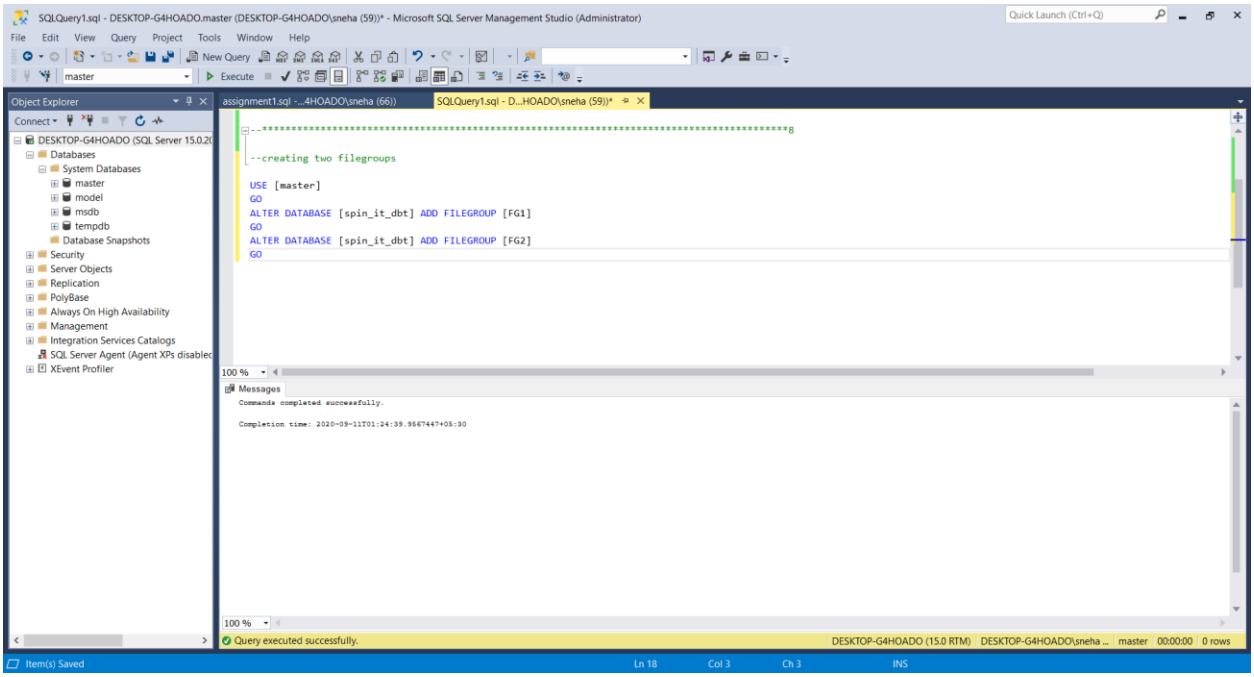
DBT ASSIGNMENT 1

Sneha Hegde

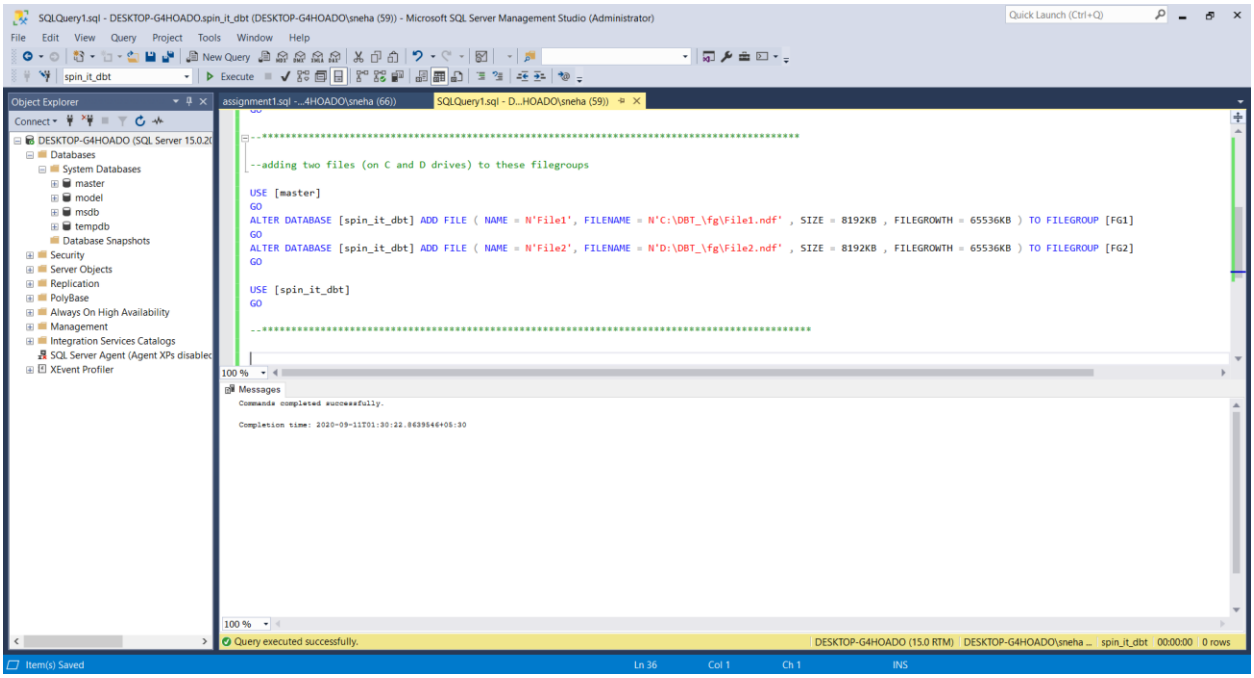
PES1201801157

In the shared folder, I have also uploaded my DBMS project report. As can be seen, my relational model is normalised and has been tested for the Lossless Join Property.

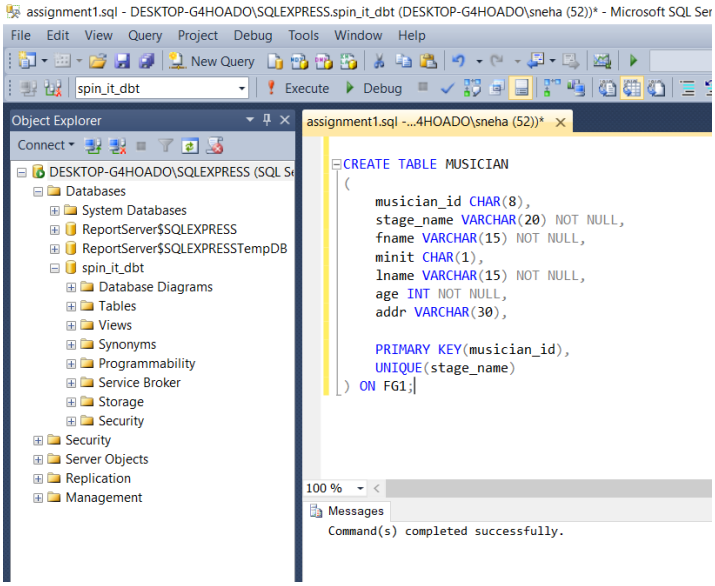
Adding filegroups to the database-



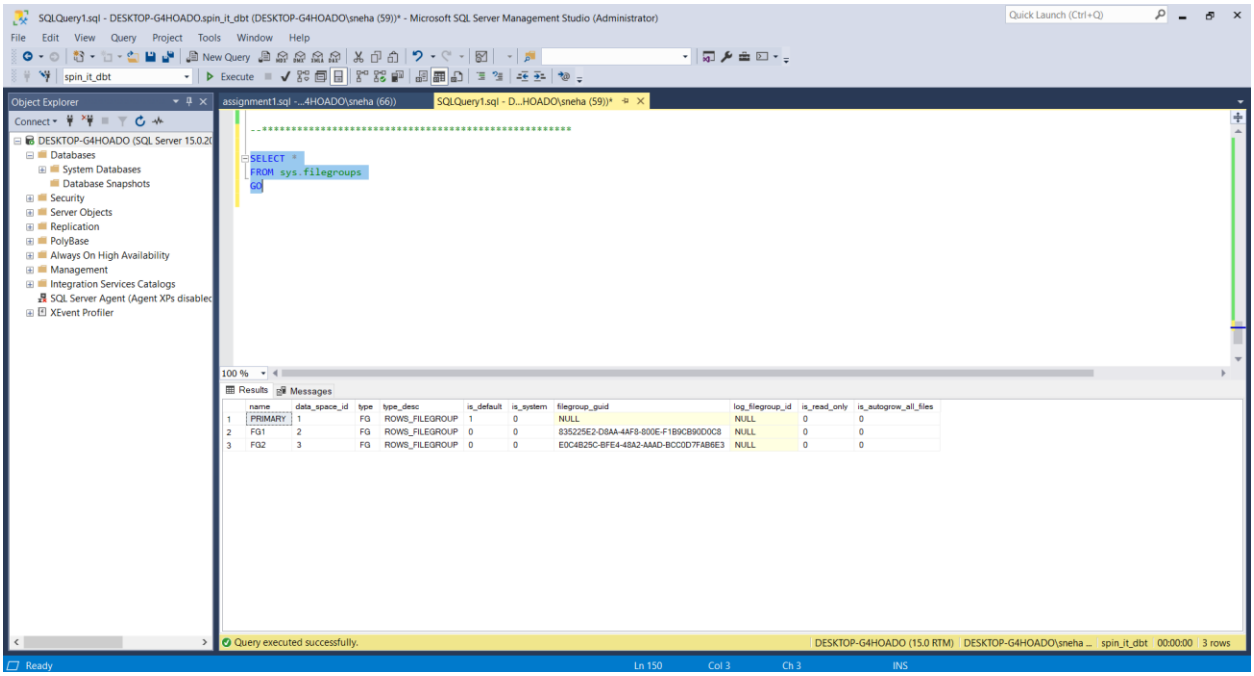
Adding files under the filegroups, in C and D drives

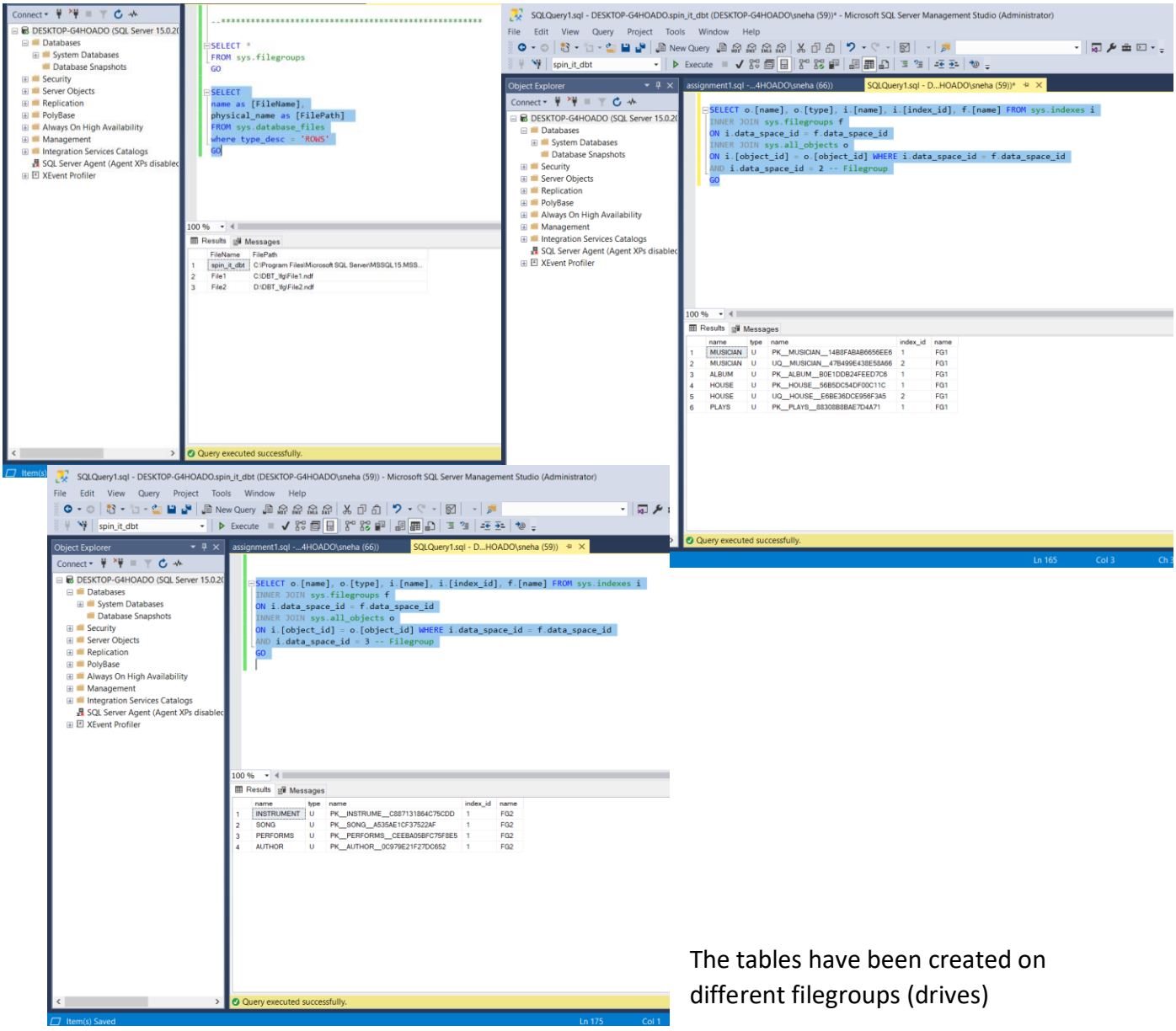


After this, I created and split the tables among the filegroups



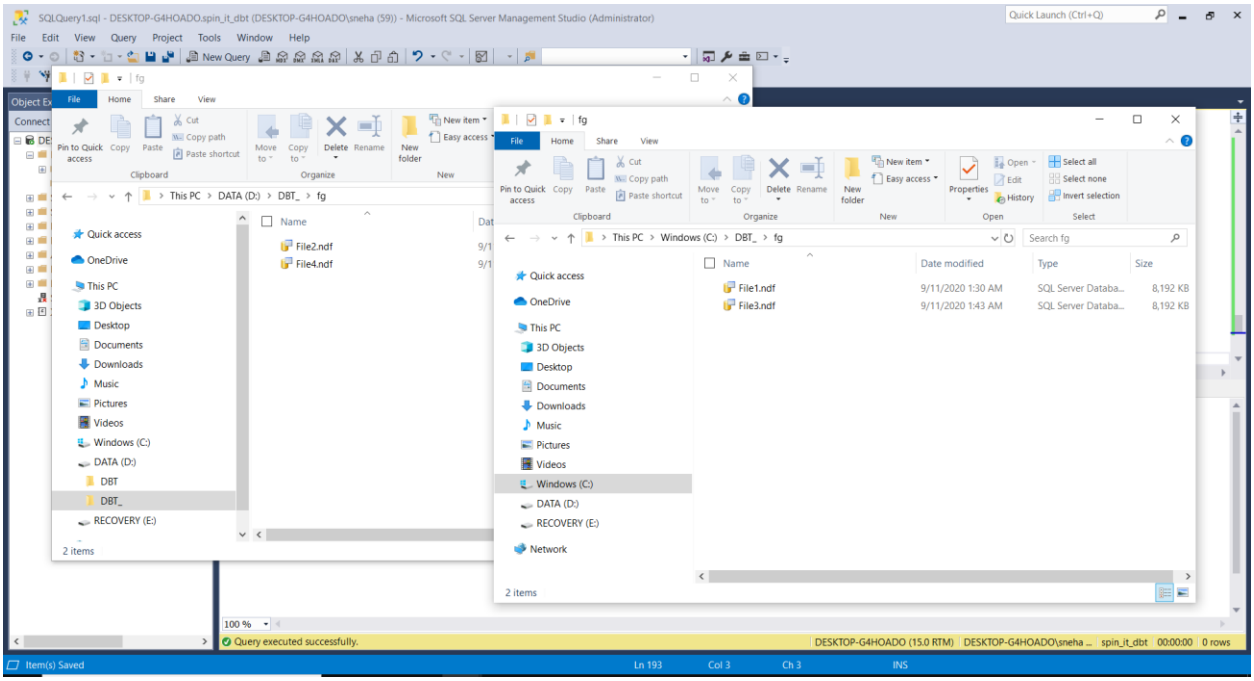
Using the following queries, I verified that the tables were created on the correct filegroups.





The tables have been created on different filegroups (drives)

To partition a table, we first create new filegroups (on C and D) and add files to them, using the same steps as above



Next, we create a partition function, to map the rows of a table into partitions, based on the partitioning column. In our case, that's sl_no

An INT can store values from -2147483648 to 2147483647. Therefore, when we give the boundary value as 1000, with RANGE LEFT, our partitions will be

>-2147483648 and <=1000, and

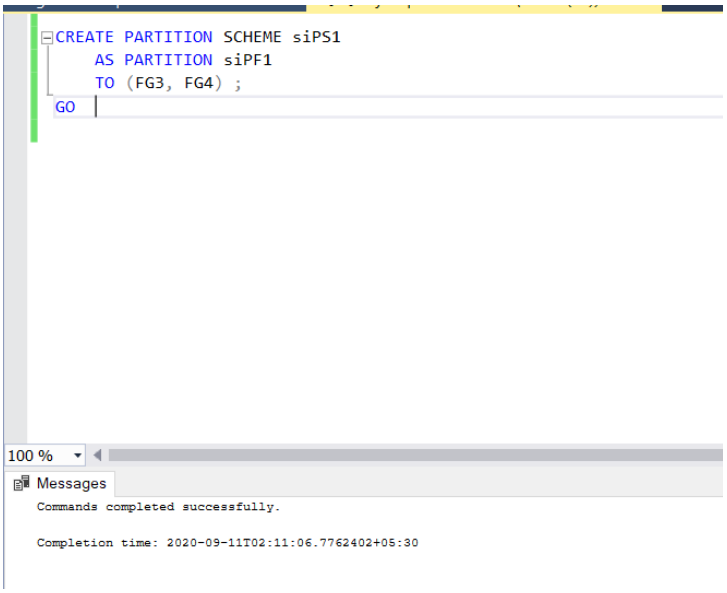
>1000 and <=2147483647,

if explicitly mentioned. Otherwise, the upper and lower bounds will appear as NULL.

```
CREATE PARTITION FUNCTION siPF1 (int)
    AS RANGE LEFT FOR VALUES (1000) ;
GO
```

```
100 %
Messages
Commands completed successfully.

Completion time: 2020-09-11T01:44:40.2530022+05:30
```

Now, we have to create a partition scheme to map the partitions to filegroups



Creating a partition table on the above scheme-

```
CREATE TABLE musician_partition (sl_no INT NOT NULL,
                                 musician_id CHAR(8) NOT NULL,
                                 stage_name VARCHAR(20) NOT NULL,
                                 fname VARCHAR(15) NOT NULL,
                                 minit CHAR(1) NOT NULL,
                                 lname VARCHAR(15) NOT NULL,
                                 age INT NOT NULL,
                                 addr VARCHAR(30) NOT NULL,

                                 PRIMARY KEY(sl_no))
    ON siPS1 (sl_no)
GO
```

```
100 %
Messages
Commands completed successfully.

Completion time: 2020-09-11T02:20:26.1379618+05:30
```

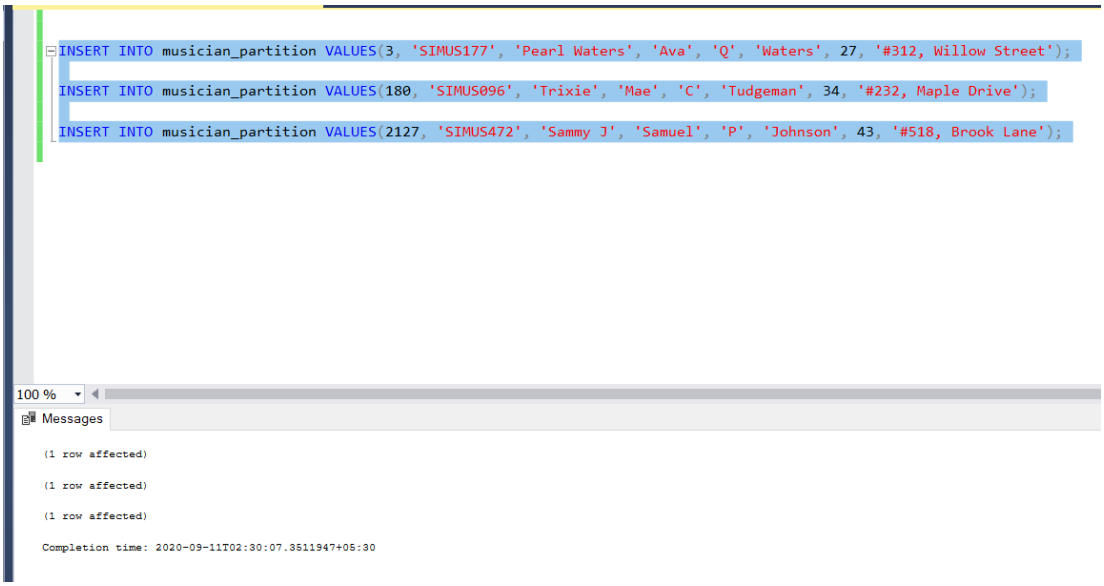
We can see that musician_partition is indeed a partitioned table, as for each object and index id, if there is a count of more than 1, then it means that there are multiple objects with the same index, which is only possible when the table has been partitioned.

```
select distinct object_name(object_id) from sys.partitions
group by object_id, index_id
having COUNT(*) > 1;
```

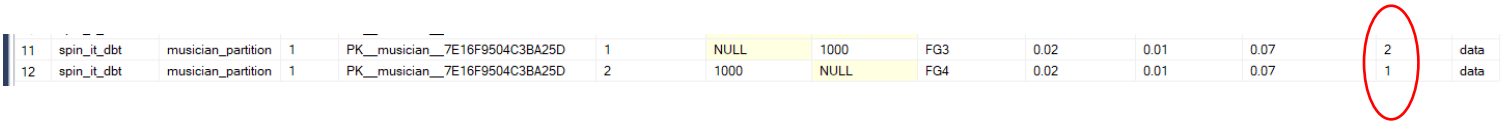
```
100 %
Results  Messages  Execution plan
```

	(No column name)
1	musician_partition

Next, inserting data into this table-



After running a code that displays all information about all the filegroups, we get-



11	spin_it_dbt	musician_partition	1	PK__musician__7E16F9504C3BA25D	1	NULL	1000	FG3	0.02	0.01	0.07	2	data
12	spin_it_dbt	musician_partition	1	PK__musician__7E16F9504C3BA25D	2	1000	NULL	FG4	0.02	0.01	0.07	1	data

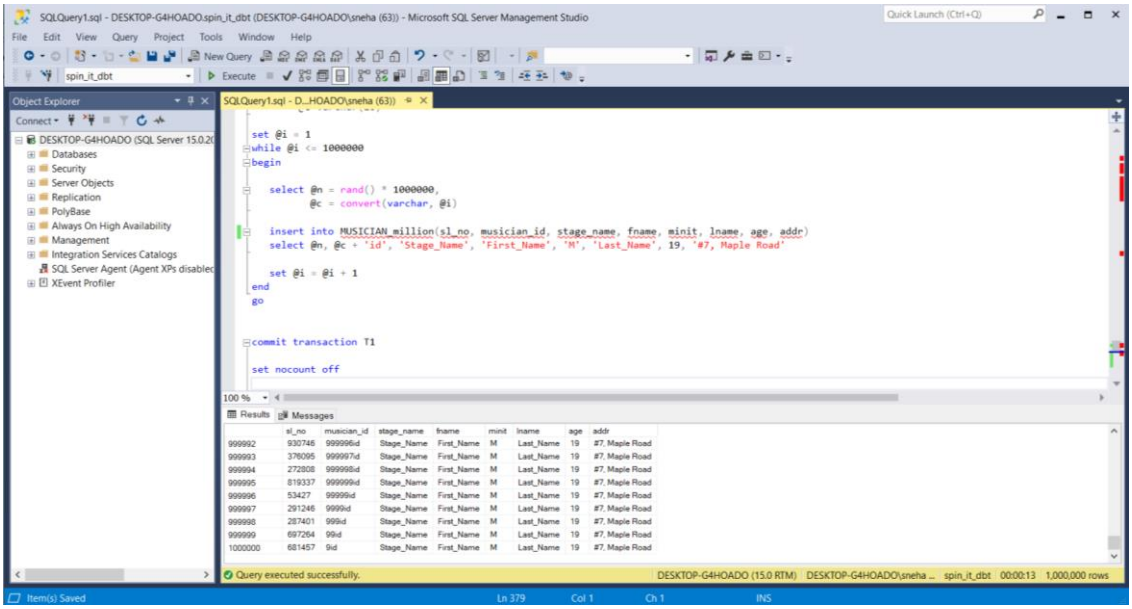
We can see that the row count is as per our logic. The two rows with a sl_no <= 1000 have been placed in the first partition, and the last row, with its sl_no > 1000 has been placed in the second partition.

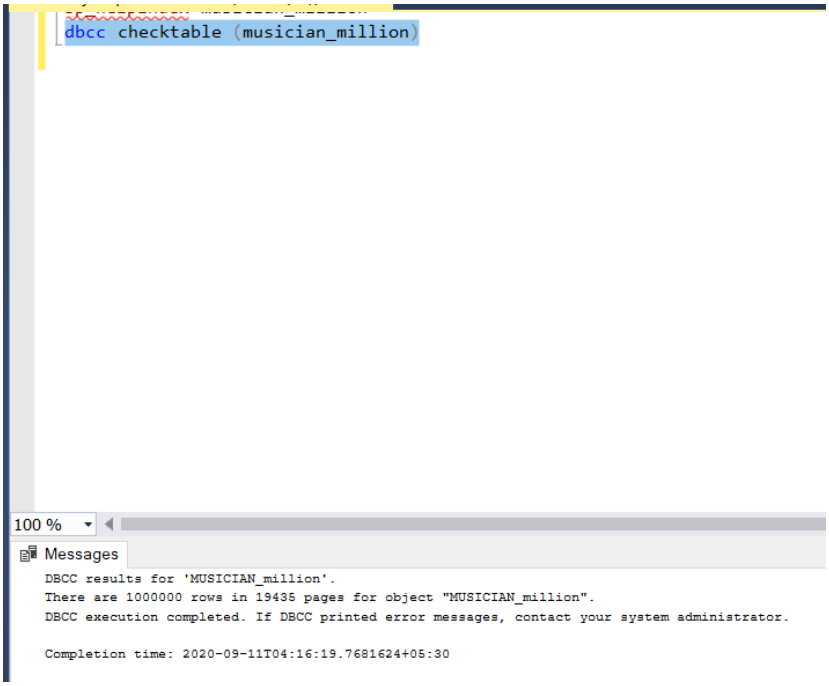
Inserting millions of rows into the table-

We can run the following code-

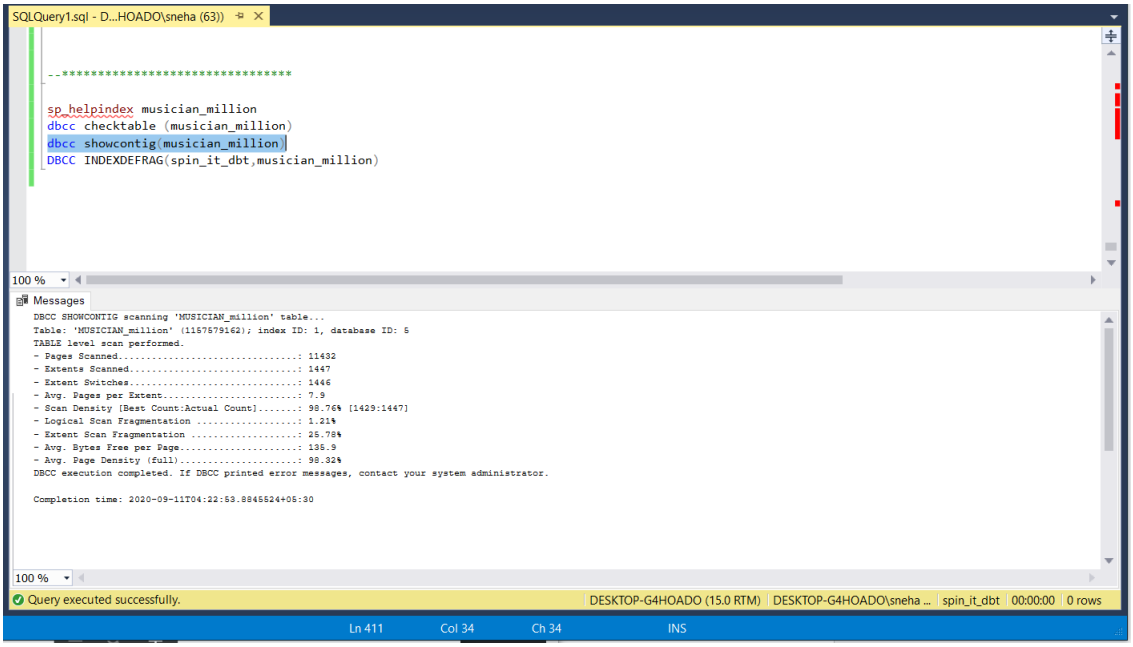
```
set nocount on

select @@TRANCOUNT

begin transaction T1

declare @i int,
        @n numeric(10),
        @c varchar(10)

set @i = 1
```

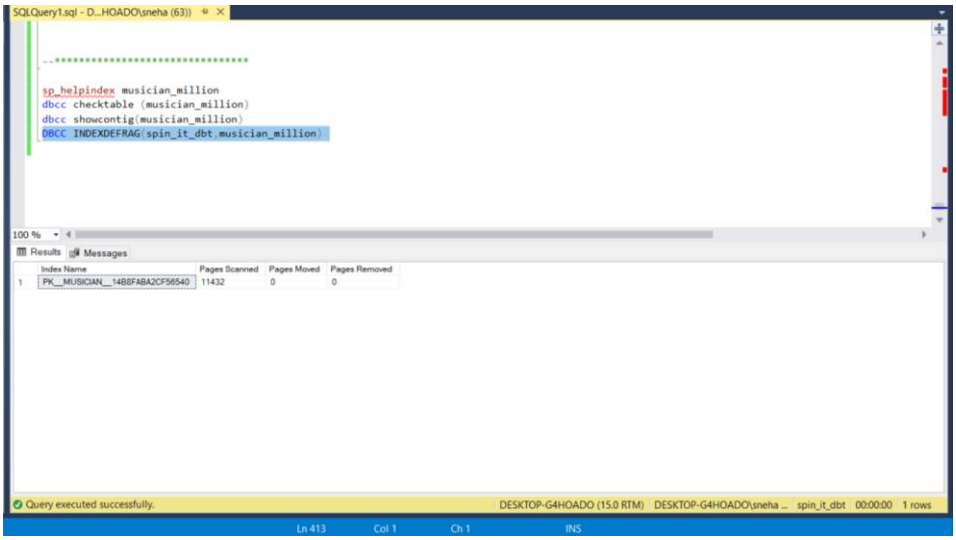
```
while @i <= 1000000
begin

   select @n = rand() * 1000000,
          @c = convert(varchar, @i)

   insert into MUSICIAN_million(sl_no, musician_id, stage_name, fname, minit, lname, age,
addr)
   select @n, @c + 'id', 'Stage_Name', 'First_Name', 'M', 'Last_Name', 19, '#7, Maple
Road'

   set @i = @i + 1
end
go


commit transaction T1

set nocount off
```



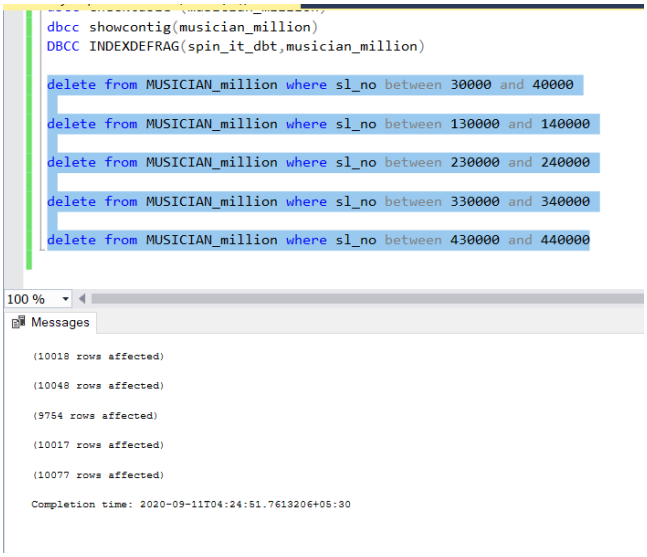


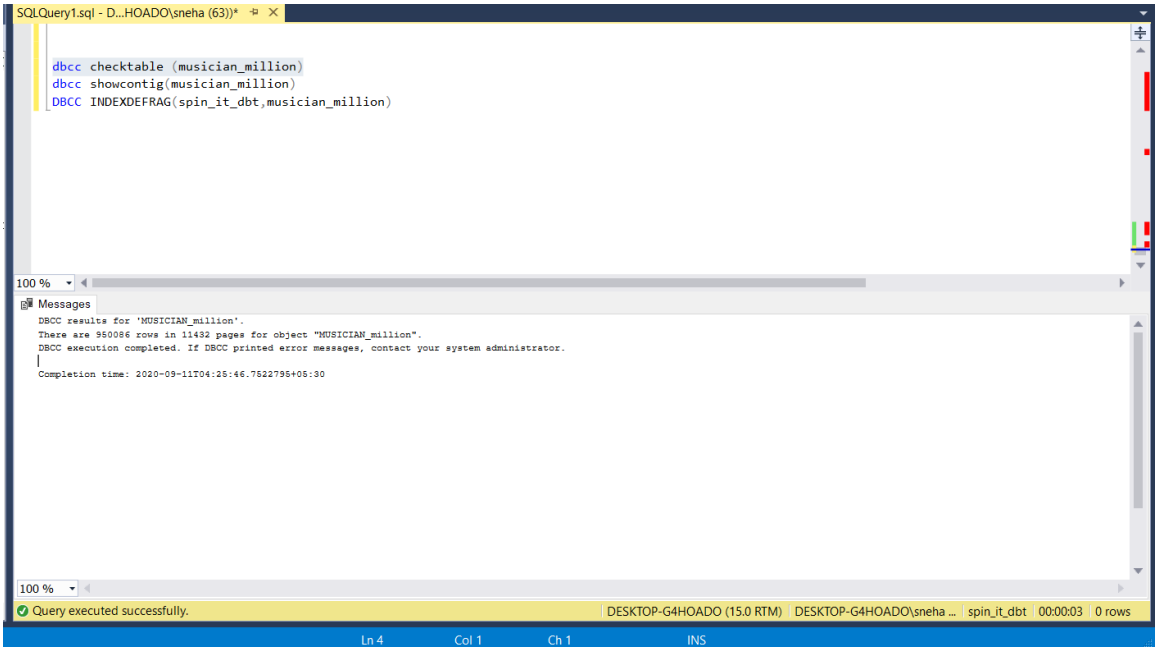
As we can see, 1000000 tuples have been loaded into the table

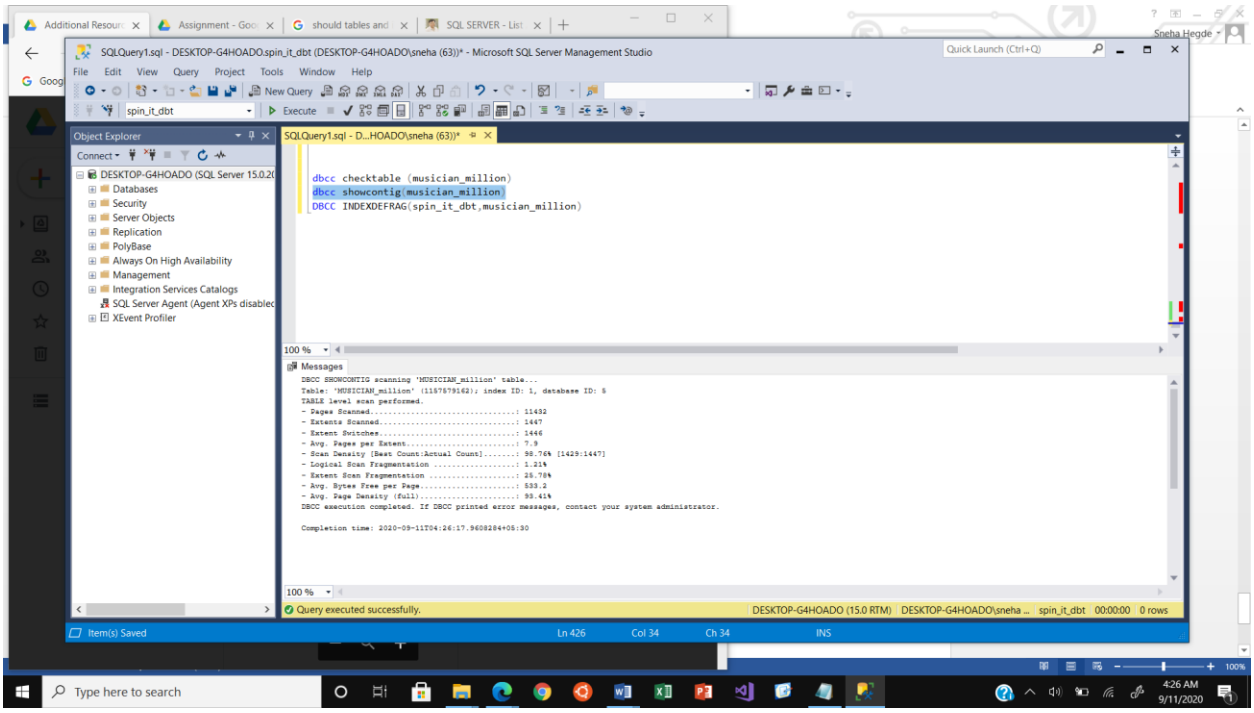


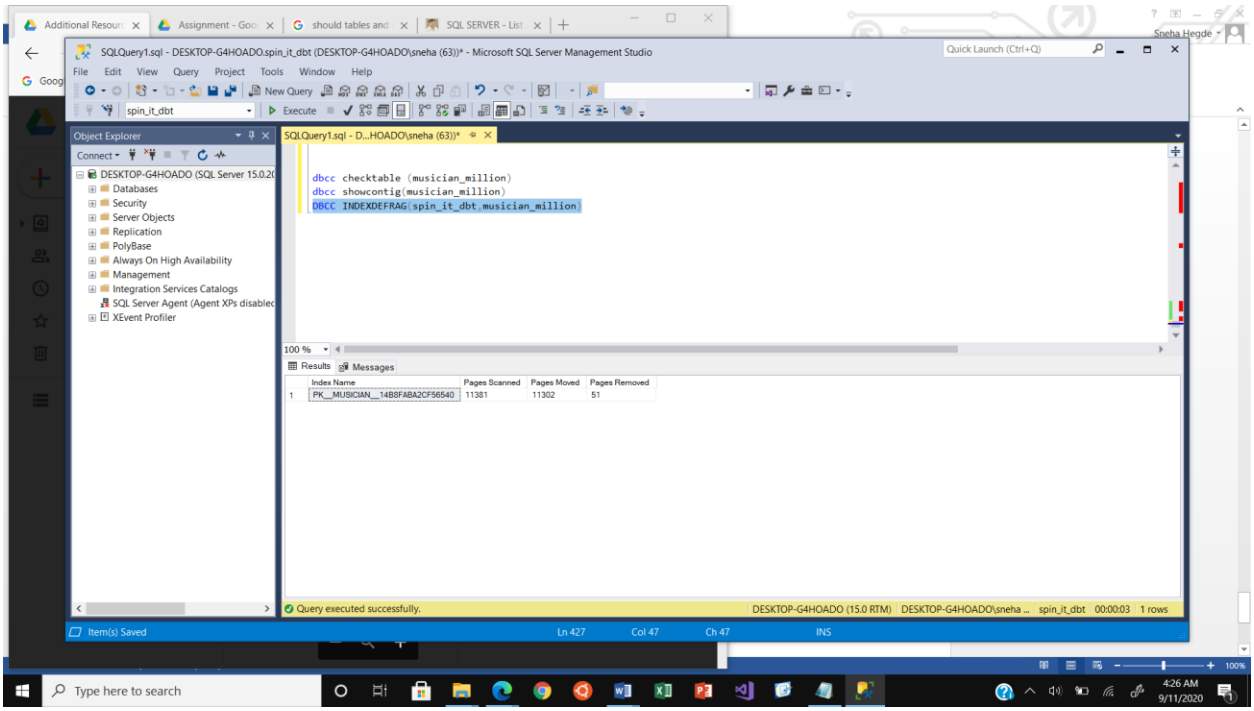


After deleting tuples, we can check the stats again




SQLQuery1.sql - D...HOADO\sneha (63))*
dbcc checktable (musician_million)
dbcc showcontig(musician_million)
DBCC INDEXDEFRAG(spin_it_dbt,musician_million)
100 %
Messages
DBCC results for 'MUSICIAN_million'.
There are 950086 rows in 11432 pages for object "MUSICIAN_million".
DBCC execution completed. If DBCC printed error messages, contact your system administrator.
Completion time: 2020-09-11T04:25:46.7522795+05:30
100 %
Query executed successfully.
DESKTOP-G4HOADO (15.0 RTM)
DESKTOP-G4HOADO\sneha ...
spin_it_dbt
00:00:03
0 rows
Ln 4
Col 1
Ch 1
INS

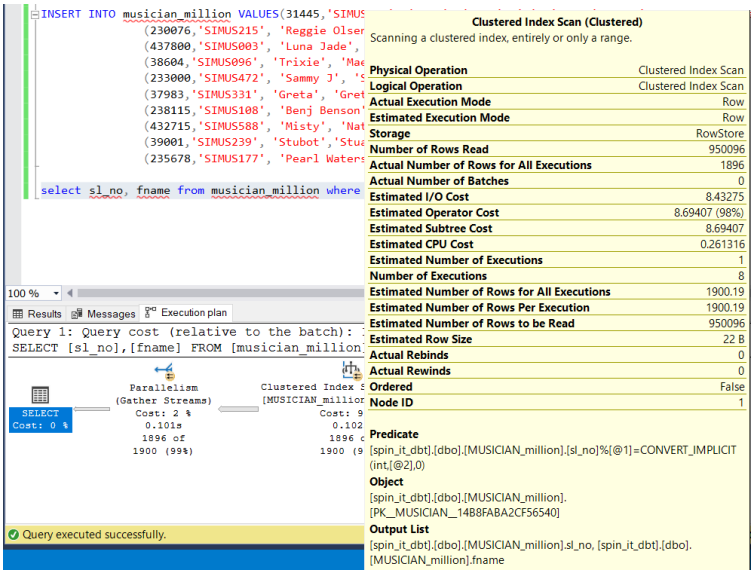

SQLQuery1.sql - DESKTOP-G4HOADO.spin_it_dbt (DESKTOP-G4HOADO\sneha (63))* - Microsoft SQL Server Management Studio
File Edit View Query Project Tools Window Help
New Query
Execute
spin_it_dbt
Object Explorer
Connect
DESKTOP-G4HOADO (SQL Server 15.0.2
Databases
Security
Server Objects
Replication
PolyBase
Always On High Availability
Management
Integration Services Catalogs
SQL Server Agent (Agent XPs disabled
XEvent Profiler
SQLQuery1.sql - D...HOADO\sneha (63))*
dbcc checktable (musician_million)
dbcc showcontig(musician_million)
DBCC INDEXDEFRAG(spin_it_dbt,musician_million)
100 %
Messages
DBCC SHOWCONTIG scanning 'MUSICIAN_million' table...
Table: 'MUSICIAN_million' (1157679162); index ID: 1, database ID: 5
TABLE level scan performed.
- Pages Scanned.....: 11432
- Extents Scanned.....: 1447
- Extent Switches.....: 1446
- Avg. Pages per Extent.....: 7.9
- Scan Density [Best Count:Actual Count].....: 98.76% [1429:1447]
- Logical Scan Fragmentation .....: 1.21%
- Extent Scan Fragmentation .....: 25.78%
- Avg. Bytes Free per Page.....: 533.2
- Avg. Page Density (full).....: 93.41%
DBCC execution completed. If DBCC printed error messages, contact your system administrator.
Completion time: 2020-09-11T04:26:17.9608284+05:30
100 %
Query executed successfully.
DESKTOP-G4HOADO (15.0 RTM)
DESKTOP-G4HOADO\sneha ...
spin_it_dbt
00:00:00
0 rows
Item(s) Saved
Ln 426
Col 34
Ch 34
INS
Type here to search
4:26 AM
9/11/2020



Now, let's review the query execution plan before and after creating indexes for a few types of queries-

- `select sl_no, fname from musician_million where sl_no%500=0;`

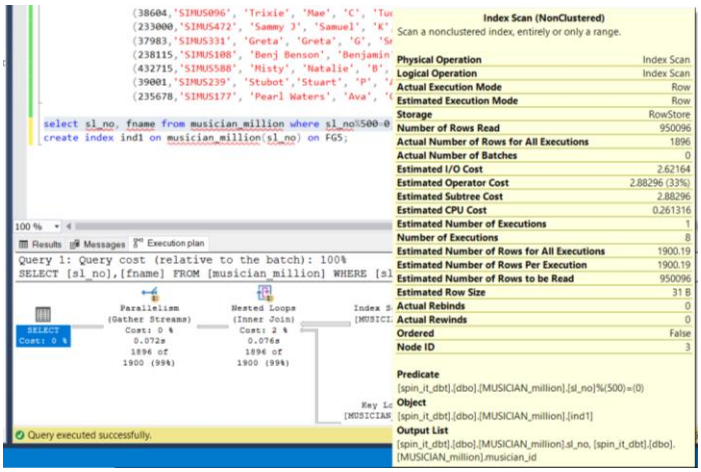
  before creating index:



We can see that the estimated I/O cost is 8.43275. Let's see if we can create an index and reduce it.

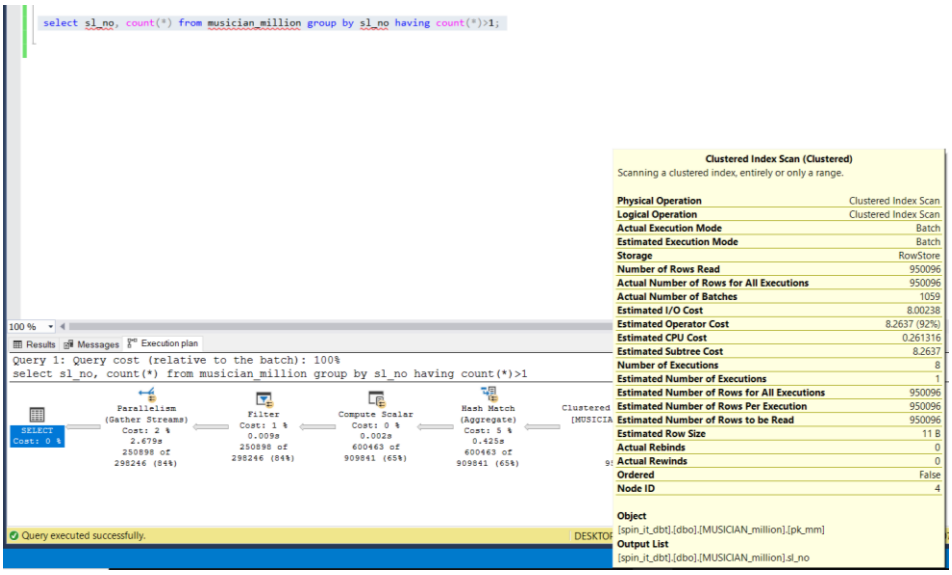
```
create index ind1 on musician_million(sl_no) on FG5;
```

Creating an index on sl_no, we get-



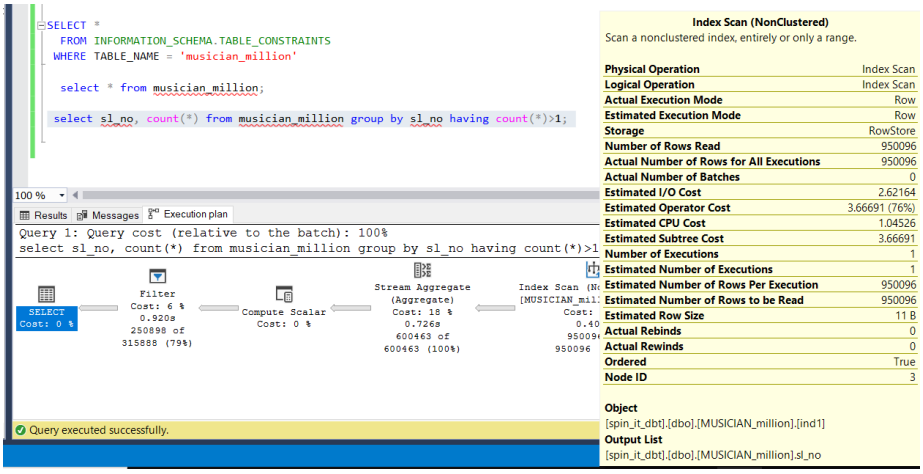
2.62, which is considerably better, as if we create an index on the search attribute (sl_no), the access will be much faster, as it wouldn't have to check the data file on the disk each time, and can just refer to the index pointers.

- Trying out a query using an aggregate function,
  - `select sl_no, count(*) from musician_million group by sl_no having count(*)>1;`



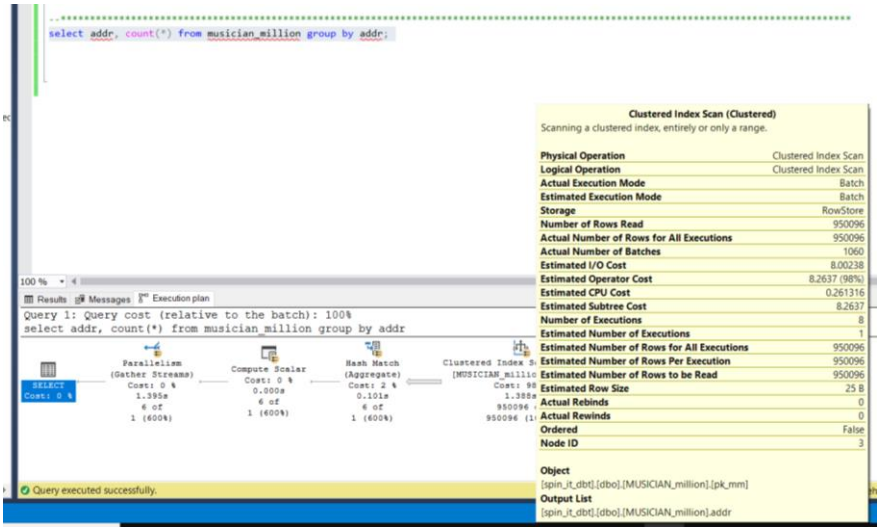
Cost is 8.00238

Creating the same index as before (on sl_no), it brings it down to 2.621



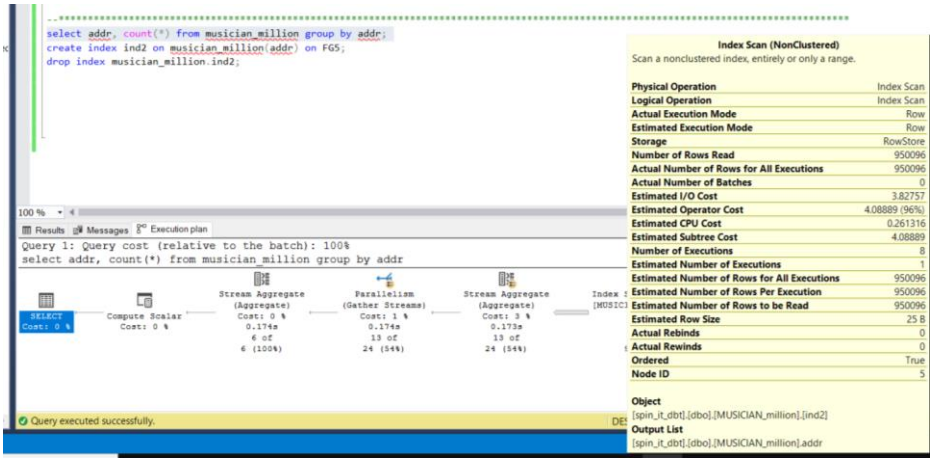
- Another example-

```
select addr, count(*) from musician_million group by addr;
```



Creating another index, on the attribute addr-

```
create index ind2 on musician_million(addr) on FG5;
```

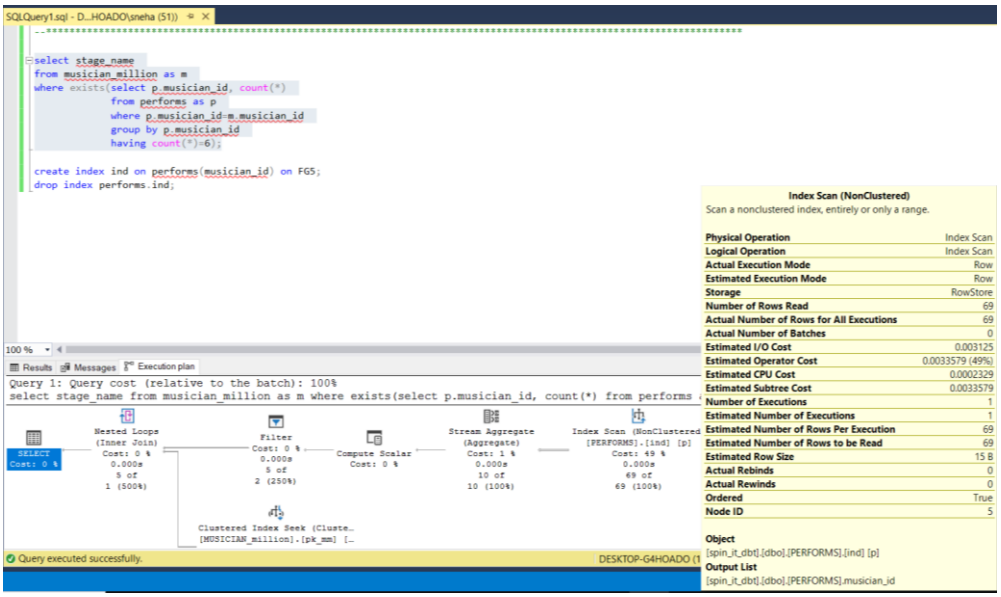


The cost is brought down to 3.8

-
  - For a correlated nested query,

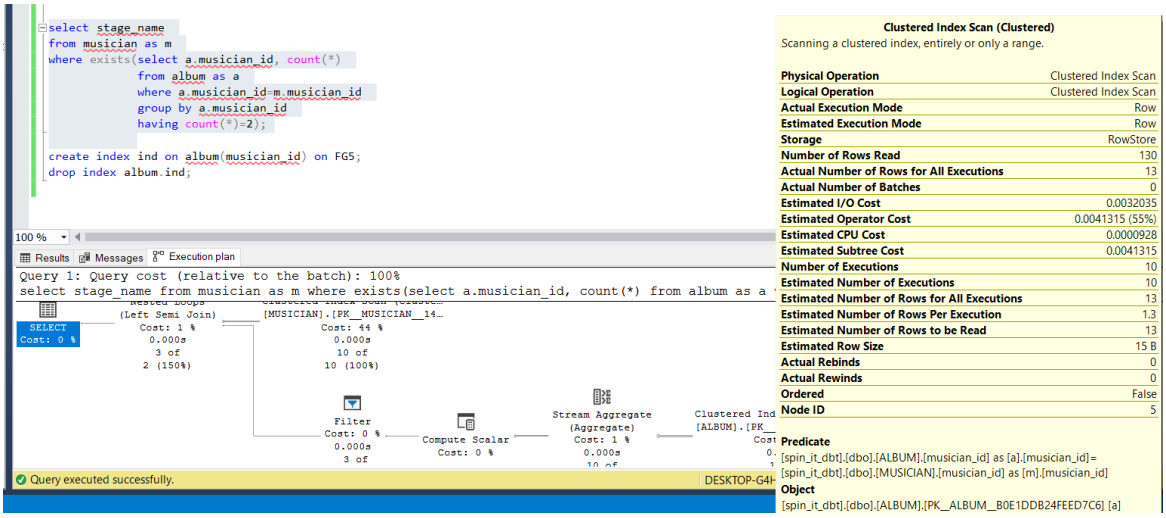
```
select stage_name
from musician_million as m
where exists(select p.musician_id, count(*)
                    from performs as p
                    where p.musician_id=m.musician_id
                    group by p.musician_id
                    having count(*)=6);
```

by default, clustered indexes are created on the primary key of a table. Hence, in this specific example, the cost wouldn't change, irrespective of whether we create an index. In a correlated subquery, the inner query gets executed once for every tuple in the outer query. Hence, it makes sense to have indexes on the attributes in the inner query, which gets repeated multiple times. That is done by default in this example.



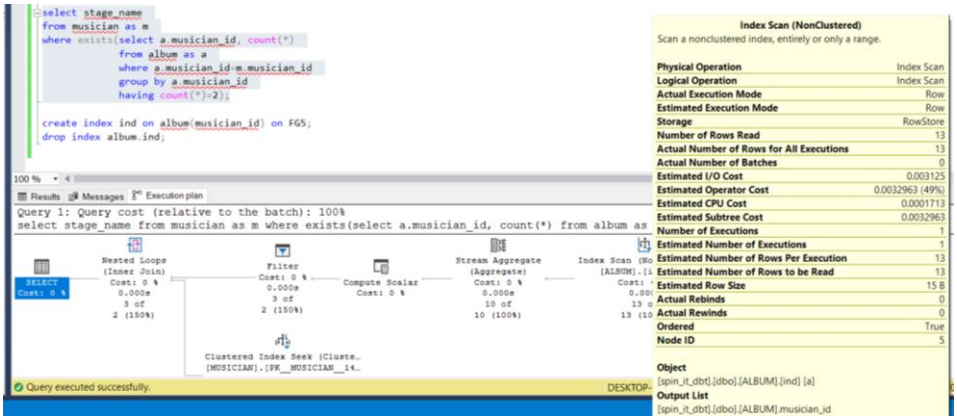
- Taking another example,

```
select stage_name
from musician as m
where exists(select a.musician_id, count(*)
             from album as a
             where a.musician_id=m.musician_id
             group by a.musician_id
             having count(*)=2);
```



The cost is 0.0032035.

After creating an index on album's musician_id,



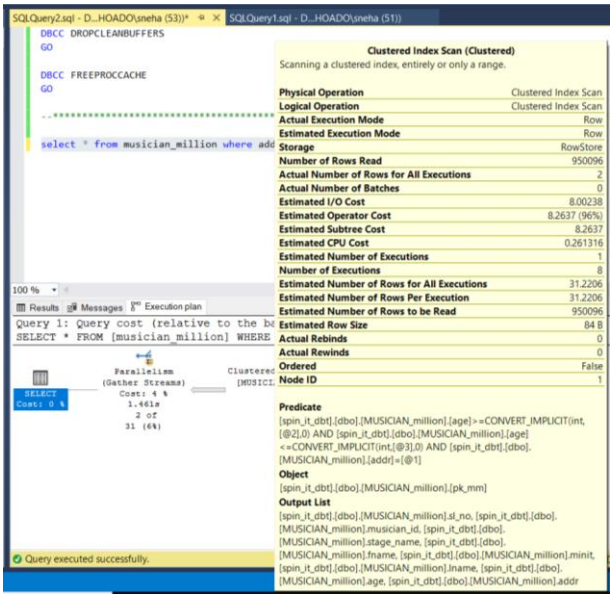
the cost reduces to 0.003125.

The change may not be significant. This is because album has a smaller number of tables (if it had a larger number of tables, the change would've been more prominent, but my system kept hanging when I tried to insert another million, so I've demonstrated the impact on a smaller scale. Theoretically, it'd work in the same way)

- Testing out a composite index-

```
select * from musician_million where addr='#312, Willow Street' and age between 20 and 40;
```

We get a higher cost of 8.002

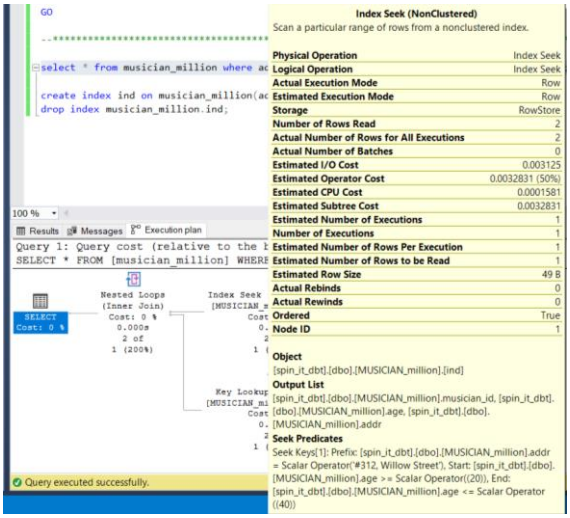


If we create a composite index on the attributes addr and age,

```
create index ind on musician_million(addr,age) on FG5;
```

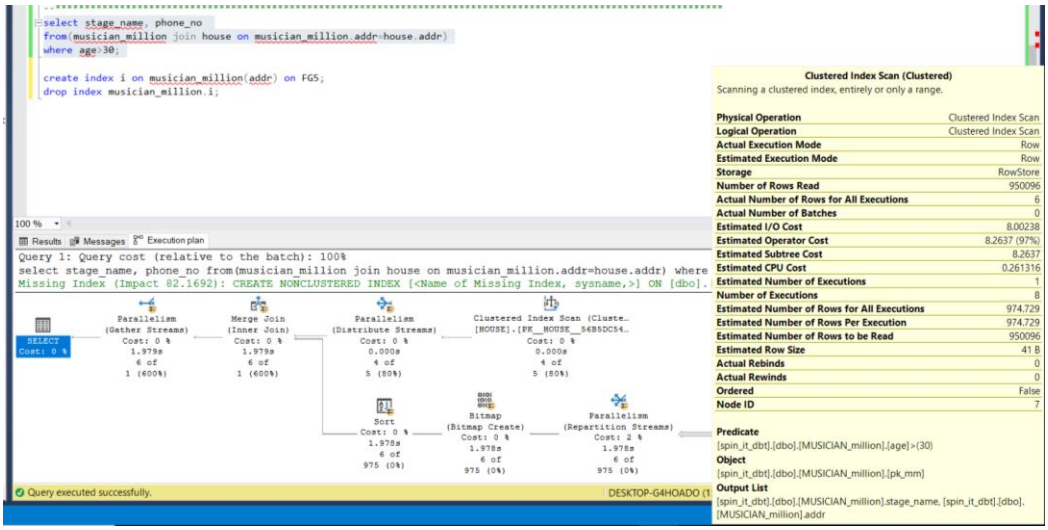
the cost gets significantly reduced (to 0.003)

as in this case, it finds all the pointers pointing to the address '#312, Willow Street' through the index on the addr attribute. It doesn't bring their tuples to the memory from the disk just yet. Instead, it also searches for all the pointers which point to tuples where the age is between 20 and 40. It intersects the pointers to produce common ones, ie, pointers which point to tuples where the addr is '#312, Willow Street' and where the age is between 2 and 40. It only retrieves those tuples, therefore, eliminating the number of disk operations.



- Testing a join-
  If we have

```
select stage_name, phone_no
from(musician_million join house on musician_million.addr=house.addr)
      where age>30;
```



the search would be tedious in the musician_million table. Therefore, it makes sense to create an index on the addr attribute, which is the join attribute. It is already an index in the house table on addr (as it is the pk), so we have to create it on the musician_million table.

```
create index i on musician_million(addr) on FG5;
```

This brings it down significantly.