

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

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 $\text{postId} \rightarrow \text{caption}, \text{latitude}, \text{longitude}, \text{timestamp}, \text{likes}, \text{comments}$ 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:
 - $\text{Att}(R1) \cup \text{Att}(R2) = \text{Att}(R)$
Union of **POSTS** and **POSTS_PICTURE** gives back the initial table
 - $\text{Att}(R1) \cap \text{Att}(R2) \neq \emptyset$
 $\text{Att}(R1) \cap \text{Att}(R2) = \text{postId}$
 - $\text{Att}(R1) \cap \text{Att}(R2) \rightarrow \text{Att}(R1) \text{ or } \text{Att}(R1) \cap \text{Att}(R2) \rightarrow \text{Att}(R2)$
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.

```
CREATE DATABASE [InstagramDB]
CONTAINMENT = NONE
ON PRIMARY
( NAME = N'InstagramDB', FILENAME = N'C:\Users\KR\Instagram_Disks\disk1\InstagramDB.mdf' , SIZE = 8192KB , FILE
LOG ON
( NAME = N'InstagramDB_log', FILENAME = N'C:\Users\KR\Instagram_Disks\disk2\InstagramDB_log.ldf' , SIZE = 8192K
GO
ALTER DATABASE [InstagramDB] SET COMPATIBILITY_LEVEL = 140
GO
ALTER DATABASE [InstagramDB] SET ANSI_NULL_DEFAULT OFF
GO
ALTER DATABASE [InstagramDB] SET ANSI_NULLS OFF
GO
```

Completion time: 2020-09-02T19:17:59.4655048+05:30

Query executed successfully.

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.

The screenshot shows the Microsoft SQL Server Management Studio interface. In the Object Explorer, the 'master' database is selected. In the center pane, a query window titled 'SQLQuery1.sql' is open, displaying T-SQL code to create filegroups and files in the 'InstagramDB' database.

```
IF NOT EXISTS (SELECT name FROM sys.filegroups WHERE is_default=1 AND name = N'PRIMARY') ALTER DATABASE [InstagramDB] ADD FILEGROUP [FG1]
GO

USE [master]
GO
ALTER DATABASE [InstagramDB] ADD FILEGROUP [FG2]
GO

USE [master]
GO
ALTER DATABASE [InstagramDB] ADD FILE ( NAME = N'F1', FILENAME = N'C:\Data\F1.ndf' , SIZE = 8192KB , FILEGROWTH=0)
GO
```

The status bar at the bottom indicates 'Query executed successfully.' and shows the completion time as 2020-09-02T19:19:15.9692883+05:30.

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.

The screenshot shows the Microsoft SQL Server Management Studio interface. In the Object Explorer, the 'master' database is selected. In the center pane, a query window titled 'SQLQuery1.sql' is open, displaying T-SQL code to add files F1 and F2 to the InstagramDB database, and to create a simple table named 'Customer'.

```
USE [master]
GO
ALTER DATABASE [InstagramDB] ADD FILE ( NAME = N'F1', FILENAME = N'C:\Users\KR\Instagram_Disks\disk1\F1.ndf'
GO
ALTER DATABASE [InstagramDB] ADD FILE ( NAME = N'F2', FILENAME = N'C:\Users\KR\Instagram_Disks\disk1\F2.ndf'
GO

USE [InstagramDB]
GO

CREATE TABLE Customer (CustId numeric(10) NOT NULL PRIMARY KEY,
CustName varchar(50) NOT NULL,
```

The status bar at the bottom indicates 'Query executed successfully.' and shows the completion time as 2020-09-02T19:21:09.6281521+05:30.

5. Create table Users on FG1

The screenshot shows the Microsoft SQL Server Management Studio interface. The Object Explorer on the left shows the database 'InstagramDB'. The central query window contains the following SQL code:

```
USE [InstagramDB]
GO

CREATE TABLE USERS(
    userId      INT          NOT NULL  PRIMARY KEY,
    username    VARCHAR(15)   NOT NULL  UNIQUE,
    firstName   VARCHAR(15)   NOT NULL,
    lastName    VARCHAR(15)   NOT NULL,
    email       VARCHAR(50)   NOT NULL
        CONSTRAINT email_format
        CHECK (email like '%@%.%'),
    bio         VARCHAR(15),
    password    VARCHAR(15)   NOT NULL,
    privacyLevel BIT         NOT NULL
)
ON FG1;
```

The status bar at the bottom indicates 'Query executed successfully.' and shows the completion time as 2020-09-02T19:23:28.6287131+05:30.

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

The screenshot shows the Microsoft SQL Server Management Studio interface. The Object Explorer on the left shows the database 'InstagramDB'. The central query window contains the following SQL code:

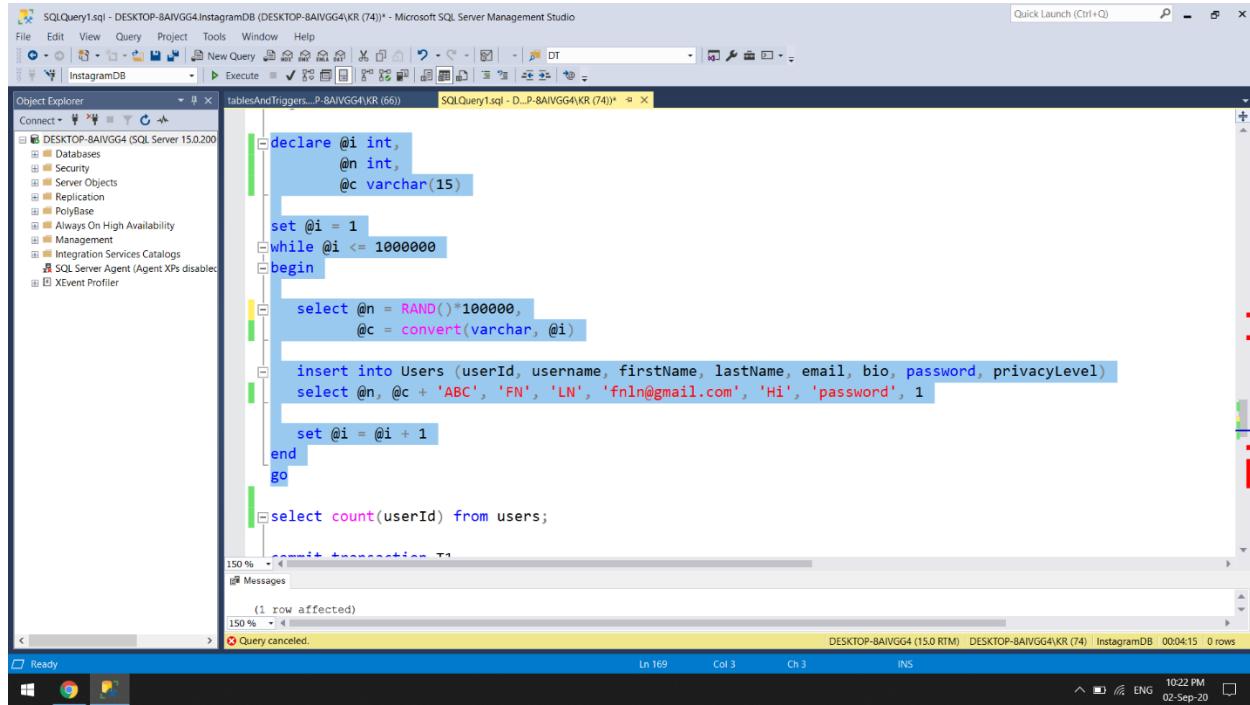
```
USE [InstagramDB]
GO

CREATE TABLE USERS(
    userId      INT          NOT NULL  PRIMARY KEY,
    username    VARCHAR(15)   NOT NULL  UNIQUE,
    firstName   VARCHAR(15)   NOT NULL,
    lastName    VARCHAR(15)   NOT NULL,
    email       VARCHAR(50)   NOT NULL
        CONSTRAINT email_format
        CHECK (email like '%@%.%'),
    bio         VARCHAR(15),
    password    VARCHAR(15)   NOT NULL,
    privacyLevel BIT         NOT NULL
)
ON FG1;

create index Users_username_idx on Users(username) ON FG2;
```

The status bar at the bottom indicates 'Query executed successfully.' and shows the completion time as 2020-09-02T19:25:53.0984853+05:30.

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.



```
declare @i int,
        @n int,
        @c varchar(15)

set @i = 1
while @i <= 1000000
begin

    select @n = RAND()*100000,
           @c = convert(varchar, @i)

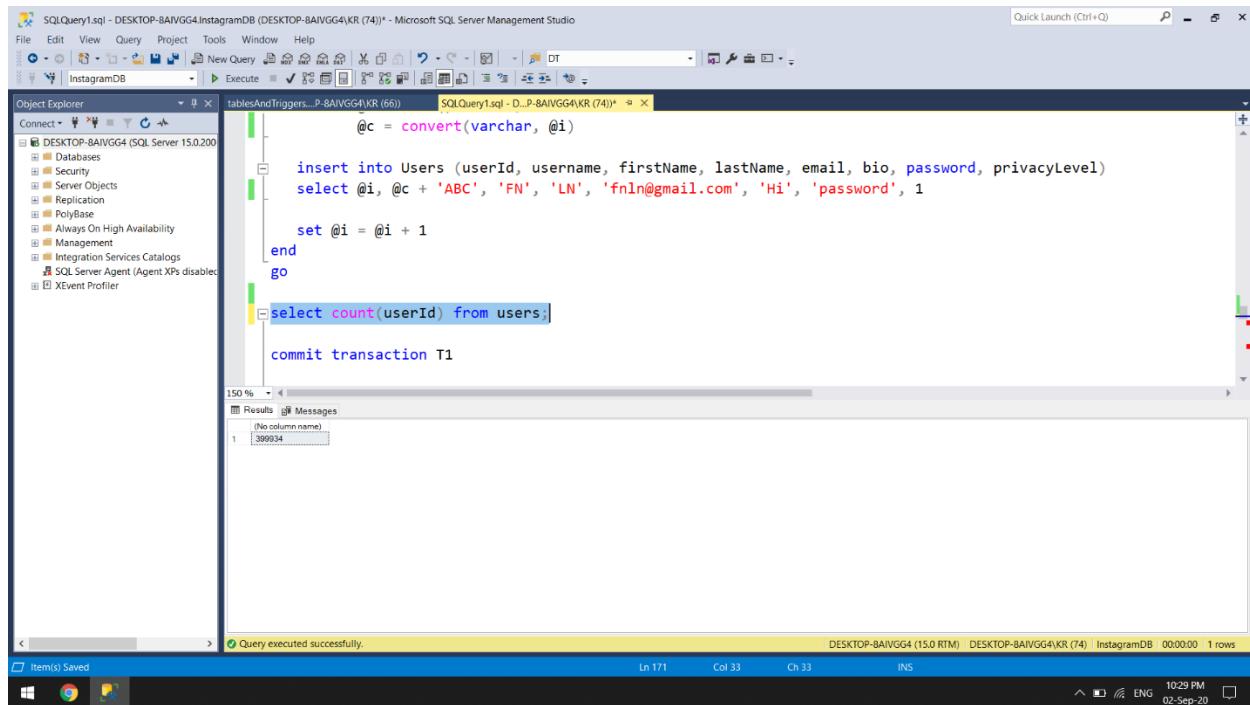
    insert into Users (userId, username, firstName, lastName, email, bio, password, privacyLevel)
    select @n, @c + 'ABC', 'FN', 'LN', 'fnln@gmail.com', 'Hi', 'password', 1

    set @i = @i + 1
end
go

select count(userId) from users;
```

The screenshot shows the Microsoft SQL Server Management Studio interface. The Object Explorer on the left shows the database structure. The main window contains a T-SQL script for generating 1,000,000 rows in the 'Users' table. The script uses a loop to generate random user IDs and names, and inserts them into the table with fixed values for other columns. The results pane shows a single row returned by the final 'select' statement, indicating 1 row affected.

8. We have loaded 399934 rows.



```
declare @c = convert(varchar, @i)

insert into Users (userId, username, firstName, lastName, email, bio, password, privacyLevel)
select @i, @c + 'ABC', 'FN', 'LN', 'fnln@gmail.com', 'Hi', 'password', 1

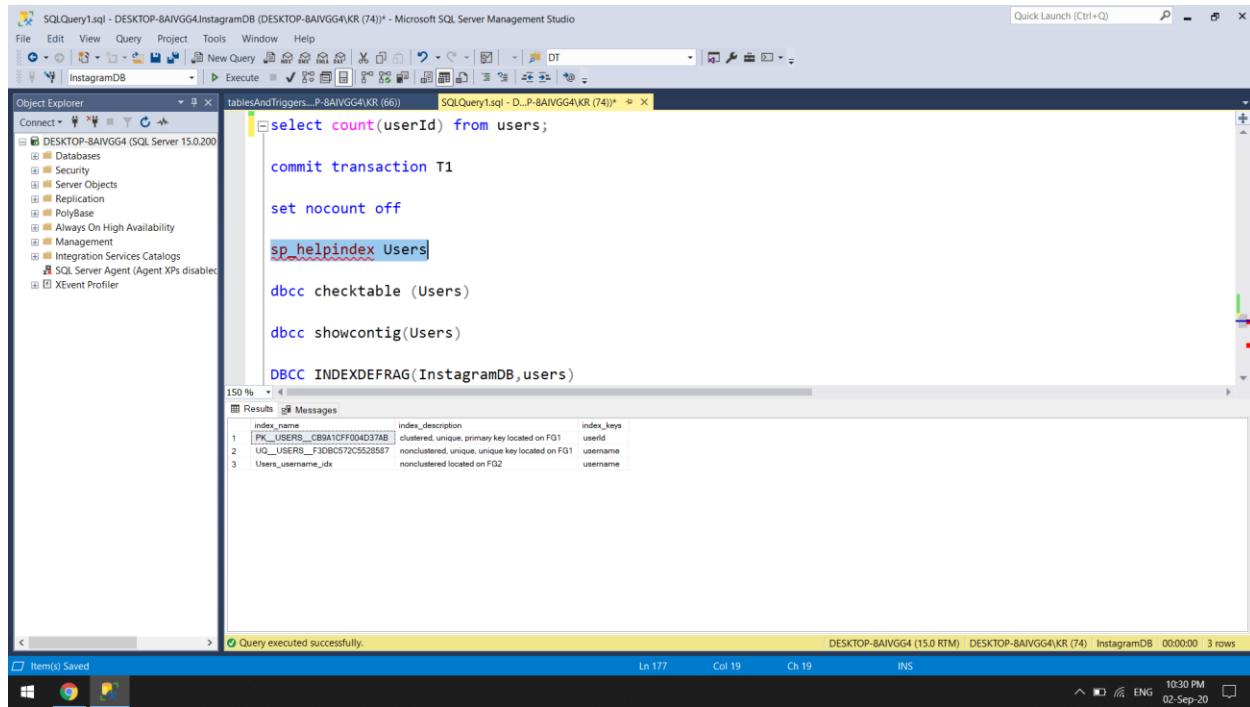
set @i = @i + 1
end
go

select count(userId) from users;

commit transaction T1
```

The screenshot shows the Microsoft SQL Server Management Studio interface after the query has been executed. The results pane displays the count of rows inserted, which is 399934. A message at the bottom indicates that the query was executed successfully.

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.



The screenshot shows the Microsoft SQL Server Management Studio interface. In the Object Explorer, the database 'InstagramDB' is selected. In the center pane, a query window displays the following T-SQL code:

```
select count(userId) from users;

commit transaction T1

set nocount off

sp_helpindex Users

dbcc checktable (Users)

dbcc showcontig(Users)

DBCC INDEXDEFRAG(InstagramDB,users)
```

The results pane shows a table titled 'Messages' with three rows of index information:

index_name	index_description	index_keys
1 PK_USERS_CB9A1CFF004D37AB	clustered, unique, primary key located on FG1	userId
2 UX_USERS_F30BC572C5528587	nonclustered, unique, unique key located on FG1	username
3 Users_username_idx	nonclustered located on F02	username

At the bottom of the results pane, a message says 'Query executed successfully.' The status bar at the bottom right shows the date and time: '10:30 PM 02-Sep-20'.

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.

```

SQLQuery1.sql - DESKTOP-8AIVGG4.InstagramDB (DESKTOP-8AIVGG4\KR (74)) - Microsoft SQL Server Management Studio
File Edit View Query Project Tools Window Help
New Query DT Execute
Object Explorer
Connect Databases Security Server Objects Replication PolyBase Always On High Availability Management Integration Services Catalogs SQL Server Agent (Agent XPs disabled) XEvent Profiler
InstagramDB
tablesAndTriggers_P-8AIVGG4(KR (66)) SQLQuery1.sql - D-P-8AIVGG4(KR (74))*
select count(userId) from users;
commit transaction T1
set nocount off
sp_helpindex Users
dbcc checktable (Users)
dbcc showcontig(Users)
DBCC INDEXDEFrag(InstagramDB,users)

150 % 
Messages
DBCC results for 'USERS'.
There are 399934 rows in 4645 pages for object "USERS".
DBCC execution completed. If DBCC printed error messages, contact your system administrator.

Completion time: 2020-09-02T22:30:14.9563253+05:30

```

Query executed successfully.

LN 179 COL 24 CH 24 INS

DESKTOP-8AIVGG4 (15.0 RTM) DESKTOP-8AIVGG4\KR (74) InstagramDB 00:00:00 0 rows

Item(s) Saved

10:30 PM ENG 02-Sep-20

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

```

SQLQuery1.sql - DESKTOP-8AIVGG4.InstagramDB (DESKTOP-8AIVGG4\KR (74)) - Microsoft SQL Server Management Studio
File Edit View Query Project Tools Window Help
New Query DT Execute
Object Explorer
Connect Databases Security Server Objects Replication PolyBase Always On High Availability Management Integration Services Catalogs SQL Server Agent (Agent XPs disabled) XEvent Profiler
InstagramDB
tablesAndTriggers_P-8AIVGG4(KR (66)) SQLQuery1.sql - D-P-8AIVGG4(KR (74))*
select count(userId) from users;
commit transaction T1
set nocount off
sp_helpindex Users
dbcc checktable (Users)
dbcc showcontig(Users)
DBCC INDEXDEFrag(InstagramDB,users)

150 % 
Messages
DBCC SHOWCONTIG scanning 'USERS' table...
Table: 'USERS' (581577110); index ID: 1, database ID: 6
TABLE level scan performed.
- Pages Scanned.....: 4645
- Extents Scanned....: 583
- Extent Switches.....: 4644
- Avg. Pages per Extent.....: 8.0
- Scan Density [Best Count:Actual Count].....: 12.51% [581:4645]
- Logical Scan Fragmentation .....: 99.03%
- Extent Scan Fragmentation .....: 40.99%
- Avg. Bytes Free per Page.....: 2523.2
- Avg. Page Density (full).....: 68.83%
DBCC execution completed. If DBCC printed error messages, contact your system administrator.

150 % 

```

Query executed successfully.

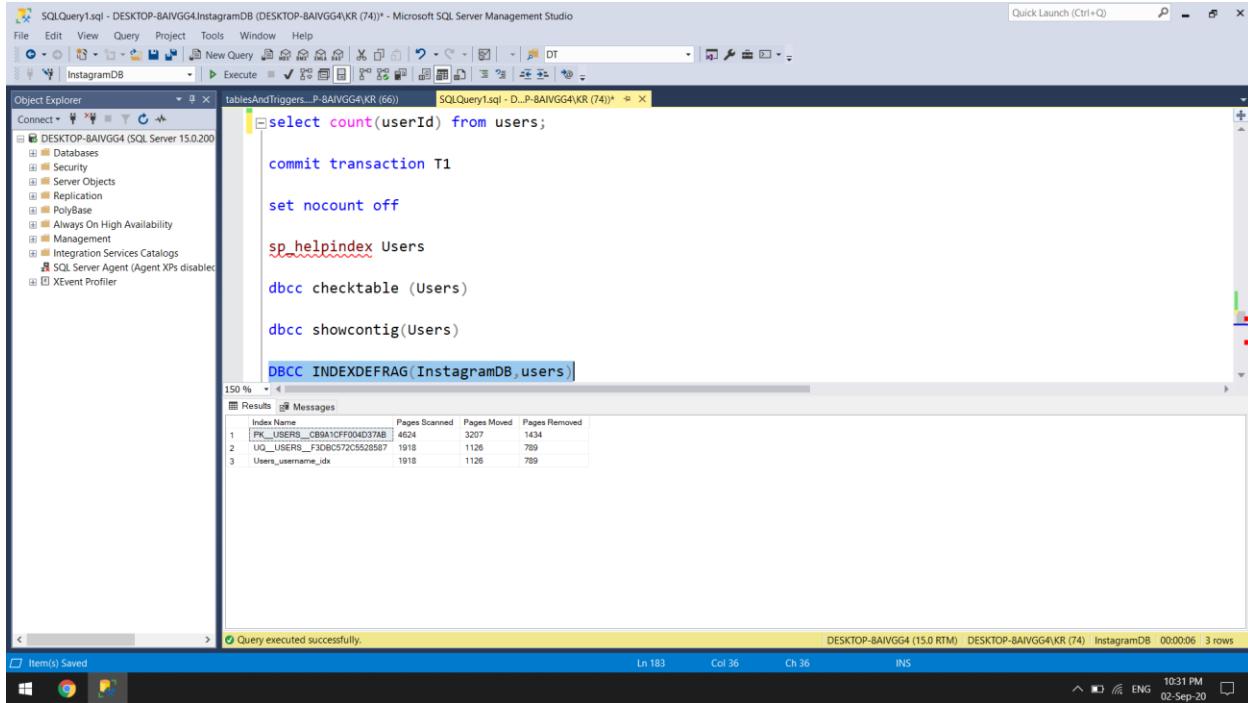
LN 181 COL 23 CH 23 INS

DESKTOP-8AIVGG4 (15.0 RTM) DESKTOP-8AIVGG4\KR (74) InstagramDB 00:00:00 0 rows

Item(s) Saved

10:30 PM ENG 02-Sep-20

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.



The screenshot shows the Microsoft SQL Server Management Studio interface. In the Object Explorer, a connection to 'DESKTOP-8AIVGG4 (SQL Server 15.0.200)' is selected, with 'Databases' expanded. In the center pane, a query window titled 'tablesAndTriggers...P-8AIVGG4(KR) (66)' contains the following T-SQL code:

```
select count(userId) from users;

commit transaction T1

set nocount off

sp_helpindex Users

dbcc checktable (Users)

dbcc showcontig(Users)

DBCC INDEXDEFrag(InstagramDB.users)
```

The 'Results' tab is active, displaying the output of the 'DBCC INDEXDEFrag' command. The results table shows the following data:

Index Name	Pages Scanned	Pages Moved	Pages Removed
PK__Users__CB9A1CFF004D37AB	4124	3207	1434
UGL__Users__F30B572C5528587	1918	1126	789
Users_username_idx	1918	1126	789

At the bottom of the results pane, a message states 'Query executed successfully.' The status bar at the bottom right shows 'DESKTOP-8AIVGG4 (15.0 RTM) DESKTOP-8AIVGG4(KR) InstagramDB 00:00:06 3 rows' and the date '02-Sep-20'.

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.

```
set nocount off

sp_helpindex Users

dbcc checktable (Users)

dbcc showcontig(Users)

DBCC INDEXDEFRAG(InstagramDB,users)

update Users set bio = 'Hey there!'
where userId between 200000 and 300000
```

(40748 rows affected)

Completion time: 2020-09-02T22:40:03.6263619+05:30

Query executed successfully.

```
delete from Users where userId between 30000 and 40000

delete from Users where userId between 130000 and 140000

delete from Users where userId between 430000 and 440000

delete from Users where userId between 530000 and 540000

dbcc showcontig(Users)

DBCC INDEXDEFRAG(InstagramDB,Users)
```

(4051 rows affected)

(4117 rows affected)

(3951 rows affected)

(3958 rows affected)

Completion time: 2020-09-02T22:42:28.4507205+05:30

Query executed successfully.

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

```

delete from Users where userId between 30000 and 40000
delete from Users where userId between 130000 and 140000
delete from Users where userId between 430000 and 440000
delete from Users where userId between 530000 and 540000
dbcc showcontig(Users)
DBCC INDEXDEFRAG(InstagramDB,Users)

```

Messages

```

DBCC SHOWCONTIG scanning 'USERS' table...
Table: 'USERS' (581577110); index ID: 1, database ID: 6
TABLE level scan performed.
- Pages Scanned.....: 3411
- Extents Scanned.....: 432
- Extent Switches.....: 999
- Avg. Pages per Extent.....: 7.9
- Scan Density [Best Count:Actual Count].....: 42.70% [427:1000]
- Logical Scan Fragmentation .....: 19.29%
- Extent Scan Fragmentation .....: 40.74%
- Avg. Bytes Free per Page.....: 716.6
- Avg. Page Density (full).....: 91.15%
DBCC execution completed. If DBCC printed error messages, contact your system administrator.

```

Query executed successfully.

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.

Index Name	Pages Scanned	Pages Moved	Pages Removed
1 PK_USERS_CBA1CF004D37AB	3378	2725	285
2 UG__USERs_F3DB6C572C552B567	1124	1001	5
3 Users_username_dx	1126	843	3

Query executed successfully.

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

The screenshot shows the Microsoft SQL Server Management Studio interface. The title bar reads "SQLQuery1.sql - DESKTOP-BAIVGG4.InstagramDB (DESKTOP-BAIVGG4(KR (74)) - Microsoft SQL Server Management Studio". The left pane is the Object Explorer, showing the connection to "DESKTOP-BAIVGG4 (SQL Server 15.0.200)" and its databases, including "InstagramDB". The right pane contains a query window with the following T-SQL code:

```
delete from Users where userId between 430000 and 440000

delete from Users where userId between 530000 and 540000

dbcc showcontig(Users)

DBCC INDEXDEFRAG(InstagramDB,Users)

dbcc checktable(Users)

select Db_name (database_id), * from sys.dm_os_buffer_descriptors
order by 1, file_id, page_id
```

Below the code, the status bar indicates "150 %". Under the "Messages" section, the output is:

```
DBCC results for 'USER$'.
There are 383857 rows in 3126 pages for object "USERS".
DBCC execution completed. If DBCC printed error messages, contact your system administrator.
```

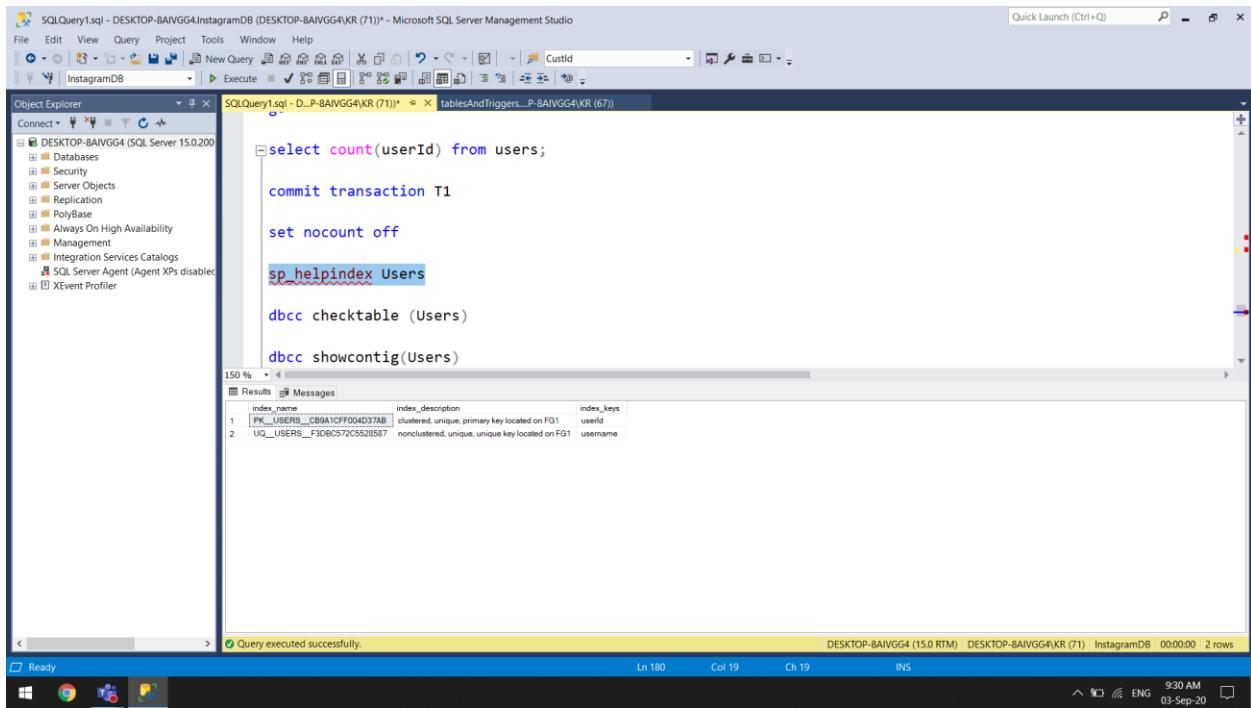
The completion time is listed as "Completion time: 2020-09-02T22:43:58.0666402+05:30". The bottom status bar shows "DESKTOP-BAIVGG4 (15.0 RTM) DESKTOP-BAIVGG4(KR (74)) InstagramDB 00:00:00 0 rows".

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.



The screenshot shows the Microsoft SQL Server Management Studio interface. A query window is open with the following T-SQL code:

```
select count(userId) from users;

commit transaction T1

set nocount off

sp_helpindex Users

dbcc checktable (Users)

dbcc showcontig(Users)
```

The results grid displays the following information:

index_name	index_description	index_keys
PK_USERS_CB9A1CFF004D37AB	clustered; unique, primary key located on FG1	userId
UQ_USERS_F30B0572C5528587	nonclustered, unique, unique key located on FG1	username

Below the results grid, a message bar indicates: "Query executed successfully." The status bar at the bottom shows: DESKTOP-8AIVGG4 (15.0 RTM) | DESKTOP-8AIVGG4\KR (71) | InstagramDB | 00:00:00 | 2 rows.

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.

```

SQLQuery1.sql - DESKTOP-8AIVGG4.InstagramDB (DESKTOP-8AIVGG4(KR (64)) - Microsoft SQL Server Management Studio
File Edit View Query Project Tools Window Help
New Query Execute
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 disabled) XEvent Profiler
SQLQuery1.sql - D-P-8AIVGG4(KR (64)* - tablesAndTriggers_P-8AIVGG4(KR (60))
GO
bio      VARCHAR(15),
password  VARCHAR(15) NOT NULL,
privacyLevel  BIT      NOT NULL
ON UsersRangePS1 (userId)
GO
alter table Users drop constraint UQ__USERS__F3DBC572C5528587
set nocount on
select @@TRANCOUNT
150 %
Messages
Commands completed successfully.

Completion time: 2020-09-04T17:31:43.6567297+05:30

```

Query executed successfully.

LN 139 Col 62 Ch 62 INS

DESKTOP-8AIVGG4 (15.0 RTM) DESKTOP-8AIVGG4(KR (64) InstagramDB 00:00:00 | 0 rows

Ready 5:32 PM ENG 04-Sep-20

3. The current indexes on Users table

```

SQLQuery1.sql - DESKTOP-8AIVGG4.InstagramDB (DESKTOP-8AIVGG4(KR (64)) - Microsoft SQL Server Management Studio
File Edit View Query Project Tools Window Help
New Query Execute
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 disabled) XEvent Profiler
SQLQuery1.sql - D-P-8AIVGG4(KR (64)* - tablesAndTriggers_P-8AIVGG4(KR (60))
--SELECT username, COUNT(username) AS dup_count FROM Users GROUP BY username HAVING (COUNT(username) > 1)

commit transaction T1

set nocount off

sp_helpindex Users

dbcc checktable (Users)

dbcc showcontig(Users)

150 %
Messages

```

index_name	index_description	index_keys
PK_USERS_CB9A1CF004D3TAB	clustered, unique, primary key located on FG1	userid

Query executed successfully.

LN 172 Col 19 Ch 19 INS

DESKTOP-8AIVGG4 (15.0 RTM) DESKTOP-8AIVGG4(KR (64) InstagramDB 00:00:00 | 1 rows

Ready 5:32 PM ENG 04-Sep-20

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

The screenshot shows the Microsoft SQL Server Management Studio interface. In the Object Explorer, the database 'InstagramDB' is selected. In the center pane, a query window titled 'SQLQuery1.sql' contains the following T-SQL code:

```
--create index Users_username_idx on Users(username) ON FG2;

create PARTITION FUNCTION UsersRangePF1 (int)
    AS RANGE LEFT FOR VALUES (150000);
GO

create PARTITION SCHEME UsersRangePS1
    AS PARTITION UsersRangePF1
    TO (FG1, FG2);
GO

CREATE TABLE UsersPartitionTable (userId      INT        NOT NULL PRIMARY KEY,
                                 username   NVARCHAR(50) NOT NULL,
                                 bio        NVARCHAR(500),
                                 location  NVARCHAR(50),
                                 profilePic BLOB);

-- Create a clustered index on the partitioned table
CREATE CLUSTERED INDEX CI_UsersPartitionTable ON UsersPartitionTable (userId);
```

The execution completed successfully with the message: 'Commands completed successfully.' The completion time was 2020-09-04T17:33:51.2803409+05:30.

This screenshot is identical to the one above, showing the same T-SQL code being executed in the SQL Server Management Studio. The output shows the command completed successfully with the same timestamp: 2020-09-04T17:33:59.6546023+05:30.

5. Create the partition table schema.

```
CREATE TABLE UsersPartitionTable (userId INT NOT NULL PRIMARY KEY,
username VARCHAR(15) NOT NULL,
firstName VARCHAR(15) NOT NULL,
lastName VARCHAR(15) NOT NULL,
email VARCHAR(50) NOT NULL,
bio VARCHAR(15),
password VARCHAR(15) NOT NULL,
privacyLevel BIT NOT NULL)
ON UsersRangePS1 (userId)
GO
```

Completion time: 2020-09-04T17:34:51.3992305+05:30

Query executed successfully.

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

```
bio VARCHAR(15),
password VARCHAR(15) NOT NULL,
privacyLevel BIT NOT NULL)
ON UsersRangePS1 (userId)
GO

create index Users_username_idx on Users(username) ON FG2;

set nocount on

select @@TRANCOUNT
```

Completion time: 2020-09-04T17:41:16.2684493+05:30

Query executed successfully.

7. Add random tuples into the Users table

The screenshot shows the Microsoft SQL Server Management Studio interface. The query window displays the following T-SQL code:

```

declare @i int,
        @n int,
        @c varchar(15)

set @i = 1
while @i <= 1000000
begin
    select @i = RAND()*100000,
           @c = convert(varchar, @i)

```

The output window shows the results of the execution:

```

(1 row affected)

```

The status bar at the bottom indicates "Query canceled".

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

The screenshot shows the Microsoft SQL Server Management Studio interface. The query window displays the following T-SQL code:

```

commit transaction T1

set nocount off

sp_helpindex Users

dbcc checktable (Users)

dbcc showcontig(Users)

DBCC INDEXDEFRAG(InstagramDB,users)

```

The output window shows the results of the execution:

```

DBCC results for 'USERS'.
There are 414537 rows in 3499 pages for object "USERS".
DBCC execution completed. If DBCC printed error messages, contact your system administrator.

Completion time: 2020-09-04T17:44:29.7590580+05:30

```

The status bar at the bottom indicates "Query executed successfully".

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

```

commit transaction T1

set nocount off

sp_helpindex Users

dbcc checktable (Users)

dbcc showcontig(Users)

DBCC INDEXDEFRAG(InstagramDB,users)

```

Completion time: 2020-09-04T17:45:36.7875859+05:30

Query executed successfully.

Messages

```

DBCC SHOWCONTIG scanning 'USERS' table...
Table: 'USERS' (581577110); index ID: 1, database ID: 6
TABLE level scan performed.
- Pages Scanned.....: 3499
- Extents Scanned.....: 439
- Extent Switches.....: 1027
- Avg. Pages per Extent.....: 8.0
- Scan Density [Best Count:Actual Count].....: 42.61% [438:1028]
- Logical Scan Fragmentation .....: 19.38%
- Extent Scan Fragmentation .....: 41.00%
- Avg. Bytes Free per Page.....: 341.9
- Avg. Page Density (full).....: 95.78%
DBCC execution completed. If DBCC printed error messages, contact your system administrator.

```

Completion time: 2020-09-04T17:45:36.7875859+05:30

Query executed successfully.

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

```

set nocount off

sp_helpindex Users

dbcc checktable (Users)

dbcc showcontig(Users)

DBCC INDEXDEFRAG(InstagramDB,users)

update Users set bio = 'Hey there!'
where userId between 200000 and 300000

```

Index Name	Pages Scanned	Pages Moved	Pages Removed
PK__USERB__CB9A1CF004D37AB	3496	3367	128
Users_username_idx	2164	1163	999

Query executed successfully.

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

```
SQLQuery1.sql - DESKTOP-8AIVGG4.InstagramDB (DESKTOP-8AIVGG4(KR (64)) - Microsoft SQL Server Management Studio
File Edit View Query Project Tools Window Help
New Query Execute
Object Explorer
Connect Databases Security Server Objects Replication PolyBase Always On High Availability Management Integration Services Catalogs SQL Server Agent (Agent XPs disabled) XEvent Profiler
SQLQuery1.sql - D_P-8AIVGG4(KR (64)) tablesAndTriggers...P-8AIVGG4(KR (60))
set nocount off

sp_helpindex Users

dbcc checktable (Users)

dbcc showcontig(Users)

DBCC INDEXDEFrag(InstagramDB,users)

update Users set bio = 'Hey there!'
where userId between 200000 and 300000

150 %
Messages
DBCC SHOWCONTIG scanning 'USERS' table...
Table: 'USERS' (581577110); index ID: 1, database ID: 6
TABLE level scan performed.
- Pages Scanned.....: 3371
- Extents Scanned.....: 423
- Extent Switches.....: 422
- Avg. Pages per Extent.....: 8.0
- Scan Density [Best Count:Actual Count].....: 99.76% [422:423]
- Logical Scan Fragmentation .....: 0.39%
- Extent Scan Fragmentation .....: 42.75%
- Avg. Bytes Free per Page.....: 47.4
- Avg. Page Density (full).....: 99.41%
DBCC execution completed. If DBCC printed error messages, contact your system administrator.

Completion time: 2020-09-04T17:46:51.2106159+05:30
150 %
Query executed successfully.
```

Ready S47 PM ENG 04-Sep-20

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

```
SQLQuery1.sql - DESKTOP-8AIVGG4.InstagramDB (DESKTOP-8AIVGG4(KR (64)) - Microsoft SQL Server Management Studio
File Edit View Query Project Tools Window Help
New Query Execute
Object Explorer
Connect Databases Security Server Objects Replication PolyBase Always On High Availability Management Integration Services Catalogs SQL Server Agent (Agent XPs disabled) XEvent Profiler
SQLQuery1.sql - D_P-8AIVGG4(KR (64)) tablesAndTriggers...P-8AIVGG4(KR (60))
dbcc checktable (Users)

dbcc showcontig(Users)

DBCC INDEXDEFrag(InstagramDB,users)

update Users set bio = 'Hey there!'
where userId between 200000 and 300000

delete from Users where userId between 30000 and 40000

150 %
Messages
(40748 rows affected)

Completion time: 2020-09-04T17:47:42.6147020+05:30
150 %
Query executed successfully.
```

Ready S47 PM ENG 04-Sep-20

```

File Edit View Project Tools Window Help
File New Query Object Explorer
SQLQuery1.sql - DESKTOP-8AIVGG4.InstagramDB (DESKTOP-8AIVGG4(KR (64)) - Microsoft SQL Server Management Studio
File Edit View Project Tools Window Help
File New Query Object Explorer
SQLQuery1.sql - D-P-8AIVGG4(KR (64)* tablesAndTriggers_P-8AIVGG4(KR (60))
update Users set bio = 'Hey there!'
where userId between 200000 and 300000

delete from Users where userId between 3000 and 4000
delete from Users where userId between 10000 and 22000
delete from Users where userId between 280000 and 320000
delete from Users where userId between 60000 and 750000

select count(userId) from USERS;
select count(userId) from Users where userId between 60000 and 750000

(9669 rows affected)
(20597 rows affected)
(0 rows affected)
(19492 rows affected)

Completion time: 2020-09-04T18:11:37.5786618+05:30

```

150 %

Messages

Ready

6:07 PM 04-Sep-20

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

```

File Edit View Project Tools Window Help
File New Query Object Explorer
SQLQuery1.sql - DESKTOP-8AIVGG4.InstagramDB (DESKTOP-8AIVGG4(KR (64)) - Microsoft SQL Server Management Studio
File Edit View Project Tools Window Help
File New Query Object Explorer
SQLQuery1.sql - D-P-8AIVGG4(KR (64)* tablesAndTriggers_P-8AIVGG4(KR (60))
delete from Users where userId between 280000 and 320000
delete from Users where userId between 60000 and 750000

select count(userId) from USERS;
select count(userId) from Users where userId between 60000 and 750000

dbcc showcontig(Users)

DBCC INDEXDEFRAG(InstagramDB,Users)

dbcc checktable(Users)

150 %
Messages
DBCC SHOWCONTIG scanning 'USERS' table...
Table: 'USERS' (58157710); index ID: 1, database ID: 6
TABLE level scan performed.
- Pages Scanned.....: 2390
- Extents Scanned...: 316
- Extent Switches...: 1184
- Avg. Pages per Extent....: 7.6
- Scan Density [Best Count:Actual Count].....: 25.23% [299:1185]
- Logical Scan Fragmentation .....: 42.22%
- Extent Scan Fragmentation .....: 23.42%
- Avg. Bytes Free per Page.....: 1119.2
- Avg. Page Density (full).....: 86.17%
DBCC execution completed. If DBCC printed error messages, contact your system administrator.

Completion time: 2020-09-04T18:07:40.1272592+05:30

```

150 %

Messages

Ready

6:09 PM 04-Sep-20

14. Perform the defragmentation process again.

```

SQLQuery1.sql - DESKTOP-BAIVGG4.InstagramDB (DESKTOP-BAIVGG4(KR (64)) - Microsoft SQL Server Management Studio
File Edit View Query Project Tools Window Help
New Query Execute CustId
InstagramDB Object Explorer
Connect Databases Security Server Objects Replication PolyBase Always On High Availability Management Integration Services Catalogs SQL Server Agent (Agent XPs disabled) XEvent Profiler
SQLQuery1.sql - D_P-BAIVGG4KR (64) tablesAndTriggers_P-BAIVGG4KR (60)
delete from Users where userId between 2800000 and 3200000
delete from Users where userId between 600000 and 7500000

select count(userId) from USERS;

dbcc showcontig(Users)

DBCC INDEXDEFrag(InstagramDB_Users)

dbcc checktable(Users)

select Db_name (database_id), * from sys.dm_os_buffer_descriptors

```

Results of Messages

Index Name	Pages Scanned	Pages Moved	Pages Removed
PK_USERS_C9B9A1CF004D37AB	2372	2070	318
Users_username_idx	1360	708	639

Query executed successfully.

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

```

SQLQuery1.sql - DESKTOP-BAIVGG4.InstagramDB (DESKTOP-BAIVGG4(KR (64)) - Microsoft SQL Server Management Studio
File Edit View Query Project Tools Window Help
New Query Execute CustId
InstagramDB Object Explorer
Connect Databases Security Server Objects Replication PolyBase Always On High Availability Management Integration Services Catalogs SQL Server Agent (Agent XPs disabled) XEvent Profiler
SQLQuery1.sql - D_P-BAIVGG4KR (64) tablesAndTriggers_P-BAIVGG4KR (60)
select count(userId) from USERS;

dbcc showcontig(Users)

DBCC INDEXDEFrag(InstagramDB_Users)

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

```

Results of Messages

DBCC results for 'USERS'.
There are 254740 rows in 2072 pages for object "USERS".
DBCC execution completed. If DBCC printed error messages, contact your system administrator.

Completion time: 2020-09-04T18:25:37.3593882+05:30

Query executed successfully.

16. To see the pages used by the resource database

SQLQuery1.sql - DESKTOP-8AIVGG4.InstagramDB (DESKTOP-8AIVGG4\KR (64)) - Microsoft SQL Server Management Studio

File Edit View Query Project Tools Window Help

New Query Execute

Object Explorer

Connect Databases Security Server Objects Replication PolyBase Always On High Availability Management Integration Services Catalogs SQL Server Agent (Agent XPs disabled) XEvent Profiler

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
```

Result Messages

(No column name)	database_id	file_id	page_id	page_level	allocation_unit_id	page_type	row_count	free_space_in_bytes	is_modified	numa_node	read_microsec	is_in_bpool_extension	error_code	op_history	
1	InstagramDB	6	1	0	6488064	FILEHEADER_PAGE	1	6953	0	0	1959	NULL	0	0x15AD8215AB21215A	
2	InstagramDB	6	1	0	6488064	PFS_PAGE	1	2	0	0	1959	NULL	0	0x82121215AB21215A	
3	InstagramDB	6	1	2	0	6488064	GAM_PAGE	2	6	0	1959	NULL	0	0x15AD8215AB21215A	
4	InstagramDB	6	1	3	0	6488064	SGAM_PAGE	2	6	0	1959	NULL	0	0x15AD8215AB21215A	
5	InstagramDB	6	1	6	0	6488064	DIFF_MAP_PAGE	2	6	0	1959	NULL	0	0x15AD8215AB21215A	
6	InstagramDB	6	1	7	0	6488064	MLM_PAGE	2	6	0	1959	NULL	0	0x15AD8215AB21215A	
7	InstagramDB	6	1	8	0	6488064	BOOT_PAGE	1	6326	0	0	9305	NULL	0	0x1215AB215AB21215A
8	InstagramDB	6	1	17	0	327600	DATA_PAGE	66	3872	0	0	441	NULL	0	0x82115AB21215AB21215A
9	InstagramDB	6	1	20	0	458752	DATA_PAGE	88	1496	0	0	441	NULL	0	0x82115AB21215AB21215A
10	InstagramDB	6	1	43	0	281474880642816	INDEX_PAGE	20	7656	0	0	510	NULL	0	0x01215AD821215AB21215A
11	InstagramDB	6	1	64	0	196608	DATA_PAGE	44	5456	0	0	375	NULL	0	0x84D21215AB21215AB21215A
12	InstagramDB	6	1	88	0	196608	DATA_PAGE	96	2336	0	0	338	NULL	0	0x84B21215AB21215AB21215A
13	InstagramDB	6	1	87	0	281474880904960	DATA_PAGE	22	6644	0	0	1959	NULL	0	0xCC8ABAAACAB215AB21215A
14	InstagramDB	6	1	96	0	84424931901440	IAM_PAGE	2	6	0	0	513	NULL	0	0x215AB215ACB215AB21215A
15	InstagramDB	6	1	97	0	2814748811560320	DATA_PAGE	188	2456	0	0	625	NULL	0	0x215AB2121215AB21215A
16	InstagramDB	6	1	98	0	2814748811560320	IAM_PAGE	2	6	0	0	513	NULL	0	0xA8215AB2115AB21215A
17	InstagramDB	6	1	99	0	562949952703978	INDEX_PAGE	188	4336	0	0	513	NULL	0	0xA8215AB2115AB21215A
18	InstagramDB	6	1	100	0	562949952703978	IAM_PAGE	2	6	0	0	513	NULL	0	0xA8215AB2115AB21215A
19	InstagramDB	6	1	101	0	2814748811560320	DATA_PAGE	190	2206	0	0	513	NULL	0	0xA8215AB2115AB21215A
20	InstagramDB	6	1	102	0	2814748811560320	IAM_PAGE	2	6	0	0	513	NULL	0	0xA8215AB2115AB21215A
21	InstagramDB	6	1	103	0	2814748811560320	DATA_PAGE	108	4856	0	0	625	NULL	0	0x01215AD821215AB21215A
22	InstagramDB	6	1	104	0	281474881625858	DATA_PAGE	2	6	0	0	551	NULL	0	0xA8215AB2115AB21215A
23	InstagramDB	6	1	105	0	4609449852336512	INDEX_PAGE	108	4036	n	n	A18	NULL	n	0x80212115AB21215A

Query executed successfully.

LN 198 Col 29 Ch 29 INS

DESKTOP-8AIVGG4 (15.0 RTM) | DESKTOP-8AIVGG4\KR (64) | InstagramDB 00:00:00 4,345 rows

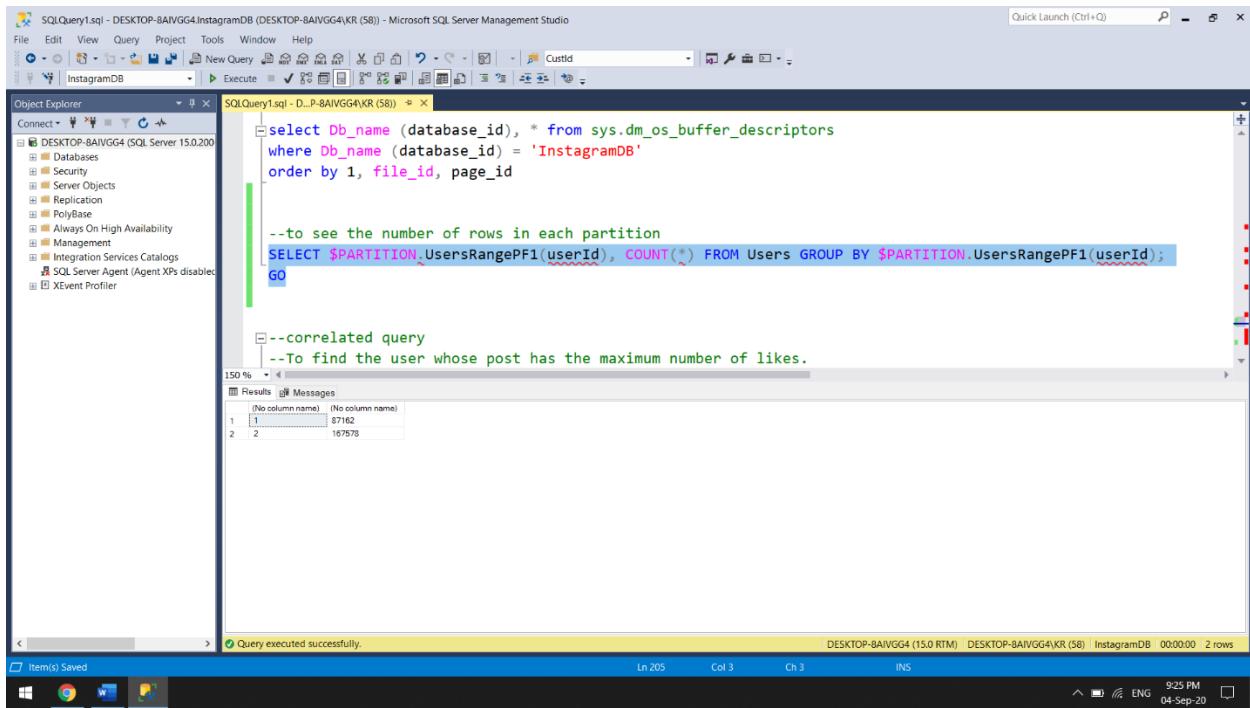
Ready

626 PM ENG 04-Sep-20

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.



The screenshot shows the Microsoft SQL Server Management Studio interface. In the center, there is a 'Results' grid displaying the output of a query. The query results are as follows:

(No column name)	(No column name)
1	87162
2	187578

The status bar at the bottom indicates 'Query executed successfully.' and shows the system information: DESKTOP-8AIVGG4 (15.0 RTM) | DESKTOP-8AIVGG4KR (58) | InstagramDB | 00:00:00 | 2 rows.

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

SQLQuery1.sql - DESKTOP-BAIVGG4.InstagramDB (DESKTOP-BAIVGG4\KR (58)) - Microsoft SQL Server Management Studio

```

-- select Db_name (database_id, * from sys.dm_os_buffer_descriptors
-- where Db_name (database_id) = 'InstagramDB'
-- order by 1, file_id, page_id

-- to see the number of rows in each partition
SELECT $PARTITION.UsersRangePF1(userID), COUNT(*) FROM Users GROUP BY $PARTITION.UsersRangePF1(userID);
GO

CREATE TABLE Posts (
    postId INT PRIMARY KEY,
    postUserId INT FOREIGN KEY(postUserId) REFERENCES USERS(userID)
        ON DELETE CASCADE ON UPDATE CASCADE
    ) on FG1;

-- create PARTITION FUNCTION PostsRangePF1 (int)
--     AS RANGE LEFT FOR VALUES (150000);
GO

-- create PARTITION SCHEME PostsRangePS1
--     AS PARTITION PostsRangePF1
--     TO (FG1, FG2);
GO

CREATE TABLE PostsPartitionTable (
    postId INT NOT NULL,
    postUserId INT NOT NULL,
    caption VARCHAR(100),
    likes INT DEFAULT 0,
    comments INT DEFAULT 0,
    PRIMARY KEY(postId),
    FOREIGN KEY(postUserId) REFERENCES USERS(userID)
        ON DELETE CASCADE ON UPDATE CASCADE
    ) on FG1;

```

Query executed successfully.

LN 249 Col 26 Ch 4 INS

DESKTOP-BAIVGG4 (15.0 RTM) DESKTOP-BAIVGG4\KR (58) InstagramDB 00:00:00 0 rows

Ready 11:34 PM ENG 04-Sep-20

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

SQLQuery1.sql - DESKTOP-BAIVGG4.InstagramDB (DESKTOP-BAIVGG4\KR (58)) - Microsoft SQL Server Management Studio

```

-- create PARTITION FUNCTION PostsRangePF1 (int)
--     AS RANGE LEFT FOR VALUES (150000);
GO

-- create PARTITION SCHEME PostsRangePS1
--     AS PARTITION PostsRangePF1
--     TO (FG1, FG2);
GO

CREATE TABLE PostsPartitionTable (
    postId INT NOT NULL,
    postUserId INT NOT NULL,
    caption VARCHAR(100),
    likes INT DEFAULT 0,
    comments INT DEFAULT 0,
    PRIMARY KEY(postId),
    FOREIGN KEY(postUserId) REFERENCES USERS(userID)
        ON DELETE CASCADE ON UPDATE CASCADE
    ) on FG1;

-- Commands completed successfully.

Completion time: 2020-09-04T23:55:24.203478+05:30

```

Query executed successfully.

LN 230 Col 3 Ch 3 INS

DESKTOP-BAIVGG4 (15.0 RTM) DESKTOP-BAIVGG4\KR (58) InstagramDB 00:00:00 0 rows

Ready 11:55 PM ENG 04-Sep-20

The screenshot shows the Microsoft SQL Server Management Studio interface. The title bar reads "SQLQuery1.sql - DESKTOP-BAIVGG4.InstagramDB (DESKTOP-BAIVGG4\KR (58)) - Microsoft SQL Server Management Studio". The Object Explorer sidebar shows the database structure for "DESKTOP-BAIVGG4 (SQL Server 15.0.2000.5)". The main query window displays T-SQL code for creating a partitioned table:

```
CREATE TABLE PostsPartitionTable (
    postId INT NOT NULL,
    postUserId INT NOT NULL,
    caption VARCHAR(100),
    latitude INT CHECK(latitude > -90 AND latitude < 90),
    longitude INT CHECK(longitude > -180 AND longitude < 180),
    timePost TIMESTAMP NOT NULL,
    disableComments BIT NOT NULL DEFAULT 0,
    likes INT DEFAULT 0,
    comments INT DEFAULT 0,
    PRIMARY KEY(postId),
    FOREIGN KEY(postUserId) REFERENCES USERS(userId)
    ON DELETE CASCADE ON UPDATE CASCADE
) ON PostsRangePS1 (postId)
GO
```

The code is highlighted in blue, indicating it is syntax-highlighted. Below the code, the message "Commands completed successfully." is displayed in the status bar.

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())
```

SQLQuery1.sql - DESKTOP-BAIVGG4.InstagramDB (DESKTOP-BAIVGG4\KR (78)) - Microsoft SQL Server Management Studio

File Edit View Query Project Tools Window Help

Object Explorer

Connect ▾ Databases Security Server Objects Replication PolyBase Always On High Availability Management Integration Services Catalogs SQL Server Agent (Agent XPs) XEvent Profiler

tablesAndTriggers..._P-BAIVGG4\KR (82) SQLQuery1.sql - D...\P-BAIVGG4\KR (78)*

```
declare @i int,
       @pui int

set @i = 1
while @i <= 1000000
begin

    select @i = RAND()*100000
    set @pui = (select top 1 userId from Users order by newid())
    insert into posts (postId, postUserId, caption, latitude, longitude)
    select @i, @pui, 'Caption', 45, 45

    set @i = @i + 1
end
```

150 %

Messages

(1 row affected)

150 %

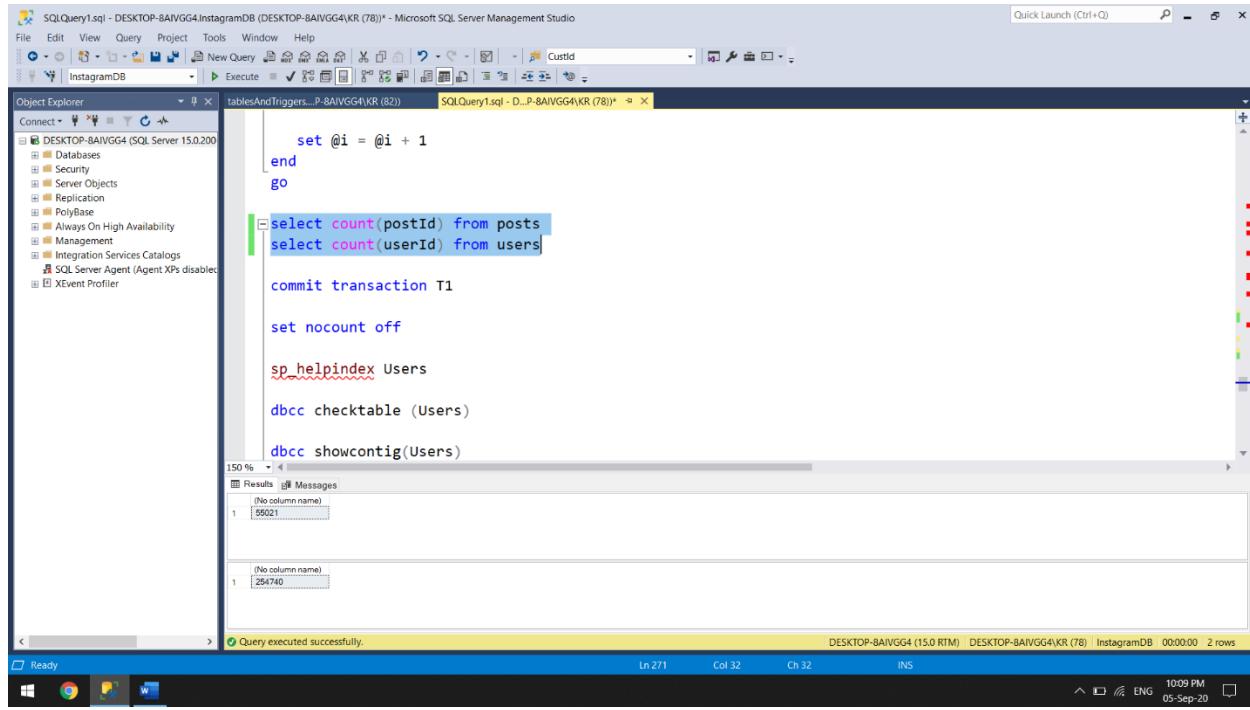
Query canceled.

Ln 269 Col 1 Ch 1 INS

DESKTOP-BAIVGG4 (15.0 RTM) DESKTOP-BAIVGG4\KR (78) InstagramDB 00:07:33 0 rows

848 PM 05-Sep-20 ENG

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)



SQLQuery1.sql - DESKTOP-BAIVGG4.InstagramDB (DESKTOP-BAIVGG4(KR (78)) - Microsoft SQL Server Management Studio

```
set @i = @i + 1
end
go

select count(postId) from posts
select count(userId) from users

commit transaction T1

set nocount off

sp_helpindex Users

dbcc checktable (Users)

dbcc showcontig(Users)
```

Results

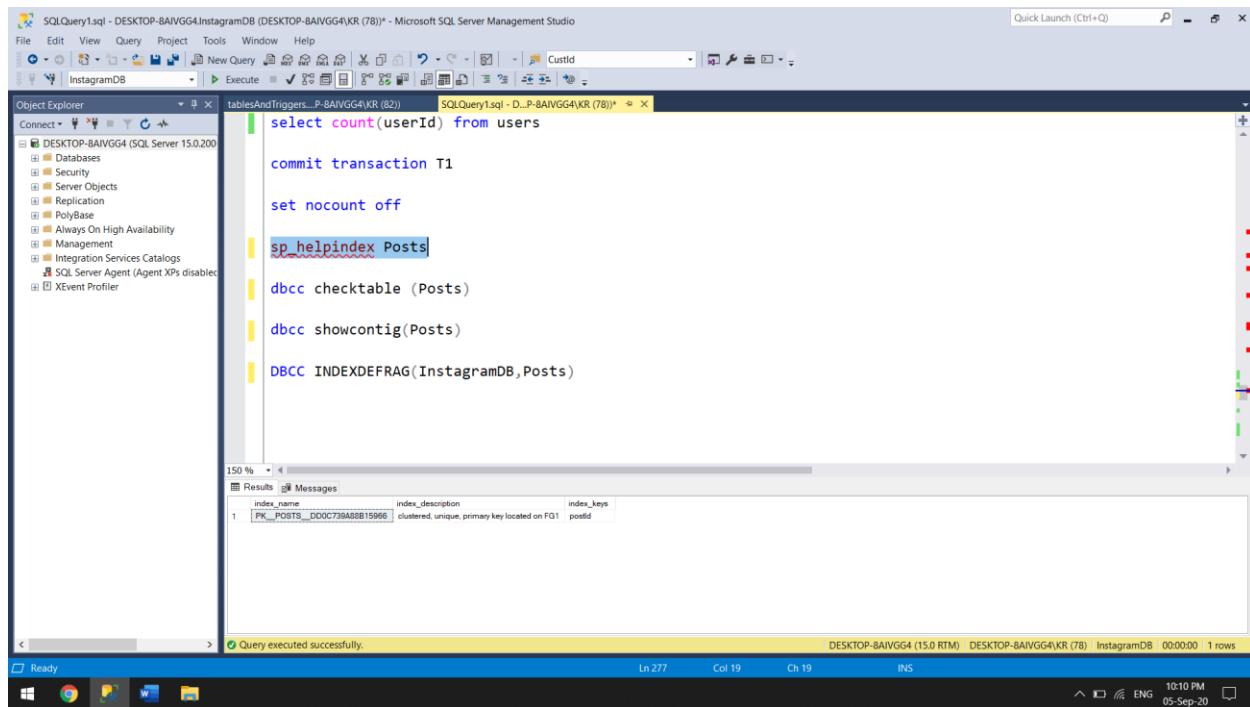
(No column name)	1
	55021

Messages

(No column name)	1
	254740

Query executed successfully.

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



SQLQuery1.sql - DESKTOP-BAIVGG4.InstagramDB (DESKTOP-BAIVGG4(KR (78)) - Microsoft SQL Server Management Studio

```
select count(userId) from users

commit transaction T1

set nocount off

sp_helpindex Posts

dbcc checktable (Posts)

dbcc showcontig(Posts)

DBCC INDEXDEFRAG(InstagramDB,Posts)
```

Results

index_name	index_description	index_keys
PK_POSTS__DD0C739A88B15966	clustered, unique, primary key located on FG1	postId

Query executed successfully.

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

```

SQLQuery1.sql - DESKTOP-BAIVGG4.InstagramDB (DESKTOP-BAIVGG4\KR (78)) - Microsoft SQL Server Management Studio
File Edit View Query Project Tools Window Help
Object Explorer Connect TablesAndTriggers_P-8AIVGG4(KR(82)) SQLQuery1sql - D-P-8AIVGG4(KR(78))*
select count(userId) from users

commit transaction T1

set nocount off

sp_helpindex Posts

dbcc checktable (Posts)

dbcc showcontig(Posts)

DBCC INDEXDEFRAG(InstagramDB,Posts)

150 % ▾
Messages
DBCC results for 'POSTS'.
There are 55021 rows in 503 pages for object "POSTS".
DBCC execution completed. If DBCC printed error messages, contact your system administrator.

Completion time: 2020-09-05T22:10:44.9780311+05:30

150 % ▾
Query executed successfully.
Ln 279 Col 24 Ch 24 INS
DESKTOP-BAIVGG4 (15.0 RTM) DESKTOP-BAIVGG4\KR (78) InstagramDB 00:00:00 0 rows
10:10 PM ENG 05-Sep-20

```

```

SQLQuery1.sql - DESKTOP-BAIVGG4.InstagramDB (DESKTOP-BAIVGG4\KR (78)) - Microsoft SQL Server Management Studio
File Edit View Query Project Tools Window Help
Object Explorer Connect TablesAndTriggers_P-8AIVGG4(KR(82)) SQLQuery1sql - D-P-8AIVGG4(KR(78))*
select count(userId) from users

commit transaction T1

set nocount off

sp_helpindex Posts

dbcc checktable (Posts)

dbcc showcontig(Posts)

150 % ▾
Messages
DBCC SHOWCONTIG scanning 'POSTS' table...
Table: 'POSTS' (997578592); index ID: 1, database ID: 6
TABLE level scan performed.
- Pages Scanned.....: 503
- Extents Scanned.....: 63
- Extent Switches.....: 502
- Avg. Pages per Extent.....: 8.0
- Scan Density [Best Count:Actual Count].....: 12.52% [63:503]
- Logical Scan Fragmentation .....: 98.61%
- Extent Scan Fragmentation .....: 15.87%
- Avg. Bytes Free per Page.....: 2189.2
- Avg. Page Density (full).....: 72.95%
DBCC execution completed. If DBCC printed error messages, contact your system administrator.

Completion time: 2020-09-05T22:10:52.0345779+05:30

150 % ▾
Query executed successfully.
Ln 281 Col 23 Ch 23 INS
DESKTOP-BAIVGG4 (15.0 RTM) DESKTOP-BAIVGG4\KR (78) InstagramDB 00:00:00 0 rows
10:11 PM ENG 05-Sep-20

```

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

The screenshot shows the Microsoft SQL Server Management Studio interface. The Object Explorer on the left shows the database structure. In the center, a query window titled 'SQLQuery1.sql' is open, displaying the following T-SQL code:

```

sp_helpindex Posts
dbcc checktable (Posts)
dbcc showcontig(Posts)
DBCC INDEXDEFRAG(InstagramDB.Posts)

```

The results pane shows a table with one row:

Index Name	Pages Scanned	Pages Moved	Pages Removed
PK_POSTS__DD0C739A88B15966	496	371	132

A message at the bottom of the results pane says 'Query executed successfully.' The status bar at the bottom right indicates the session ID is 78.

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

The screenshot shows the Microsoft SQL Server Management Studio interface. The Object Explorer on the left shows the database structure. In the center, a query window titled 'SQLQuery1.sql' is open, displaying the following T-SQL code:

```

set nocount off
sp_helpindex Posts
dbcc checktable (Posts)
dbcc showcontig(Posts)
DBCC INDEXDEFRAG(InstagramDB.Posts)

```

The results pane shows a table with one row:

Index Name	Pages Scanned	Pages Moved	Pages Removed
PK_POSTS__DD0C739A88B15966	496	371	132

The 'Messages' pane displays the output of the DBCC SHOWCONTIG command:

```

DBCC SHOWCONTIG scanning 'POSTS' table...
Table: 'POSTS' (997578592); index ID: 1, database ID: 6
TABLE level scan performed.
- Pages Scanned.....: 471
- Extents Scanned.....: 47
- Extent Switches.....: 46
- Avg. Pages per Extent.....: 7.9
- Scan Density [Best Count:Actual Count].....: 100.00% [47:47]
- Logical Scan Fragmentation .....: 0.27%
- Extent Scan Fragmentation .....: 23.40%
- Avg. Bytes Free per Page.....: 67.6
- Avg. Page Density (full).....: 98.92%
DBCC execution completed. If DBCC printed error messages, contact your system administrator.

Completion time: 2020-09-05T22:11:22.1939112+05:30

```

A message at the bottom of the results pane says 'Query executed successfully.' The status bar at the bottom right indicates the session ID is 78.

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.

```

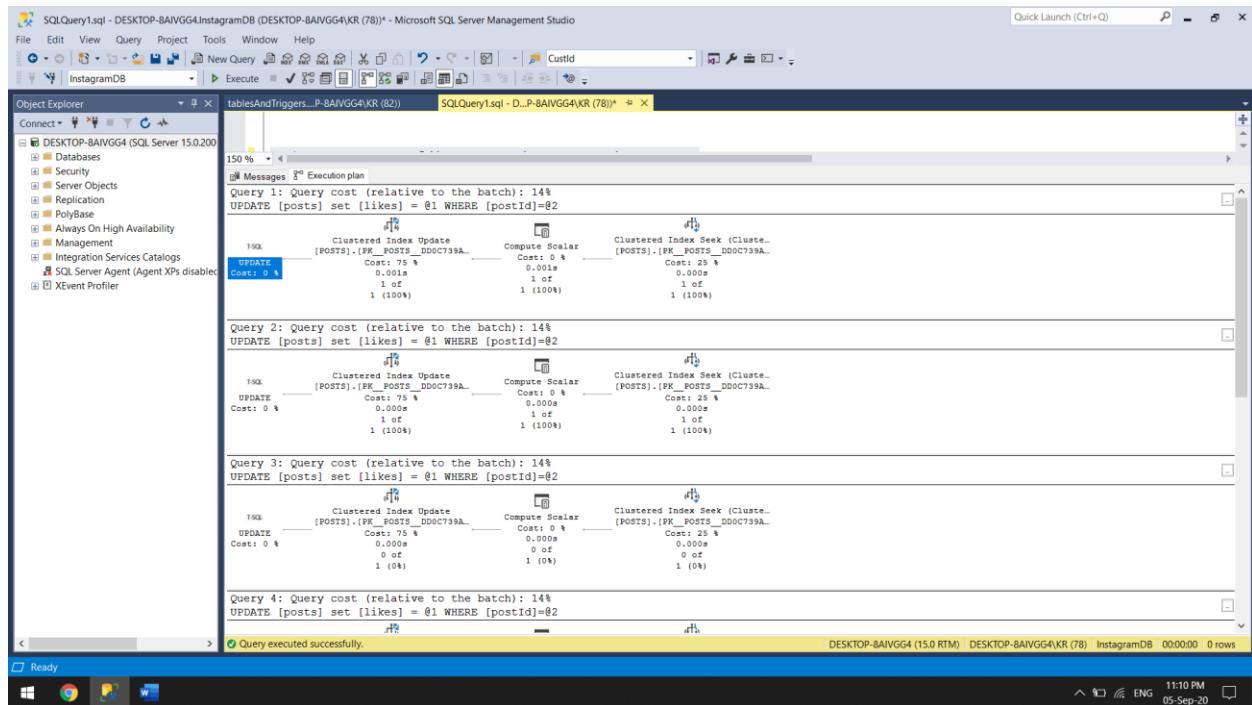
update posts set likes = 10 where postId = 71632
update posts set likes = 100 where postId = 41084
update posts set likes = 50 where postId = 233829
update posts set likes = 250 where postId = 932938
update posts set likes = 50 where postId = 367395
update posts set likes = 50 where postId = 410535
update posts set likes = 10 where postId = 777810

--correlated query
--To find the user whose post has the maximum number of likes.
SELECT firstName, lastName
FROM USERS
WHERE userId IN (SELECT postUserId
                  FROM POSTS
                  WHERE likes = (SELECT MAX(likes))

```

Query executed successfully.

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

File Edit View Query Project Tools Window Help

New Query Execute Quick Launch (Ctrl+Q)

Object Explorer

Connections Databases Security Server Objects Replication PolyBase Always On High Availability Management Integration Services Catalogs SQL Server Agent (Agent XPs disabled) XEvent Profiler

tablesAndTriggers_P-8AIVGG4(KR (82)) SQLQuery1.sql - D...P-8AIVGG4(KR (78))

Execution plan

Messages

Query 1: Query cost (relative to the batch): 14%
UPDATE [posts] set [likes] = @1 WHERE [postid]=@2

Task Clustered Index Update [POSTS].[PK_POSTS_DDOCT39A... Compute Scalar Cost: 75 % 0.001s 1 of 1 (100%) 1 (100%)

Task Clustered Index Seek (Clustered Index Scan) [POSTS].[PK_POSTS_DDOCT39A... Cost: 25 % 0.000s 1 of 1 (100%) 1 (100%)

Query 2: Query cost (relative to the batch): 14%
UPDATE [posts] set [likes] = @1 WHERE [postid]=@2

Task Clustered Index Update [POSTS].[PK_POSTS_DDOCT39A... Compute Scalar Cost: 75 % 0.000s 1 of 1 (100%) 1 (100%)

Task Clustered Index Seek (Clustered Index Scan) [POSTS].[PK_POSTS_DDOCT39A... Cost: 25 % 0.000s 1 of 1 (100%) 1 (100%)

Query 3: Query cost (relative to the batch): 14%
UPDATE [posts] set [likes] = @1 WHERE [postid]=@2

Task Clustered Index Update [POSTS].[PK_POSTS_DDOCT39A... Compute Scalar Cost: 75 % 0.000s 0 of 0 (0%) 0 (0%)

Task Clustered Index Seek (Clustered Index Scan) [POSTS].[PK_POSTS_DDOCT39A... Cost: 25 % 0.000s 0 of 0 (0%) 0 (0%)

Query 4: Query cost (relative to the batch): 14%
UPDATE [posts] set [likes] = @1 WHERE [postid]=@2

Task Clustered Index Update [POSTS].[PK_POSTS_DDOCT39A... Compute Scalar Cost: 75 % 0.000s 0 of 0 (0%) 0 (0%)

Task Clustered Index Seek (Clustered Index Scan) [POSTS].[PK_POSTS_DDOCT39A... Cost: 25 % 0.000s 0 of 0 (0%) 0 (0%)

Query executed successfully.

DESKTOP-8AIVGG4 (15.0 RTM) DESKTOP-8AIVGG4\KR (78) InstagramDB 00:00:00 0 rows

Ready

SQLQuery1.sql - DESKTOP-8AIVGG4\InstagramDB (DESKTOP-8AIVGG4\KR (78)) - Microsoft SQL Server Management Studio

File Edit View Query Project Tools Window Help

New Query Execute Quick Launch (Ctrl+Q)

Object Explorer

Connections Databases Security Server Objects Replication PolyBase Always On High Availability Management Integration Services Catalogs SQL Server Agent (Agent XPs disabled) XEvent Profiler

tablesAndTriggers_P-8AIVGG4(KR (82)) SQLQuery1.sql - D...P-8AIVGG4(KR (78))

Execution plan

Messages

Query 5: Query cost (relative to the batch): 14%
UPDATE [posts] set [likes] = @1 WHERE [postid]=@2

Task Clustered Index Update [POSTS].[PK_POSTS_DDOCT39A... Compute Scalar Cost: 75 % 0.000s 0 of 1 (0%) 0 (0%)

Task Clustered Index Seek (Clustered Index Scan) [POSTS].[PK_POSTS_DDOCT39A... Cost: 25 % 0.000s 0 of 1 (0%) 1 (100%)

Query 6: Query cost (relative to the batch): 14%
UPDATE [posts] set [likes] = @1 WHERE [postid]=@2

Task Clustered Index Update [POSTS].[PK_POSTS_DDOCT39A... Compute Scalar Cost: 75 % 0.000s 0 of 0 (0%) 0 (0%)

Task Clustered Index Seek (Clustered Index Scan) [POSTS].[PK_POSTS_DDOCT39A... Cost: 25 % 0.000s 0 of 0 (0%) 0 (0%)

Query 7: Query cost (relative to the batch): 14%
UPDATE [posts] set [likes] = @1 WHERE [postid]=@2

Task Clustered Index Update [POSTS].[PK_POSTS_DDOCT39A... Compute Scalar Cost: 75 % 0.000s 0 of 0 (0%) 0 (0%)

Task Clustered Index Seek (Clustered Index Scan) [POSTS].[PK_POSTS_DDOCT39A... Cost: 25 % 0.000s 0 of 0 (0%) 0 (0%)

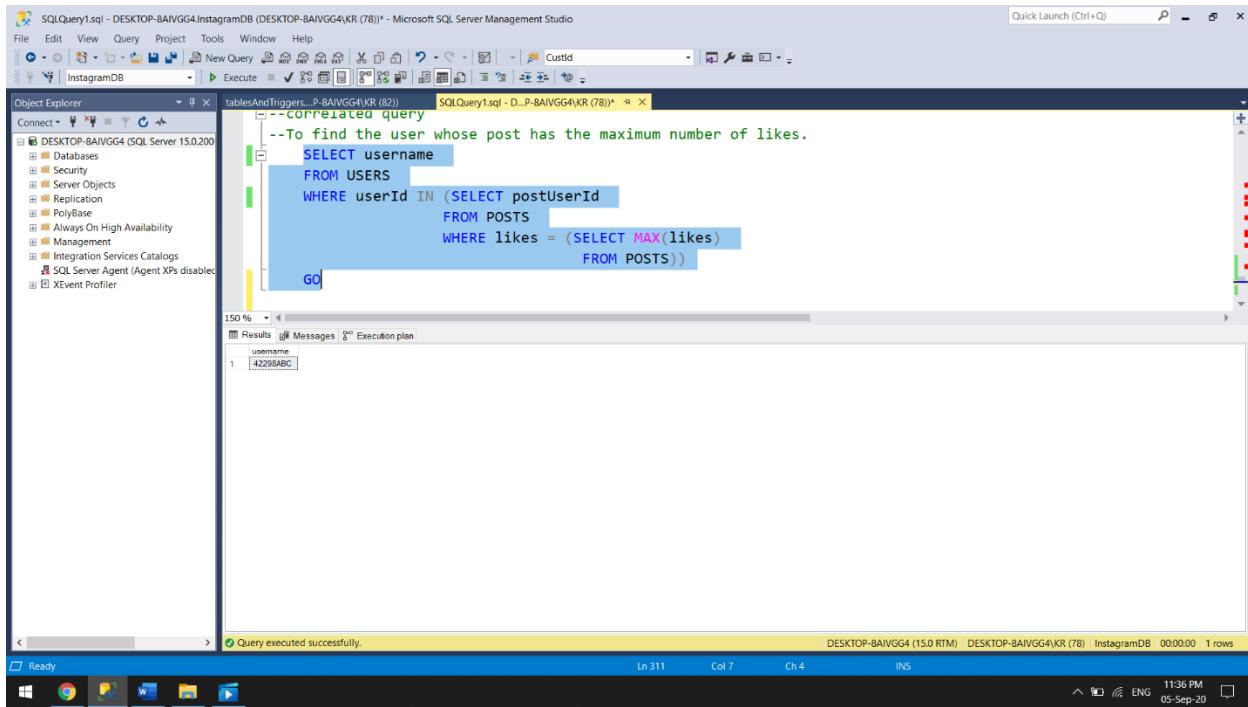
Query executed successfully.

DESKTOP-8AIVGG4 (15.0 RTM) DESKTOP-8AIVGG4\KR (78) InstagramDB 00:00:00 0 rows

Ready

12. CORRELATED QUERY

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



The screenshot shows the Microsoft SQL Server Management Studio interface. The Object Explorer on the left shows the database structure for 'DESKTOP-8AIVGG4 (SQL Server 15.0.200)'. The 'TablesAndTriggers' node under 'DESKTOP-8AIVGG4' is expanded, showing 'USERS' and 'POSTS'. The 'Query' tab in the center contains the following T-SQL code:

```
--To find the user whose post has the maximum number of likes.
SELECT username
FROM USERS
WHERE userId IN (SELECT postUserId
                  FROM POSTS
                  WHERE likes = (SELECT MAX(likes)
                                  FROM POSTS))
GO
```

The results pane below shows the output:

username
42298ABC

A green message at the bottom of the results pane says 'Query executed successfully.'

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%.

SQLQuery1.sql - DESKTOP-8AIVGG4.InstagramDB (DESKTOP-8AIVGG4\KR (78)) - Microsoft SQL Server Management Studio

```

SELECT username
FROM USERS
WHERE userId IN (SELECT postUserId
                  FROM POSTS
                  WHERE likes = (SELECT MAX(likes)
                                  FROM POSTS))
GO

```

create index Posts_postUserId_idx on Posts(postUserId) on FG2

Execution plan details:

- Missing Index (Impact 13.9578):** CREATE NONCLUSTERED INDEX <Name of Missing Index, sysname,> ON [dbo].[POSTS] ([likes]) INCLUDE ([postUserId])
- Hash Match (Right Semi Join):** Cost: 50 %
- Nested Loops (Inner Join):** Cost: 1 %
- Stream Aggregate (Aggregate):** Cost: 1 %
- Clustered Index Scan (Clustered [POSTS].[PK_POSTS_D00C739A...]):** Cost: 0.000 %
- Index Scan (NonClustered) [USERS].[Users_username_idx]:** Cost: 0.000 %

Query executed successfully.

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

SQLQuery1.sql - DESKTOP-8AIVGG4.InstagramDB (DESKTOP-8AIVGG4\KR (67)) - Microsoft SQL Server Management Studio

```

SELECT username
FROM USERS
WHERE userId IN (SELECT postUserId
                  FROM POSTS
                  WHERE likes = (SELECT MAX(likes)
                                  FROM POSTS))
GO

```

sp_helpindex Posts
create index Posts_likes_idx on Posts(likes) on FG2

--aggregate query
--To find the total number of posts on a user's profile.

Completion time: 2020-09-06T15:57:55.3448817+05:30

Query executed successfully.

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

The screenshot shows the Microsoft SQL Server Management Studio interface. In the Object Explorer, a database named 'InstagramDB' is selected. In the center pane, a query window displays the following T-SQL code:

```
--correlated query
--To find the user whose post has the maximum number of likes.
SELECT username
FROM USERS
WHERE userId IN (SELECT postUserId
                  FROM POSTS
                  WHERE likes = (SELECT MAX(likes)
                                  FROM POSTS))
GO
```

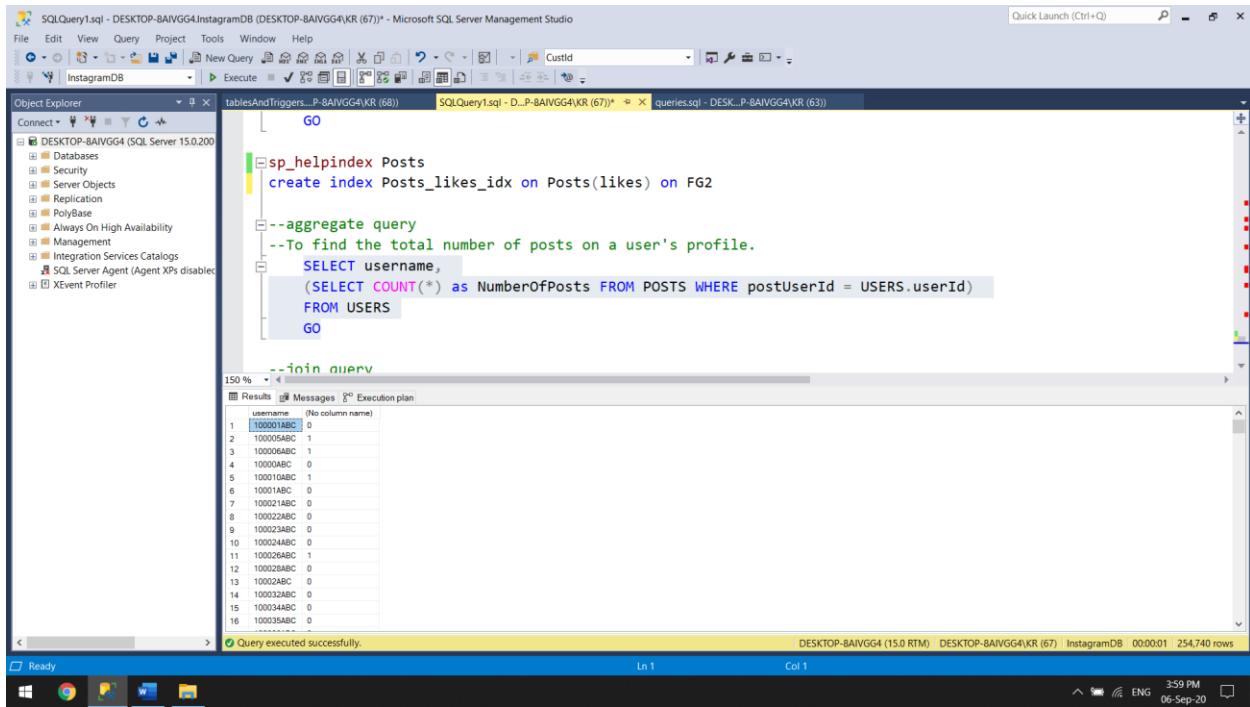
Below the code, the 'Execution plan' section shows a detailed tree diagram of the query's execution steps. The plan starts with a 'Nested Loops (Inner Join)' operator, which performs a 'Clustered Index Seek (Clustered)' on the 'POSTS' table to find posts with the maximum likes. This is followed by a 'Distinct Sort' operator. The result of this is then used in another 'Nested Loops (Inner Join)' to find the corresponding user in the 'USERS' table via a 'Key Lookup (Clustered)' operation. Finally, a 'Stream Aggregate (Aggregate)' operator calculates the maximum likes, and a 'Top' operator retrieves the single row with the maximum likes. The plan also includes 'Index Scan (NonClustered)' and 'Index Seek (NonClustered)' operations for the 'Likes' index on the 'POSTS' table.

At the bottom of the screen, a status bar indicates 'Query executed successfully.' and shows system information like the date and time.

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.



The screenshot shows the Microsoft SQL Server Management Studio interface. The Object Explorer on the left shows the database structure for 'DESKTOP-8AIVGG4'. The central pane displays a T-SQL script:

```
sp_helpindex Posts
create index Posts_likes_idx on Posts(likes) on FG2

--aggregate query
--To find the total number of posts on a user's profile.
SELECT username,
       (SELECT COUNT(*) as NumberOfPosts FROM POSTS WHERE postUserId = USERS.userId)
  FROM USERS
GO

--join query
```

The results pane below shows the output of the query:

username	NumberOfPosts
100001ABC	0
100005ABC	1
100006ABC	1
100007ABC	0
100010ABC	1
100011ABC	0
100021ABC	0
100022ABC	0
100023ABC	0
100024ABC	0
100025ABC	0
100026ABC	1
100027ABC	0
100028ABC	0
100029ABC	0
100030ABC	0
100031ABC	0
100032ABC	0
100033ABC	0
100034ABC	0
100035ABC	0

A status bar at the bottom indicates 'Query executed successfully.'

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).

SQLQuery1.sql - DESKTOP-8AIVGG4.InstagramDB (DESKTOP-8AIVGG4(KR (67)) - Microsoft SQL Server Management Studio

```

GO
sp_helpindex Posts
create index Posts_likes_idx on Posts(likes) on FG2

--aggregate query
--To find the total number of posts on a user's profile.
SELECT username,
(SELECT COUNT(*) as NumberOfPosts FROM POSTS WHERE postUserId = USERS.userId)
FROM USERS
GO

--join query

```

Result Set Messages Execution plan

Query 1: Query cost (relative to the batch): 100%

```

SELECT username, (SELECT COUNT(*) as NumberOfPosts FROM POSTS WHERE postUserId = USERS.userId) FROM USERS

```

Execution Plan:

- Hash Match (Right Outer Join)**: Cost: 50 %, 0.13s, 254740 of 254740 (100%)
- Compute Scalar**: Cost: 0 %
- Hash Match (Aggregate)**: Cost: 20 %, 0.022s, 49423 of 55021 (100%)
- Clustered Index Scan (Clustered [POSTS].[PK_POSTS_DDOC739A])**: Cost: 9 %, 0.005s, 55021 of 55021 (100%)
- Index Scan (NonClustered [USERS].[Users_username_idx])**: Cost: 21 %, 0.022s, 254740 of 254740 (100%)

Query executed successfully.

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

SQLQuery1.sql - DESKTOP-8AIVGG4.InstagramDB (DESKTOP-8AIVGG4(KR (67)) - Microsoft SQL Server Management Studio

```

GO
sp_helpindex Posts
create index Posts_likes_idx on Posts(likes) on FG2

--aggregate query
--To find the total number of posts on a user's profile.
SELECT username,
(SELECT COUNT(*) as NumberOfPosts FROM POSTS WHERE postUserId = USERS.userId)
FROM USERS
GO

drop index Users_username_idx on Users

```

Completion time: 2020-09-06T16:04:24.7509861+05:30

Commands completed successfully.

Execution Plan:

- Hash Match (Right Outer Join)**: Cost: 50 %, 0.13s, 254740 of 254740 (100%)
- Compute Scalar**: Cost: 0 %
- Hash Match (Aggregate)**: Cost: 20 %, 0.022s, 49423 of 55021 (100%)
- Clustered Index Scan (Clustered [POSTS].[PK_POSTS_DDOC739A])**: Cost: 9 %, 0.005s, 55021 of 55021 (100%)
- Index Scan (NonClustered [USERS].[Users_username_idx])**: Cost: 37 %, 0.022s, 254740 of 254740 (100%)

Query executed successfully.

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

SQLQuery1.sql - DESKTOP-8AIVGG4.InstagramDB (DESKTOP-8AIVGG4\KR (67)) - Microsoft SQL Server Management Studio

```

GO
sp_helpindex Posts
create index Posts_likes_idx on Posts(likes) on FG2

--aggregate query
--To find the total number of posts on a user's profile.
SELECT username,
       (SELECT COUNT(*) as NumberOfPosts FROM POSTS WHERE postUserId = USERS.userId)
  FROM USERS
GO
drop index Users_username_idx on Users

```

Results Messages Execution plan

Query 1: Query cost (relative to the batch): 100%

SELECT username, (SELECT COUNT(*) as NumberOfPosts FROM POSTS WHERE postUserId = USERS.userId) FROM USERS

Query executed successfully.

SQLQuery1.sql - DESKTOP-8AIVGG4.InstagramDB (DESKTOP-8AIVGG4\KR (67)) - Microsoft SQL Server Management Studio

```

GO
sp_helpindex Posts
create index Posts_likes_idx on Posts(likes) on FG2

--aggregate query
--To find the total number of posts on a user's profile.
SELECT username,
       (SELECT COUNT(*) as NumberOfPosts FROM POSTS WHERE postUserId = USERS.userId)
  FROM USERS
GO
drop index Users_username_idx on Users

```

Results Messages Execution plan

Query 1: Query cost (relative to the batch): 100%

SELECT username, (SELECT COUNT(*) as NumberOfPosts FROM POSTS WHERE postUserId = USERS.userId) FROM USERS

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	254740
Actual Number of Rows for All Executions	254740
Actual Number of Batches	0
Estimated Operator Cost	1.81757 (37%)
Estimated I/O Cost	1.5372
Estimated CPU Cost	0.280371
Estimated Substr Cost	1.81757
Number of Executions	1
Estimated Number of Executions	1
Estimated Number of Rows to be Read	254740
Estimated Number of Rows Per Execution	254740
Estimated Row Size	22 B
Actual Rebinds	0
Actual Rewinds	0
Node ID	5
Object	[InstagramDB].[dbo].[USERS].[PK_USERS_CB9A1CFF004D37AB]
Output List	[InstagramDB].[dbo].[USERS].userId, [InstagramDB].[dbo].[USERS].username

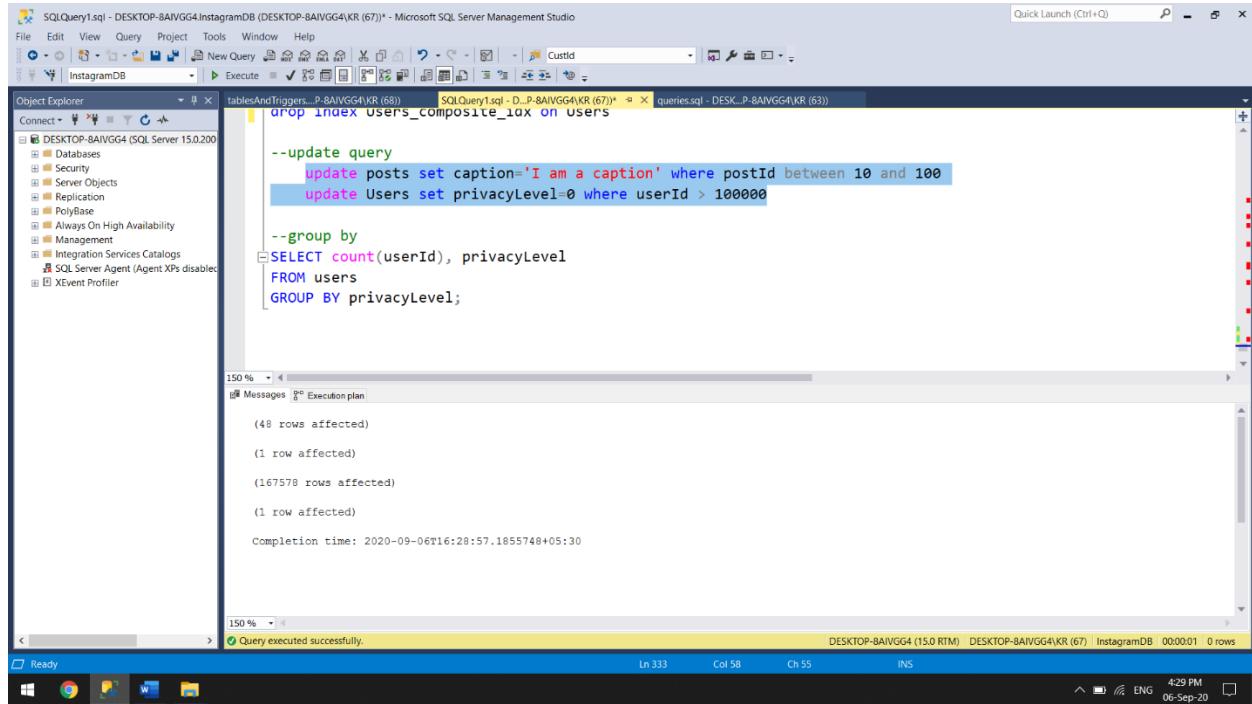
Query executed successfully.

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.



```
--update query
update posts set caption='I am a caption' where postId between 10 and 100
update Users set privacyLevel=0 where userId > 100000

--group by
SELECT count(userId), privacyLevel
FROM users
GROUP BY privacyLevel;
```

150 % → ↻
Messages Execution plan
(48 rows affected)
(1 row affected)
(167578 rows affected)
(1 row affected)
Completion time: 2020-09-06T16:28:57.1855748+05:30

150 % → ↻
Query executed successfully.

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

```

--update query
update posts set caption='I am a caption' where postId between 10 and 100
update Users set privacyLevel=0 where userId > 100000

--update query
UPDATE [posts] set [caption] = @1 WHERE [postId]>=@2 AND [postId]<=@3
UPDATE [Users] set [privacyLevel] = @1 WHERE [userId]>@2

```

Execution Plan for Query 1:

- UPDATE [posts]**: Cost: 0 %
- Clustered Index Update**: [POSTS].[PK_POSTS_DDOCT39A...]
- Compute Scalar**: Cost: 0 %
- Clustered Index Seek**: [POSTS].[PK_POSTS_DDOCT39A...]

Execution Plan for Query 2:

- UPDATE [Users]**: Cost: 0 %
- Clustered Index Update**: [USERS].[PK_USERS_CB9A1CF...]
- Compute Scalar**: Cost: 0 %
- Parallelism (Gather Streams)**: Cost: 1 %
- Index Scan (NonClustered)**: [USERS].[Users_username_idx]

Query executed successfully.

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.

```

--update query
update posts set caption='I am a caption' where postId between 10 and 100
update Users set privacyLevel=0 where userId > 100000

--group by
SELECT count(postId)
FROM posts
GROUP BY caption;

create index Users_privacylevel_idx on Users/privacylevel

```

(No column name)
54973
48

Query executed successfully.

This is its execution plan.

SQLQuery1.sql - DESKTOP-8AIVGG4.InstagramDB (DESKTOP-8AIVGG4\KR (67)) - Microsoft SQL Server Management Studio

```
--update query
update posts set caption='I am a caption' where postId between 10 and 100
update Users set privacyLevel=0 where userId > 100000

--group by
SELECT count(postId)
FROM posts
GROUP BY caption;

create index Users_privacylevel_idx on Users(privacyLevel)
```

Result Messages Execution plan

Query 1: Query cost (relative to the batch): 100%

SELECT count(postId) FROM posts GROUP BY caption

Execution Plan:

- SELECT Cost: 4 %
- Compute Scalar Cost: 0 %
- Hash Match (Aggregate) Cost: 55 %
- Clustered Index Scan [POSTS].[PK_POSTS] DDOC739A...

Cost: 47 %

0.04ms 0.015s 55021 dt 55021 (100%)

Query executed successfully.

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

SQLQuery1.sql - DESKTOP-8AIVGG4.InstagramDB (DESKTOP-8AIVGG4\KR (67)) - Microsoft SQL Server Management Studio

```
update Users set privacyLevel=0 where userId > 100000

--group by
SELECT count(postId)
FROM posts
GROUP BY caption;

create index Posts_caption_idx on Posts(caption)
drop index Posts_caption_idx on Posts
```

Execution plan:

Commands completed successfully.

Completion time: 2020-09-06T17:08:17.4638266+05:30

Query executed successfully.

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

```

--group by
SELECT count(postId)
FROM posts
GROUP BY caption;

create index Posts_caption_idx on Posts(caption)
drop index Posts_caption_idx on Posts

```

Execution plan details:

- SELECT Cost: 0 %**
- Stream Aggregate (Aggregate)**: Cost: 16 %, Cost: 0.012s, 0.007ms, 2 of 2 (100%)
- Index Scan (NonClustered)**: Cost: 54 %, Cost: 0.007ms, 55021 of 55021 (100%)

Message bar: Query executed successfully.

So we drop the index Posts_caption_idx.

```

--group by
SELECT count(postId)
FROM posts
GROUP BY caption;

create index Posts_caption_idx on Posts(caption)
drop index Posts_caption_idx on Posts

```

Message bar: Commands completed successfully.

Completion time: 2020-09-06T17:11:21.7257429+05:30

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.

The screenshot shows the Microsoft SQL Server Management Studio interface. In the Object Explorer, the database 'InstagramDB' is selected. In the center pane, a query window displays the following T-SQL code:

```
create index Users_privacylevel_idx on Users(privacyLevel)
drop index Users_privacylevel_idx on Users
sp_helpindex Users

--for composite index
select userId from USERS where password='password' and privacyLevel=1

create index Users_composite_idx on Users(password, privacyLevel)
drop index Users_composite_idx on Users
```

Below the code, the 'Execution plan' tab is selected. It shows a single step: a 'Clustered Index Scan' on the 'USERS' table, which is clustered by 'password'. The cost of the query is 100%, and it scans 87162 rows. A tooltip for the 'password' column indicates a missing index.

In the status bar at the bottom, it says 'Query executed successfully.' and shows system information like 'DESKTOP-8AIVGG4 (15.0 RTM) DESKTOP-8AIVGG4(KR (67)) InstagramDB 00:00:00 87.162 rows' and the date '06-Sep-20'.

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

The screenshot shows the Microsoft SQL Server Management Studio interface. In the Object Explorer, the database 'InstagramDB' is selected. In the center pane, a query window titled 'SQLQuery1.sql' contains the following T-SQL code:

```
create index Users_privacylevel_idx on Users(privacyLevel)
drop index Users_privacylevel_idx on Users
sp_helpindex Users

--for composite index
select userId from USERS where password='password' and privacyLevel=1

create index Users_composite_idx on Users(password, privacyLevel)
drop index Users_composite_idx on Users
```

The execution results show:

- Commands completed successfully.
- Completion time: 2020-09-06T16:55:33.5910692+05:30

The status bar at the bottom indicates 'Query executed successfully.'

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

The screenshot shows the Microsoft SQL Server Management Studio interface. In the Object Explorer, the database 'InstagramDB' is selected. In the center pane, a query window titled 'SQLQuery1.sql' contains the following T-SQL code:

```
dbcc dropcleanbuffers
go

--for composite index
select userId from USERS where password='password' and privacyLevel=1

create index Users_composite_idx on Users(password, privacyLevel)
drop index Users_composite_idx on Users
```

The execution results show:

- Results (1 row(s) affected)
- Messages (1 message)
- Execution plan

The execution plan details for Query 1 are shown:

- Query cost (relative to the batch): 100%
- SELECT [userId] FROM [USERS] WHERE [password]=@1 AND [privacyLevel]=@2
- Index Seek (NonClustered)
[USERS].[Users_composite_idx]
Cost: 100 %
0.081s
87142 of
86626 (100%)

The status bar at the bottom indicates 'Query executed successfully.'