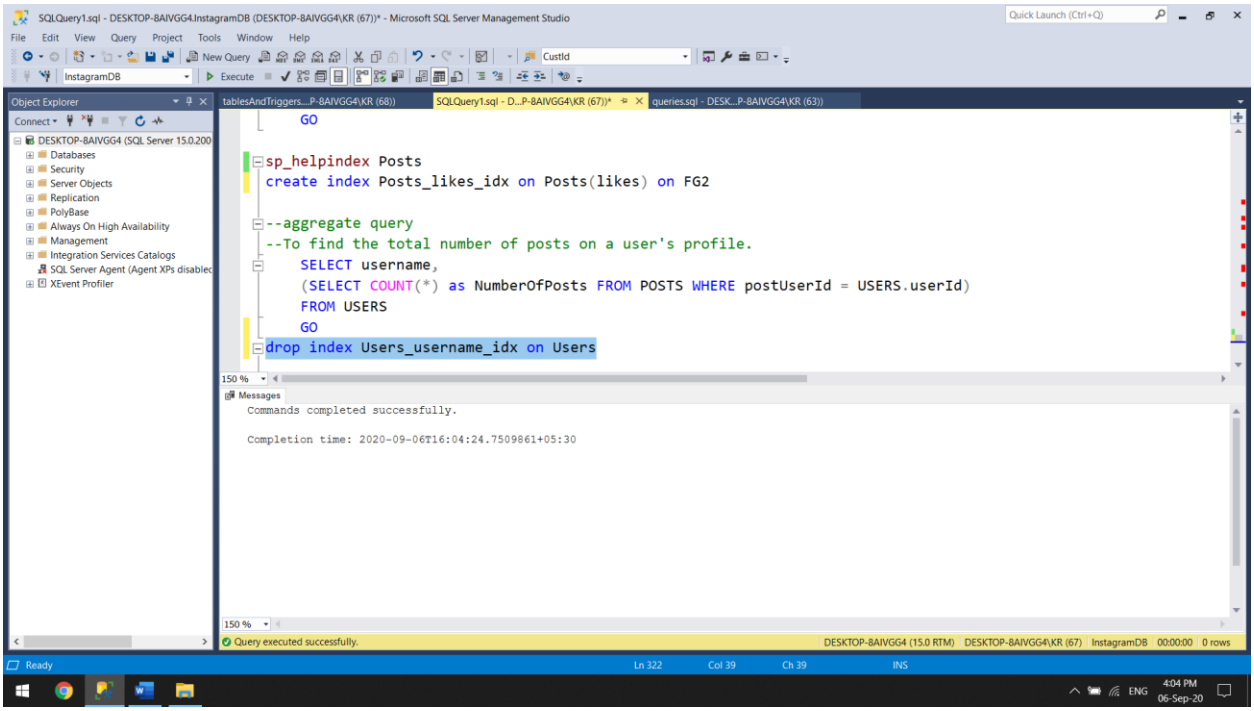


This is the execution plan of the query when the indexes on:

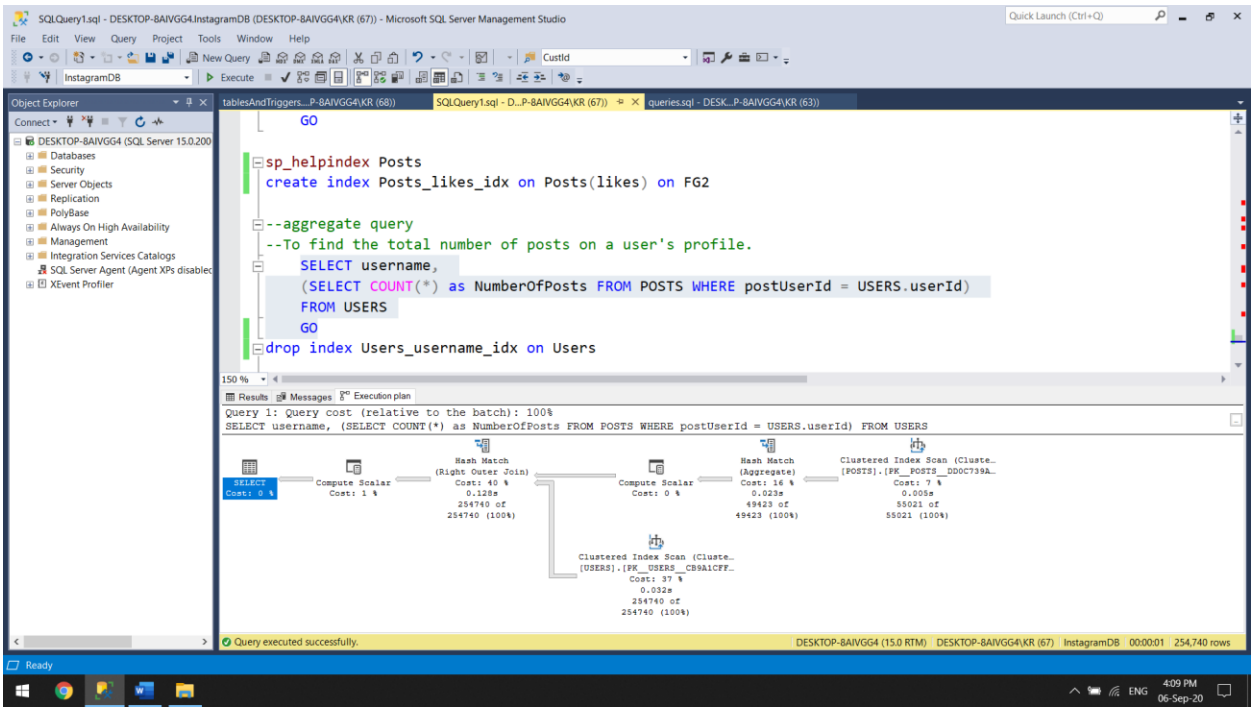
- Users is PK__USERS__CB9A1CFF004D37AB (clustered, unique, primary key located on FG1) and Users_username_idx (nonclustered located on FG2)
- Posts is PK__POSTS__DD0C739A88B15966 (clustered, unique, primary key located on FG1) and Posts_likes_idx (nonclustered located on FG2).

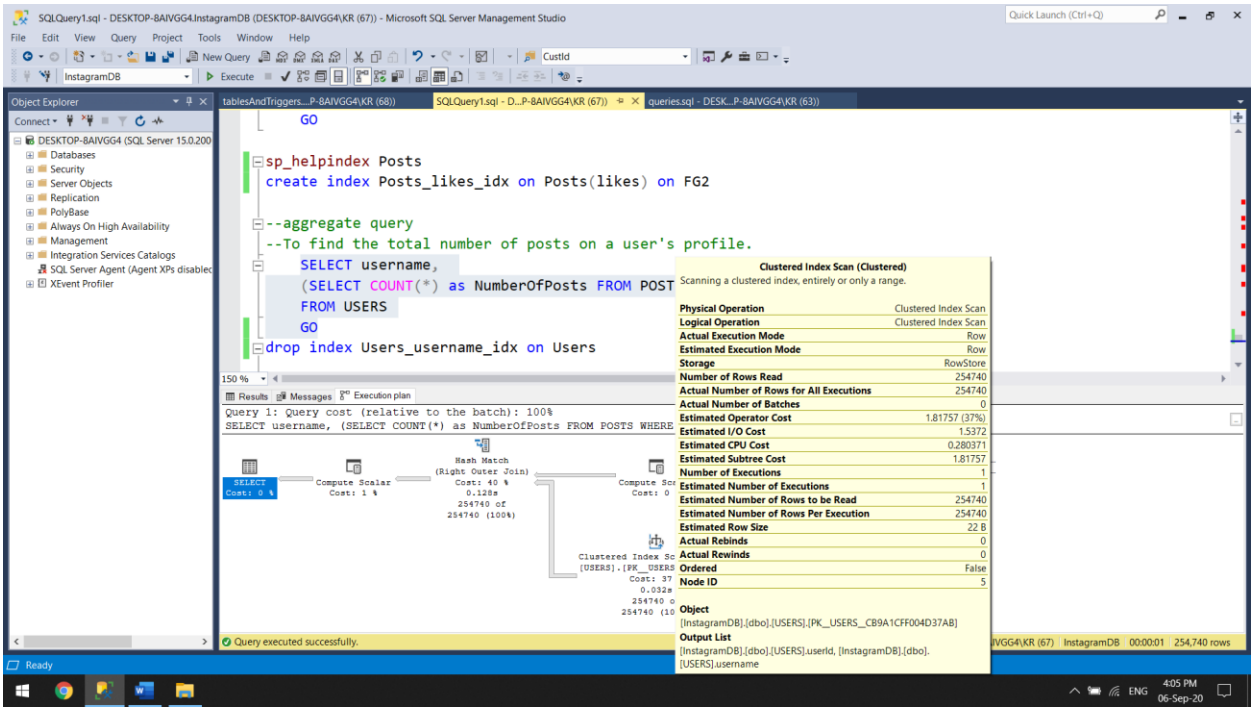


We will **drop the non-clustered index** Users_username_idx on Users to see how the cost changes.



Notice how the **cost** of index scan on Users **has gone up from 21% to 37%.**



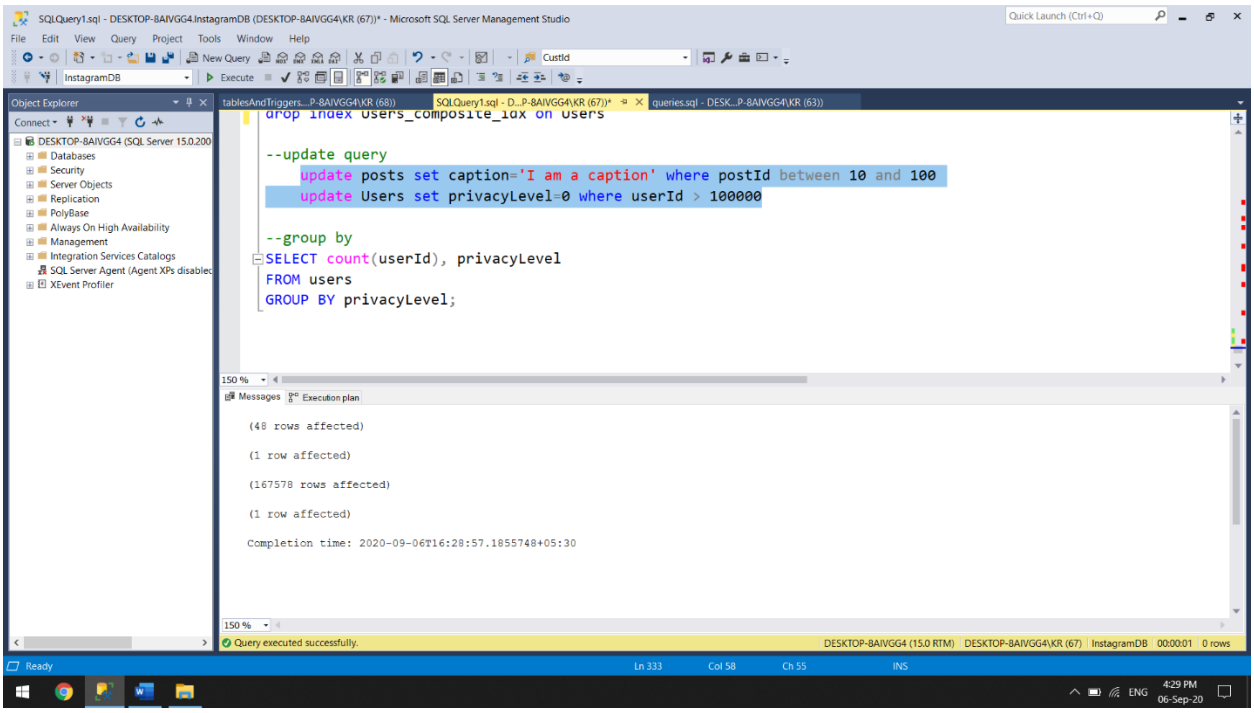


So we can conclude that **it was a bad idea to drop the Users_username_idx index. We create it again**.

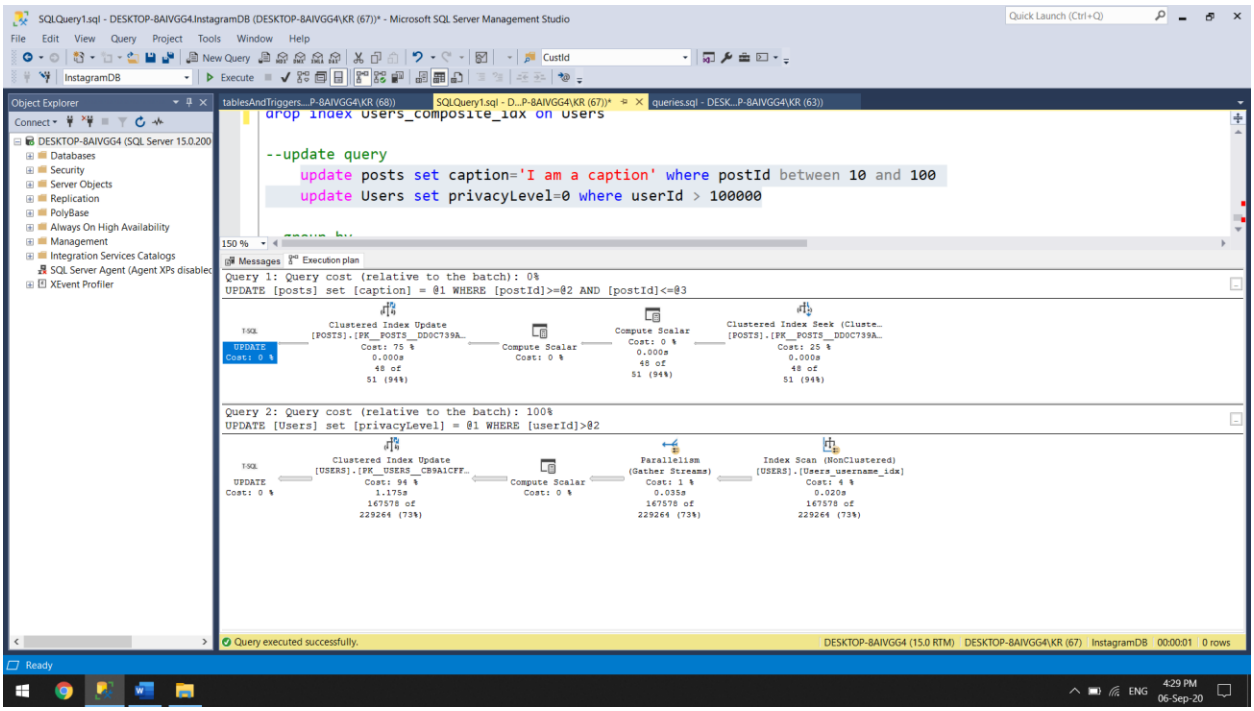
14. **UPDATE followed by GROUP BY**

Currently all the tuples in Posts have caption as ‘Caption’. So for a couple of tuples, we change the caption to a longer string.

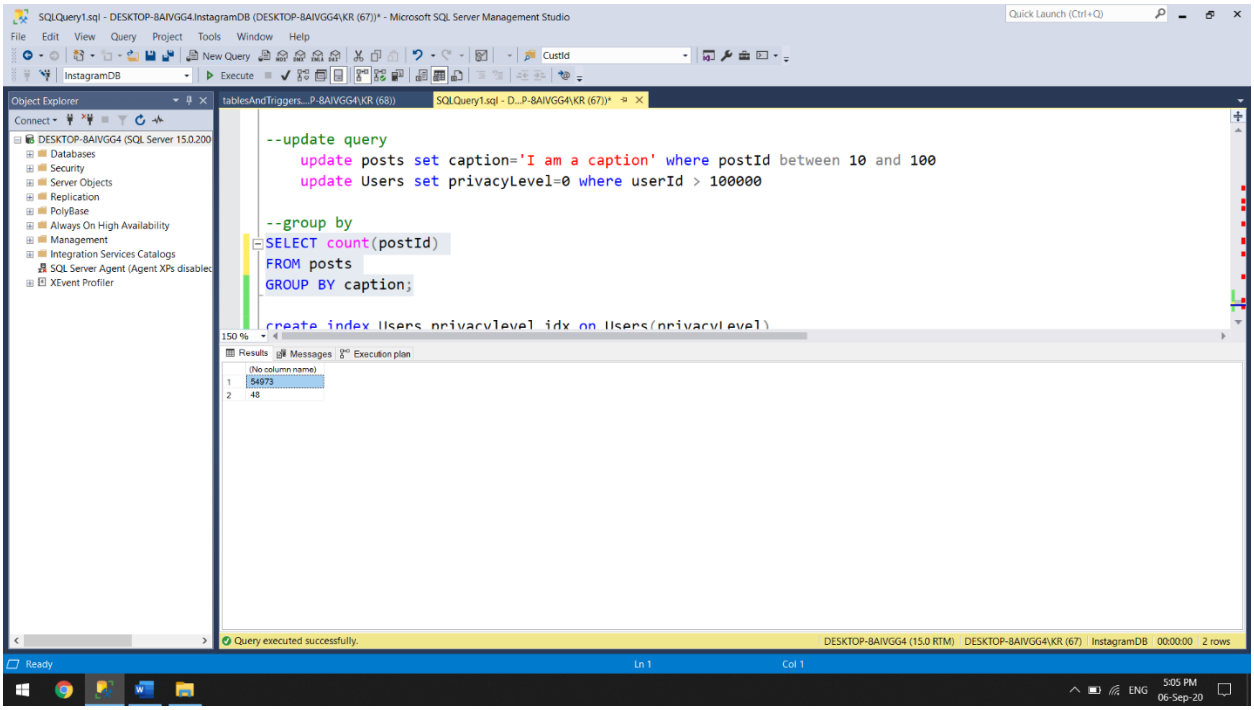
Currently all the tuples in Users have privacyLevel as 1. So for the tuples which have userId>100000 we update the privacyLevel bit to 0.



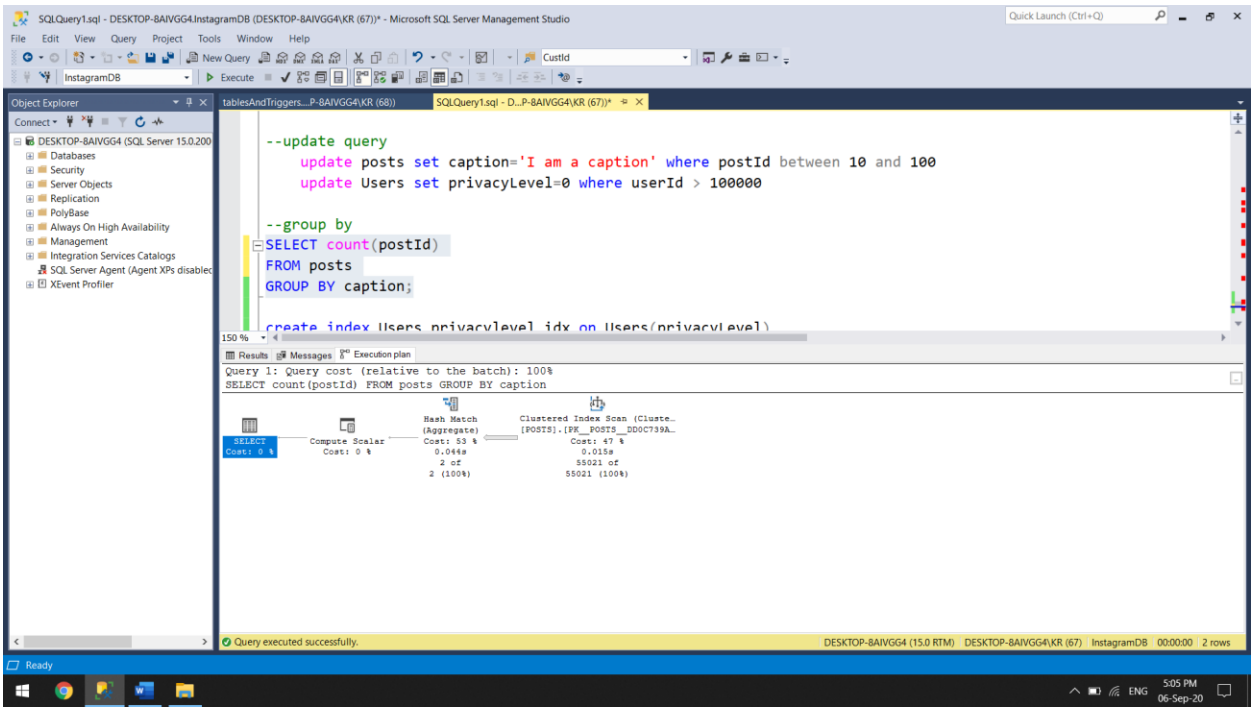
This is the execution plan of the update and set commands.



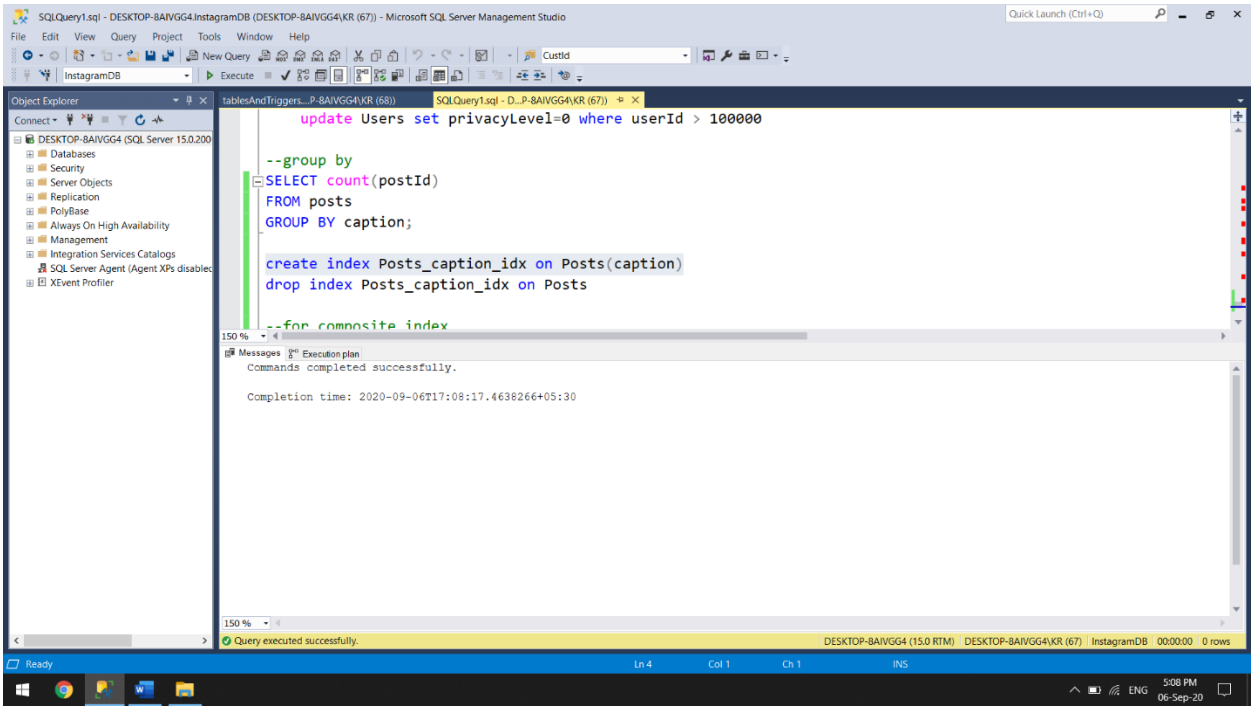
We want to get the count of posts with caption as ‘Caption’ and caption as ‘I am a caption’ separately. So we execute the below query.



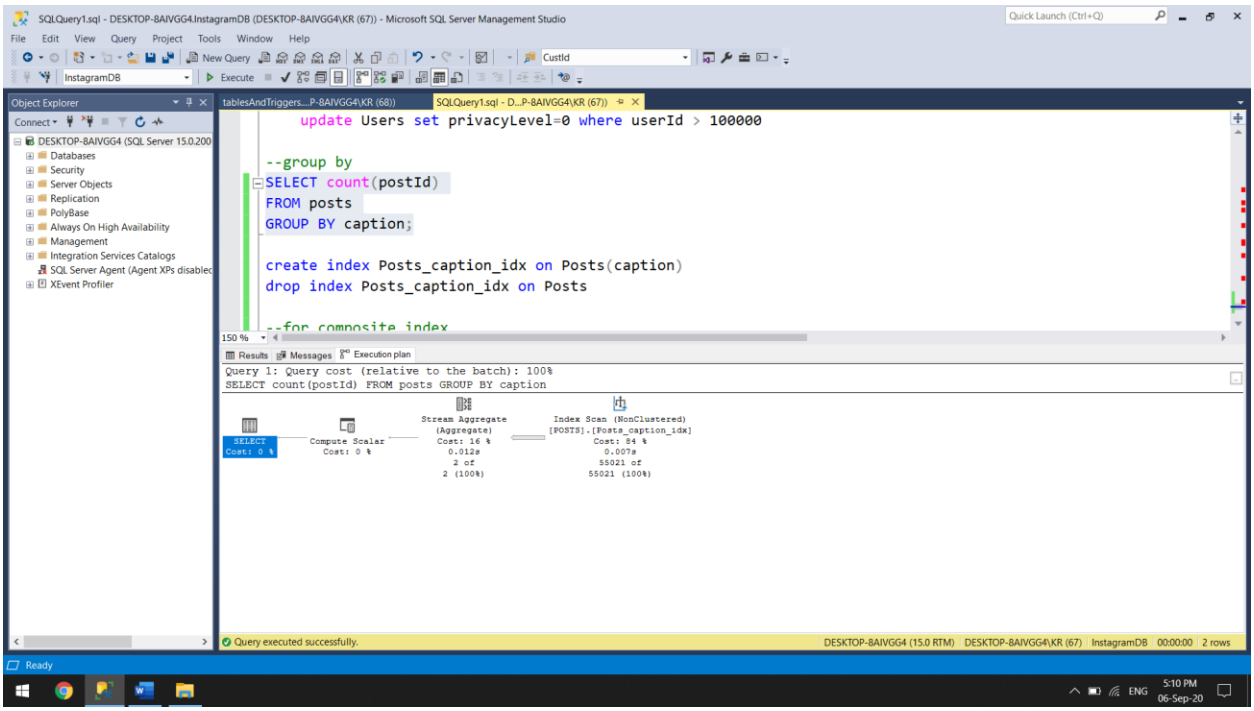
This is its execution plan.



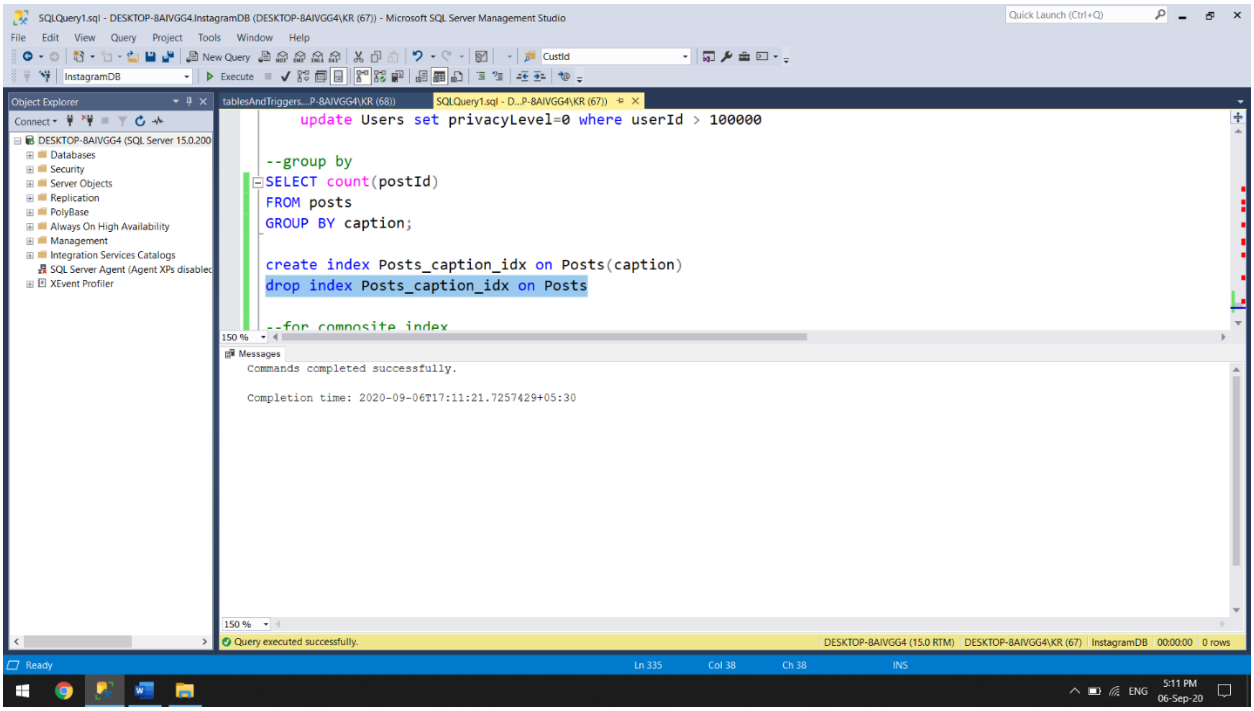
We try to see if creating an index on caption helps improve the performance. We create Posts_caption_idx on Posts.



Notice that the **cost has gone up instead of improving the performance with the newly created index**.

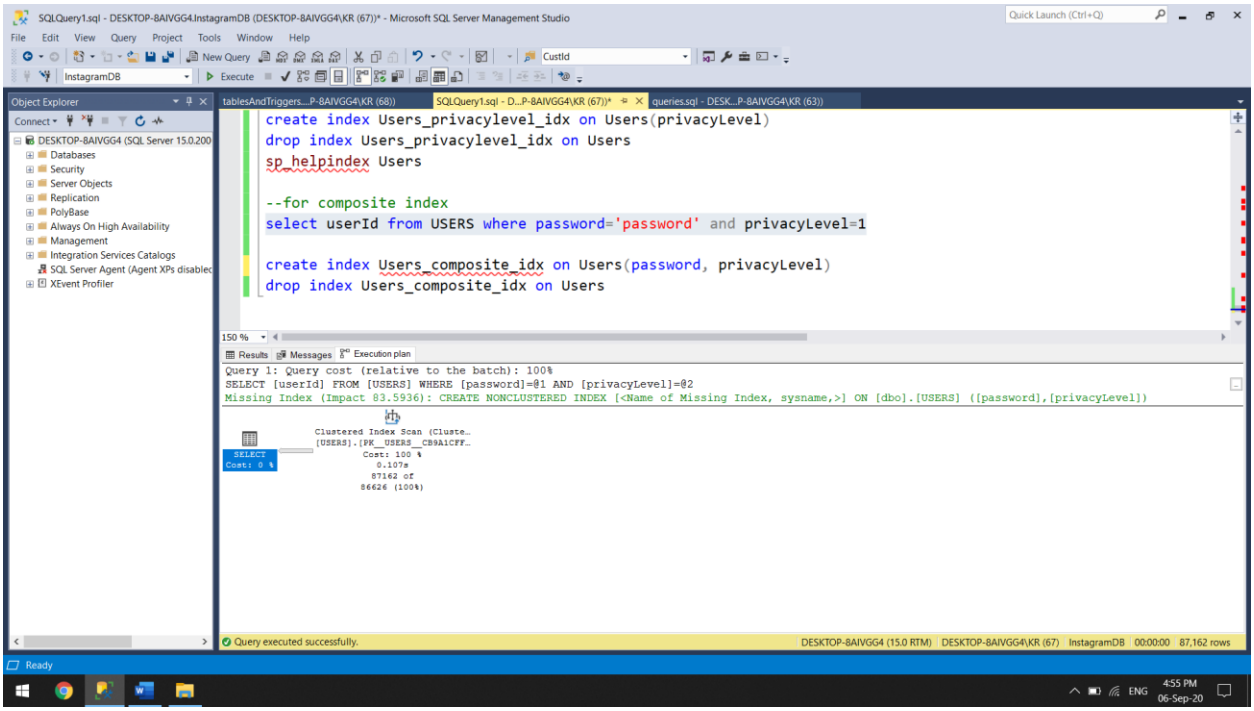


So we drop the index Posts_caption_idx.

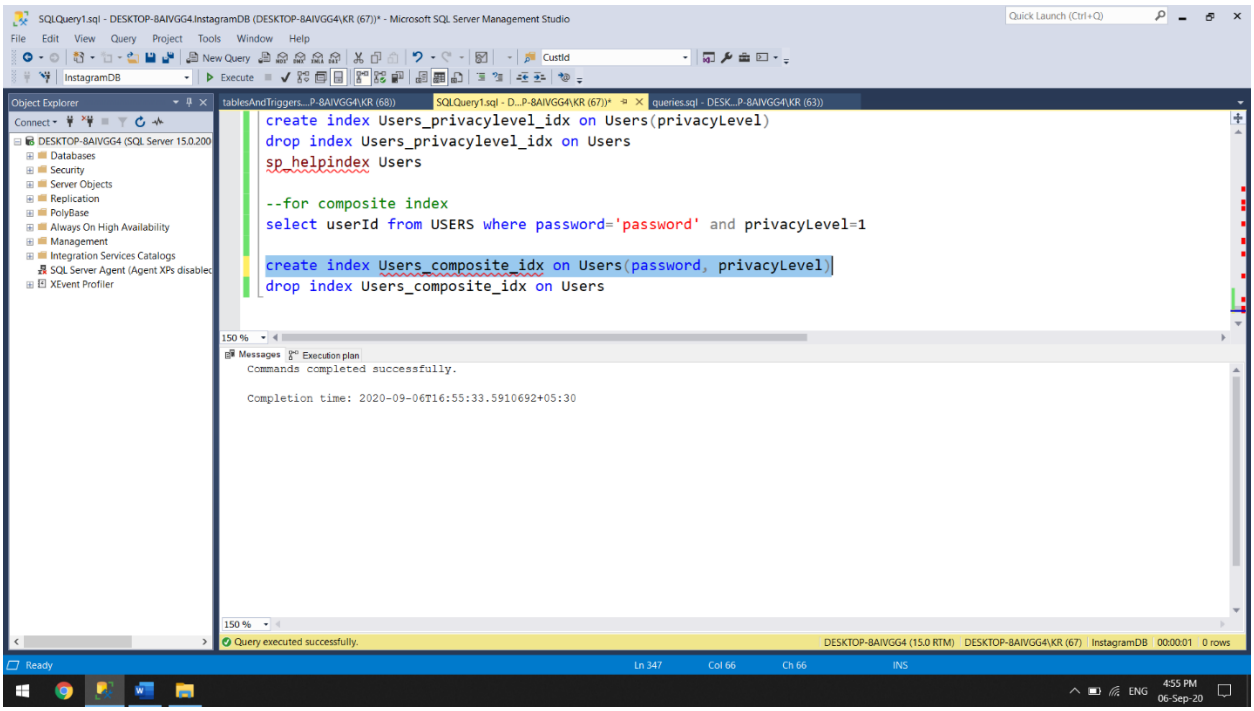


15. **SELECT QUERY**

The following is the execution plan when we try to retrieve the userId where the password is ‘password’ and the privacyLevel is set to 1. Notice the **suggestion** to create a non-clustered index on password and privacyLevel.



We create a **composite index of password and privacyLevel** on Users, call it Users_composite_index



Now notice that that the **execution plan has improved** by using this composite index.