

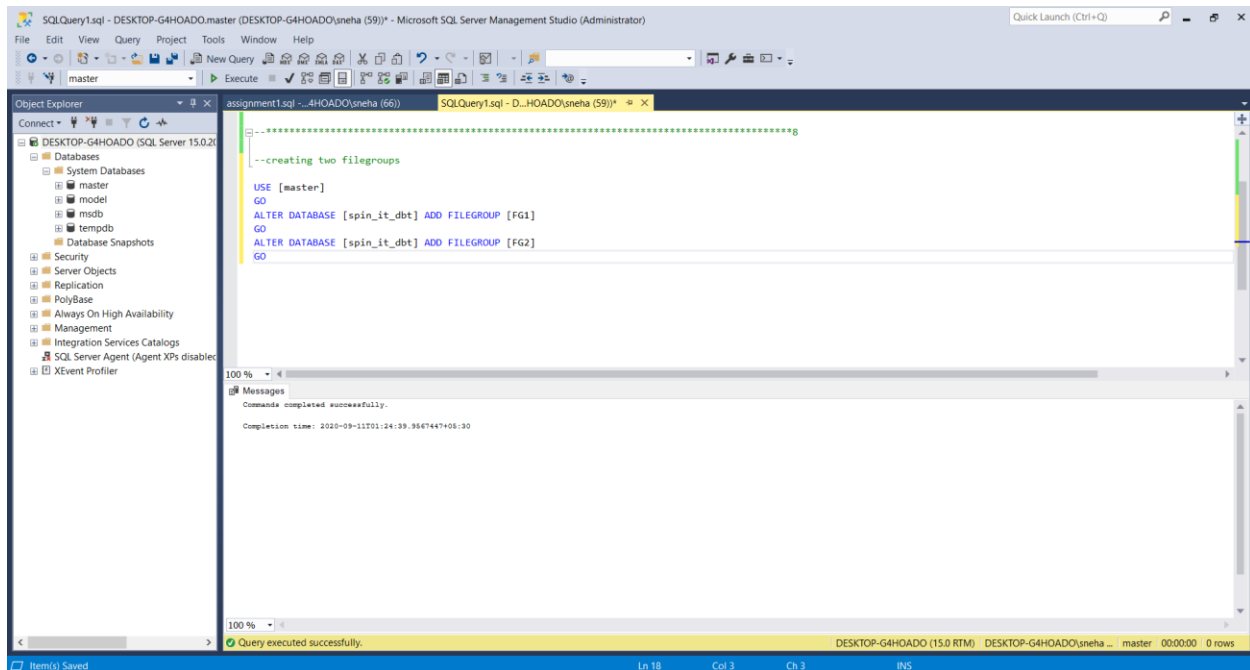
# DBT ASSIGNMENT 1

Sneha Hegde

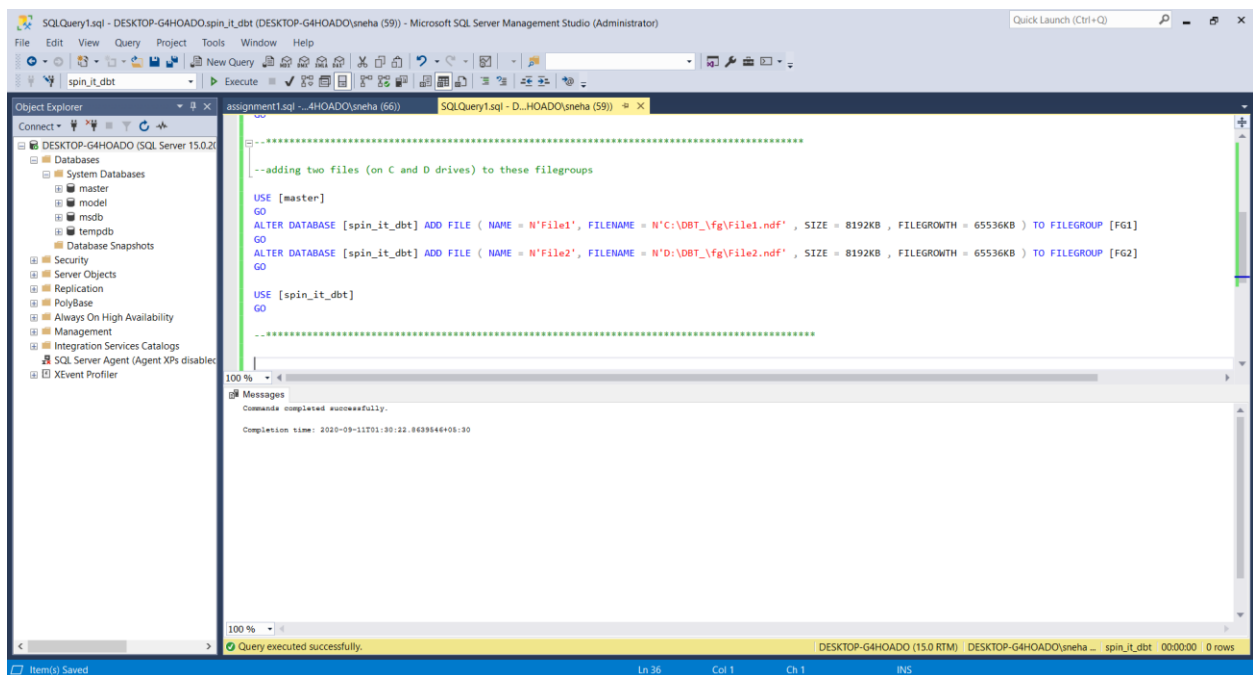
PES1201801157

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.

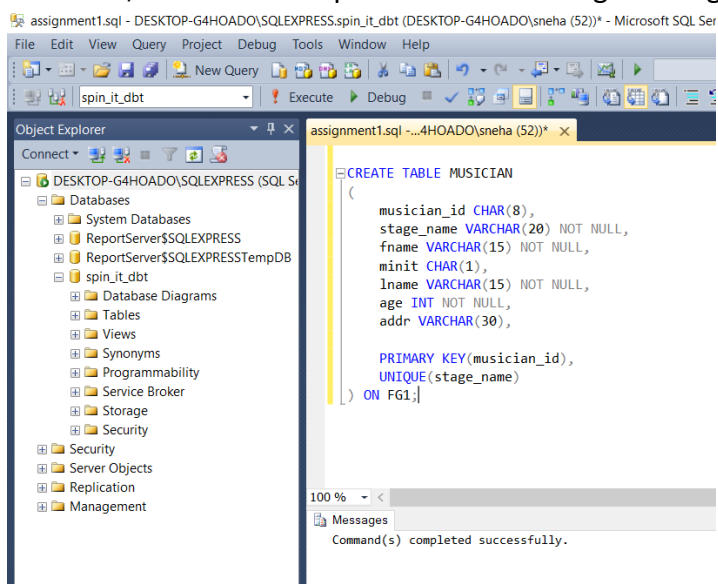
Adding filegroups to the database-



## 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 screenshot shows the Microsoft SQL Server Management Studio (SSMS) interface. The left pane displays the Object Explorer with the 'DESKTOP-G4HOADO (SQL Server 15.0.20168.1)' server selected. The right pane shows a query window with the following SQL query:

```
--*****  
SELECT  
FROM sys.filegroups  
GO
```

The query results are displayed in the lower right pane, showing a table with 10 columns: name, data\_space\_id, type, type\_desc, is\_default, is\_system, filegroup\_guid, log\_filegroup\_id, is\_read\_only, and is\_autogrow\_all\_files. The results are as follows:

name	data_space_id	type	type_desc	is_default	is_system	filegroup_guid	log_filegroup_id	is_read_only	is_autogrow_all_files
PRIMARY	1	FG	ROWS_FILEGROUP	1	0	NULL	NULL	0	0
FG1	2	FG	ROWS_FILEGROUP	0	0	835225E2-DBAA-4AF8-800E-F189C89D00C8	NULL	0	0
FG2	3	FG	ROWS_FILEGROUP	0	0	E0C4B25C-BFE4-48A2-AAAD-BCCDD7FAB6E3	NULL	0	0

The status bar at the bottom indicates that the query was executed successfully, returning 3 rows in 00:00:00. The status bar also shows the current file (Ln 150, Col 3, Ch 3) and the server name (DESKTOP-G4HOADO (15.0 RTM)).

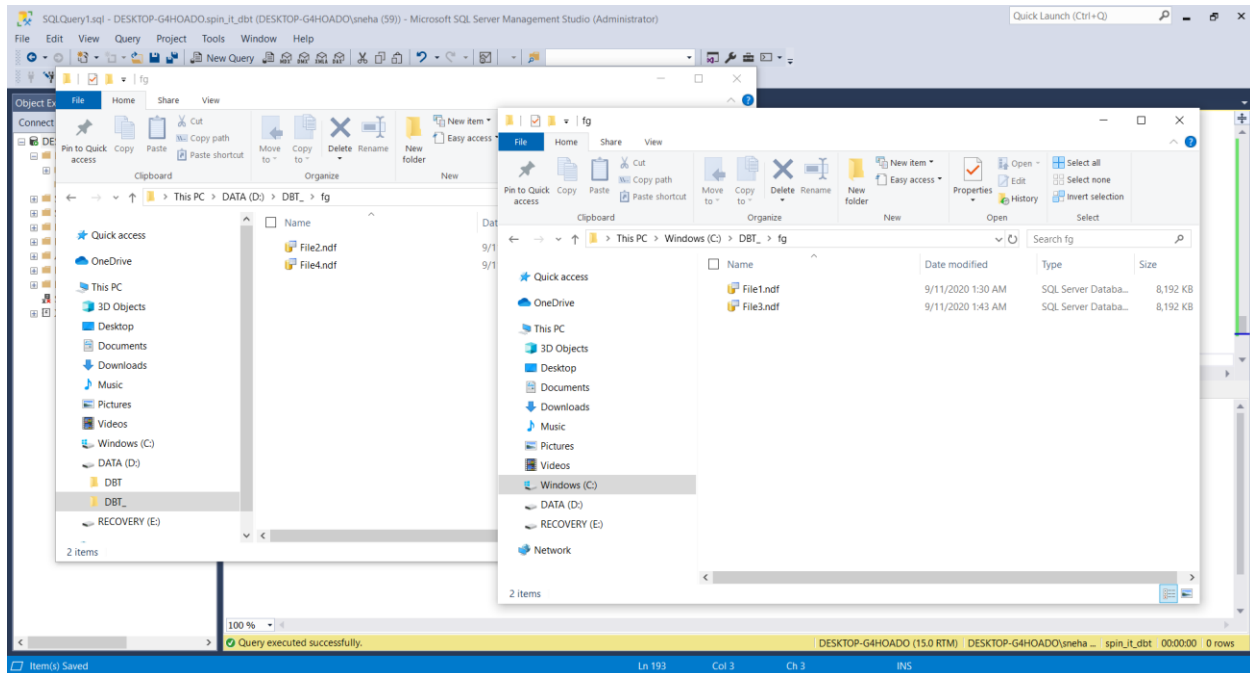
The first screenshot shows a query in the 'spin\_it\_dbt' database that selects file information from 'sys.filegroups' and 'sys.database\_files' where the type\_desc is 'ROWS'. The results show three files: 'spin\_it\_dbt', 'File1', and 'File2'.

The second screenshot shows a query in the 'assignment1.sql' database that creates a table 'MUSICIAN' with columns 'name', 'type', 'name', 'index\_id', and 'name'. The query uses a complex join to select data from 'sys.indexes', 'sys.filegroups', and 'sys.all\_objects' where the data\_space\_id is 2 (Filegroup). The results show six rows of data for the 'MUSICIAN' table.

The third screenshot shows a query in the 'assignment1.sql' database that creates a table 'INSTRUMENT' with columns 'name', 'type', 'name', 'index\_id', and 'name'. The query uses a complex join to select data from 'sys.indexes', 'sys.filegroups', and 'sys.all\_objects' where the data\_space\_id is 3 (Filegroup). The results show four rows of data for the 'INSTRUMENT' table.

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

```
CREATE PARTITION SCHEME siPS1
AS PARTITION siPF1
TO (FG3, FG4) ;
GO
```

100 %

Messages

Commands completed successfully.

Completion time: 2020-09-11T02:11:06.7762402+05:30

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-

```
INSERT INTO musician_partition VALUES(3, 'SIMUS177', 'Pearl Waters', 'Ava', 'Q', 'Waters', 27, '#312, Willow Street');
INSERT INTO musician_partition VALUES(180, 'SIMUS096', 'Trixie', 'Mae', 'C', 'Tudgeman', 34, '#232, Maple Drive');
INSERT INTO musician_partition VALUES(2127, 'SIMUS472', 'Sammy J', 'Samuel', 'P', 'Johnson', 43, '#518, Brook Lane');
```

100 %

Messages

(1 row affected)

(1 row affected)

(1 row affected)

Completion time: 2020-09-11T02:30:07.3511947+05:30

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)

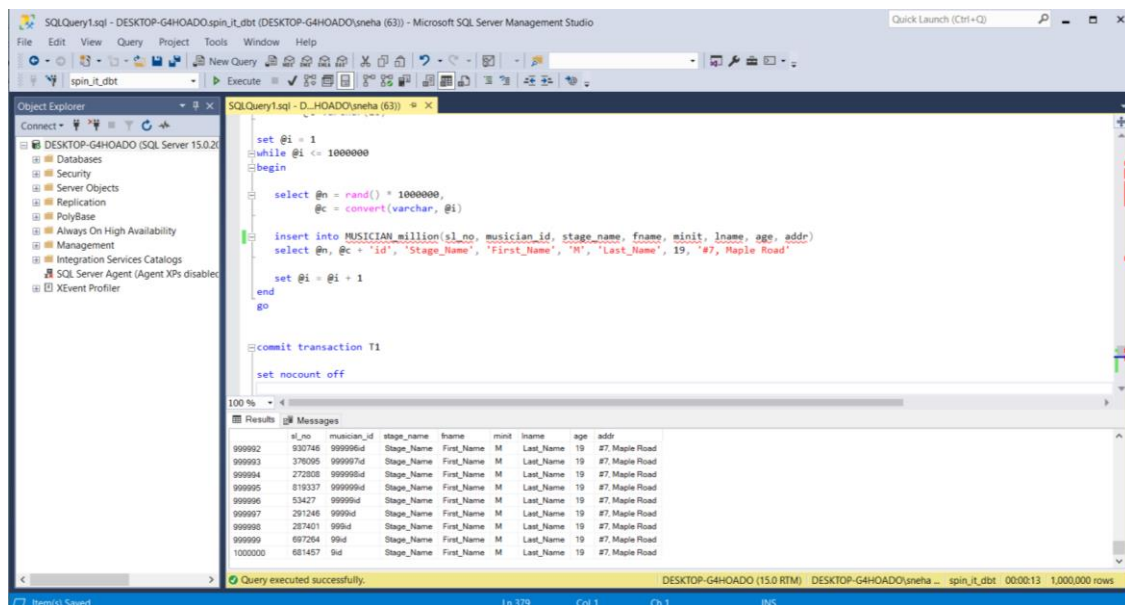
    insert into MUSICIAN_million(sl_no, musician_id, stage_name, fname, minit, lname, age,
addr)
    select @n, @c + 'id', 'Stage_Name', 'First_Name', 'M', 'Last_Name', 19, '#7, Maple
Road'

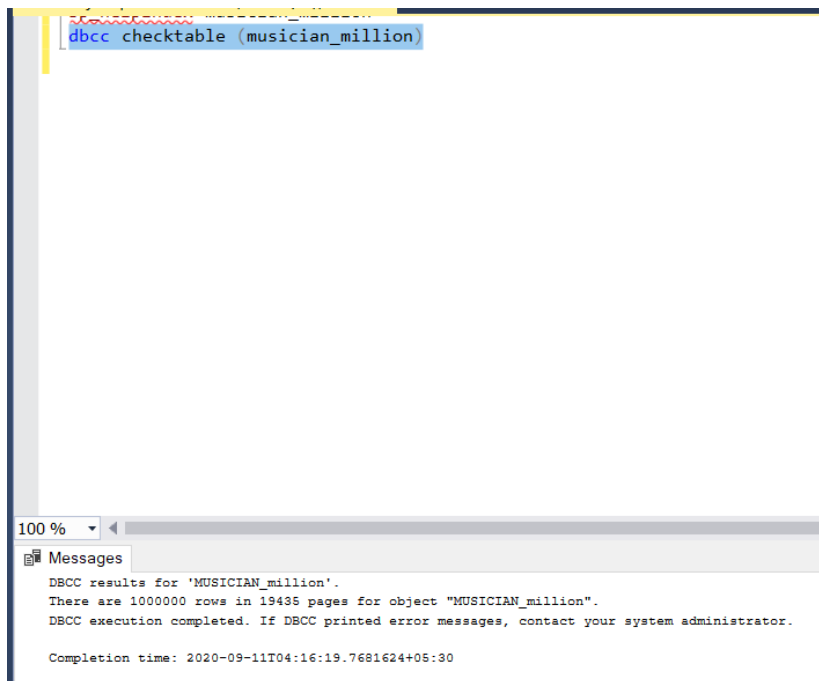
    set @i = @i + 1
end
go

commit transaction T1

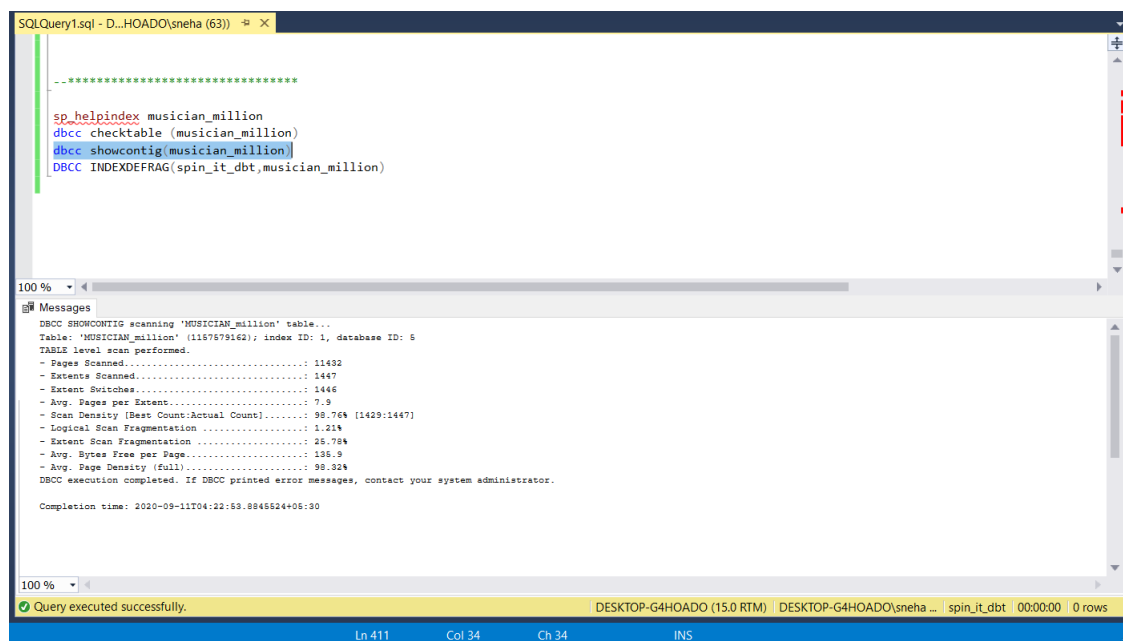
set nocount off

```





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



```
SQLQuery1.sql - D:\HOADO\neha (63)
.....
sp_helpindex musician_million
dbcc checktable (musician_million)
dbcc showcontig(musician_million)
DBCC INDEXDEFRAG(sp_in_it_dbt,musician_million)
```

Index Name	Pages Scanned	Pages Moved	Pages Removed
PK_MUSICIAN_148FABA2CF56540	11432	0	0

Query executed successfully.

DESKTOP-G4HOADO (15.0 RTM) | DESKTOP-G4HOADO\neha ... | spin\_it\_dbt | 00:00:00 | 1 rows

Ln 413 Col 1 Ch 1 INS

After deleting tuples, we can check the stats again

```
dbcc showcontig(musician_million)
DBCC INDEXDEFRAG(sp_in_it_dbt,musician_million)

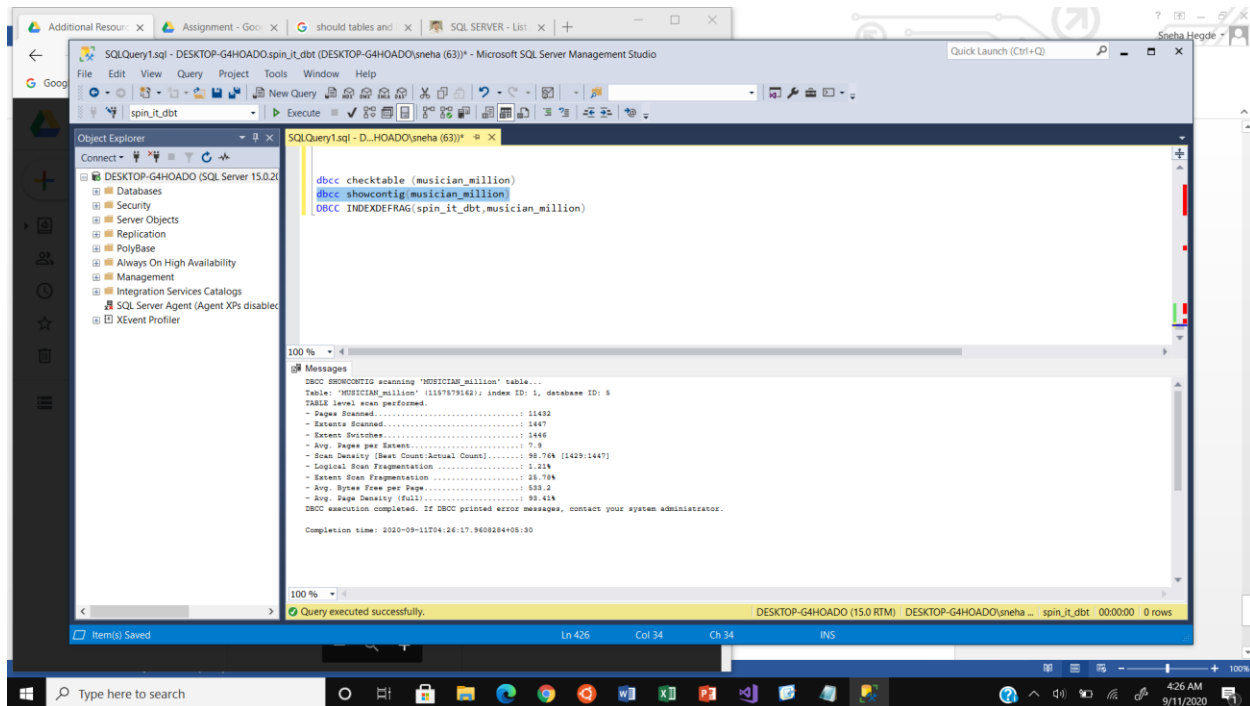
delete from MUSICIAN_million where sl_no between 30000 and 40000
delete from MUSICIAN_million where sl_no between 130000 and 140000
delete from MUSICIAN_million where sl_no between 230000 and 240000
delete from MUSICIAN_million where sl_no between 330000 and 340000
delete from MUSICIAN_million where sl_no between 430000 and 440000
```

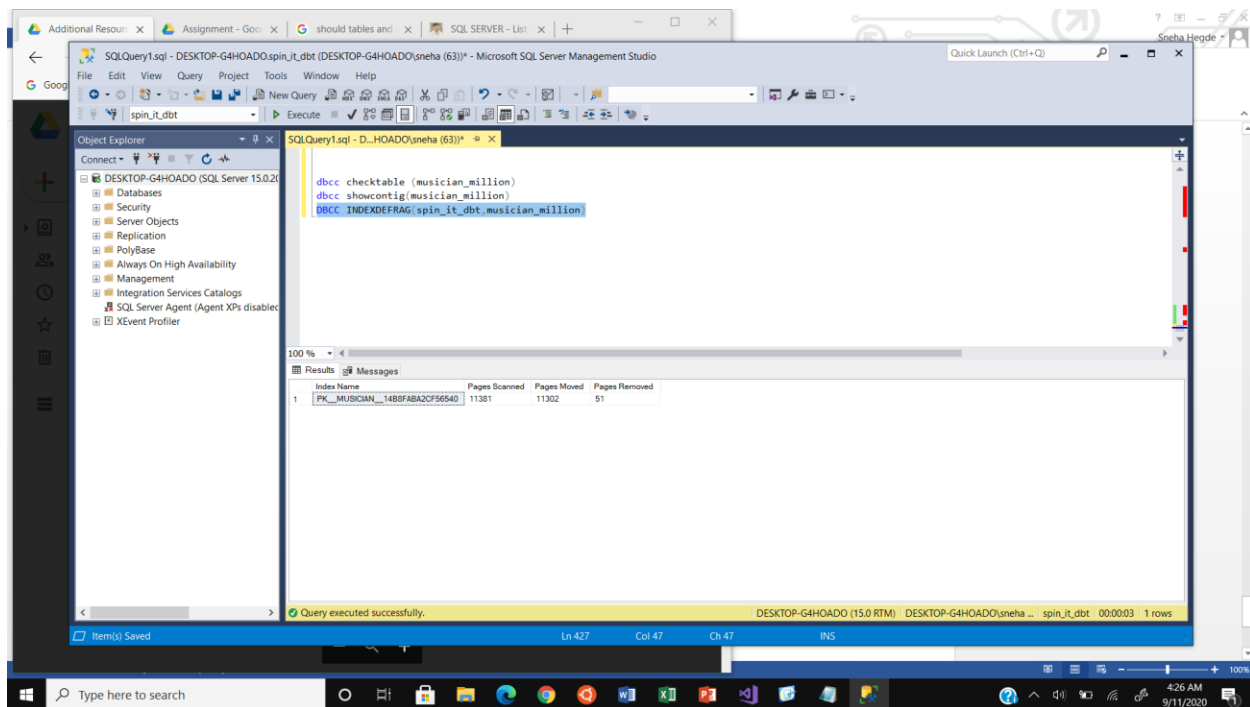
100 %

Messages

(10018 rows affected)  
(10048 rows affected)  
(9754 rows affected)  
(10017 rows affected)  
(10077 rows affected)

Completion time: 2020-09-11T04:24:51.7613206+05:30





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:

`INSERT INTO musician_million VALUES(31445,'SIMUS215','Reggie Olser')`  
`(230076,'SIMUS215','Reggie Olser')`  
`(437800,'SIMUS003','Luna Jade')`  
`(38604,'SIMUS096','Trixie','Mac')`  
`(233000,'SIMUS472','Sammy J','Gret')`  
`(37983,'SIMUS331','Greta','Gret')`  
`(238115,'SIMUS108','Benj Benson')`  
`(432715,'SIMUS588','Misty','Nat')`  
`(39001,'SIMUS239','Stubot','Stubot')`  
`(235678,'SIMUS177','Pearl Waters')`

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

**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	950096
Actual Number of Rows for All Executions	1896
Actual Number of Batches	0
Estimated I/O Cost	8.43275
Estimated Operator Cost	8.69407 (98%)
Estimated Subtree Cost	8.69407
Estimated CPU Cost	0.261316
Estimated Number of Executions	1
Number of Executions	8
Estimated Number of Rows for All Executions	1900.19
Estimated Number of Rows Per Execution	1900.19
Estimated Number of Rows to be Read	950096
Estimated Row Size	22 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Node ID	1

**Predicate**  
`[spin_it_dbt].[dbo].[MUSICIAN_million].[sl_no]%(1)=CONVERT_IMPLICIT(int,@2),0`

**Object**  
`[spin_it_dbt].[dbo].[MUSICIAN_million].`  
`[PK_MUSICIAN_1488FABA2CF56540]`

**Output List**  
`[spin_it_dbt].[dbo].[MUSICIAN_million].[sl_no], [spin_it_dbt].[dbo].[MUSICIAN_million].[fname]`

Query executed successfully.

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

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

```

SELECT *
FROM INFORMATION_SCHEMA.TABLE_CONSTRAINTS
WHERE TABLE_NAME = 'musician_million'

select * from musician_million;

select sl_no, count(*) from musician_million group by sl_no having count(*)>1;

```

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

```

select sl_no, count(*) from musician_million group by sl_no having count(*)>1;

```

Query executed successfully.

**Index Scan (NonClustered)**  
Scan a nonclustered index, entirely or only a range.

Physical Operation	Index Scan
Logical Operation	Index Scan
Actual Execution Mode	Row
Estimated Execution Mode	Row
Storage	RowStore
Number of Rows Read	950096
Actual Number of Rows for All Executions	950096
Actual Number of Batches	0
Estimated I/O Cost	2.62164
Estimated Operator Cost	3.66691 (76%)
Estimated CPU Cost	1.04526
Estimated Subtree Cost	3.66691
Number of Executions	1
Estimated Number of Executions	1
Estimated Number of Rows Per Execution	950096
Estimated Number of Rows to be Read	950096
Estimated Row Size	11 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	3

Object  
[spin\_it\_dbt].[dbo].[MUSICIAN\_million].[ind1]  
Output List  
[spin\_it\_dbt].[dbo].[MUSICIAN\_million].sl\_no

- Another example-

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

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

```

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

```

Query executed successfully.

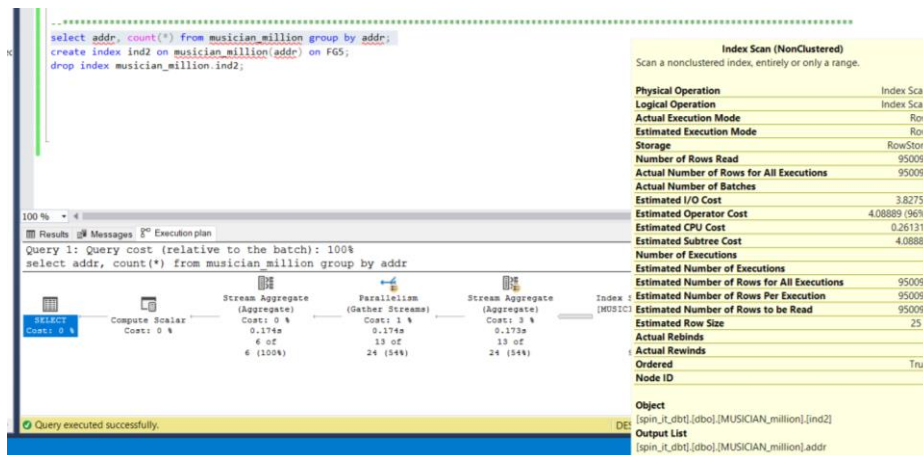
**Clustered Index Scan (Clustered)**  
Scanning a clustered index, entirely or only a range.

Physical Operation	Clustered Index Scan
Logical Operation	Clustered Index Scan
Actual Execution Mode	Batch
Estimated Execution Mode	Batch
Storage	RowStore
Number of Rows Read	950096
Actual Number of Rows for All Executions	950096
Actual Number of Batches	1060
Estimated I/O Cost	8.00238
Estimated Operator Cost	8.2637 (98%)
Estimated CPU Cost	0.261316
Estimated Subtree Cost	8.2637
Number of Executions	8
Estimated Number of Executions	1
Estimated Number of Rows for All Executions	950096
Estimated Number of Rows Per Execution	950096
Estimated Number of Rows to be Read	950096
Estimated Row Size	25 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Node ID	3

Object  
[spin\_it\_dbt].[dbo].[MUSICIAN\_million].[pk\_mm]  
Output List  
[spin\_it\_dbt].[dbo].[MUSICIAN\_million].addr

Creating another index, on the attribute addr-

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



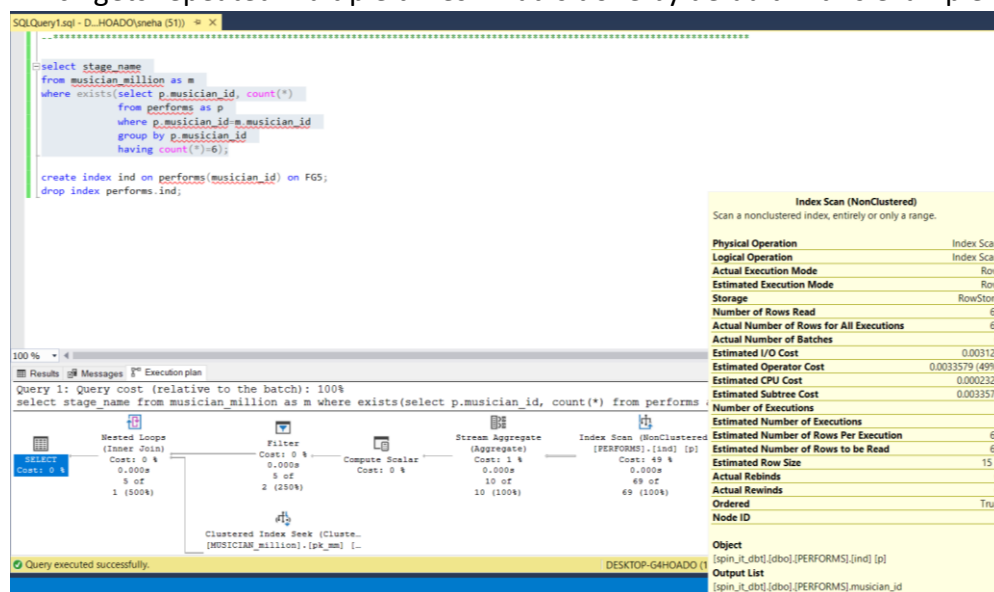
The cost is brought down to 3.8

○

- 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);
```

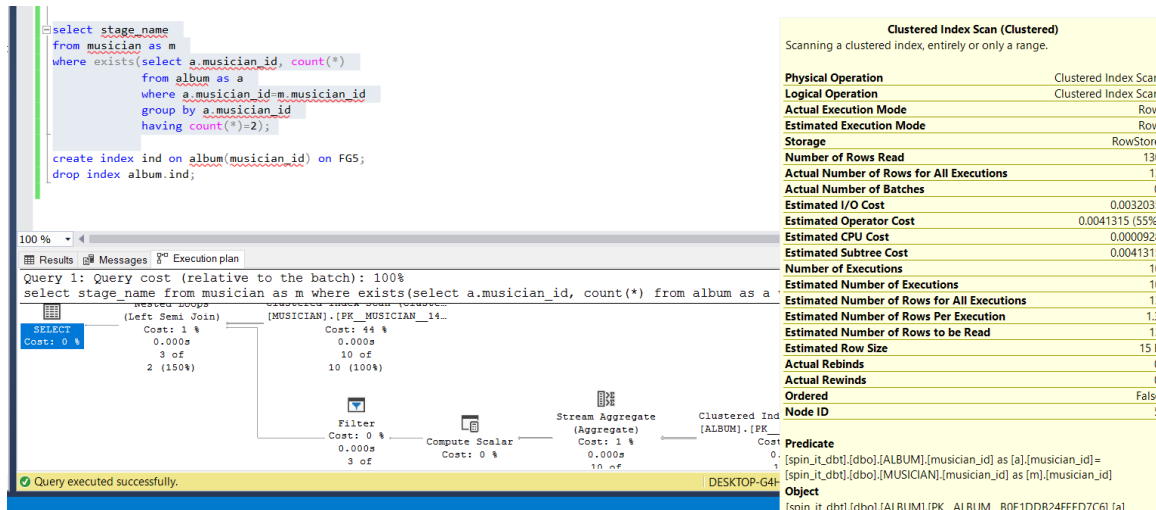
by default, clustered indexes are created on the primary key of a table. Hence, in this specific example, the cost wouldn't change, irrespective of whether we create an index. In a correlated subquery, the inner query gets executed once for every tuple in the outer query. Hence, it makes sense to have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



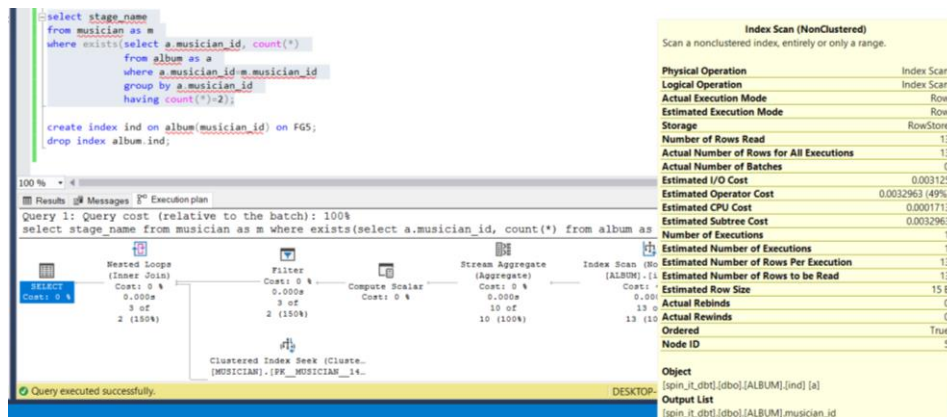
The cost is 0.0032035.

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

The screenshot displays the SQL Server Enterprise Manager interface. The top pane shows the query: `select * from musician_million where addr='#312, Willow Street' and age between 20 and 40;`. The bottom pane shows the execution plan, which is a 'Clustered Index Scan (Clustered)'. The statistics pane on the right provides the following data:

Physical Operation	Clustered Index Scan
Logical Operation	Clustered Index Scan
Actual Execution Mode	Row
Estimated Execution Mode	Row
Storage	RowStore
Number of Rows Read	950096
Actual Number of Rows for All Executions	2
Actual Number of Batches	0
Estimated I/O Cost	8.00238
Estimated Operator Cost	8.2637 (96%)
Estimated Subtree Cost	8.2637
Estimated CPU Cost	0.261316
Estimated Number of Executions	1
Number of Executions	8
Estimated Number of Rows for All Executions	31.2206
Estimated Number of Rows Per Execution	31.2206
Estimated Number of Rows to be Read	950096
Estimated Row Size	84.8
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Node ID	1

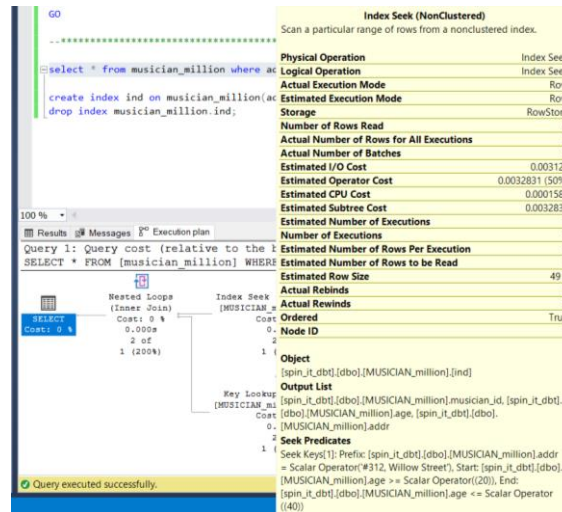
The bottom pane also shows the predicate: `[spin_it_dbo].[MUSICIAN_million].[age] >= CONVERT_IMPLICIT(int, @2, 0) AND [spin_it_dbo].[MUSICIAN_million].[age] <= CONVERT_IMPLICIT(int, @3, 0) AND [spin_it_dbo].[MUSICIAN_million].[addr] = @1`. The object is `[spin_it_dbo].[MUSICIAN_million].[pk_mim]`. The output list includes `[spin_it_dbo].[MUSICIAN_million].[id_no]`, `[spin_it_dbo].[MUSICIAN_million].[musician_id]`, `[spin_it_dbo].[MUSICIAN_million].[stage_name]`, `[spin_it_dbo].[MUSICIAN_million].[name]`, `[spin_it_dbo].[MUSICIAN_million].[minit]`, `[spin_it_dbo].[MUSICIAN_million].[name]`, `[spin_it_dbo].[MUSICIAN_million].[age]`, and `[spin_it_dbo].[MUSICIAN_million].[addr]`.

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

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

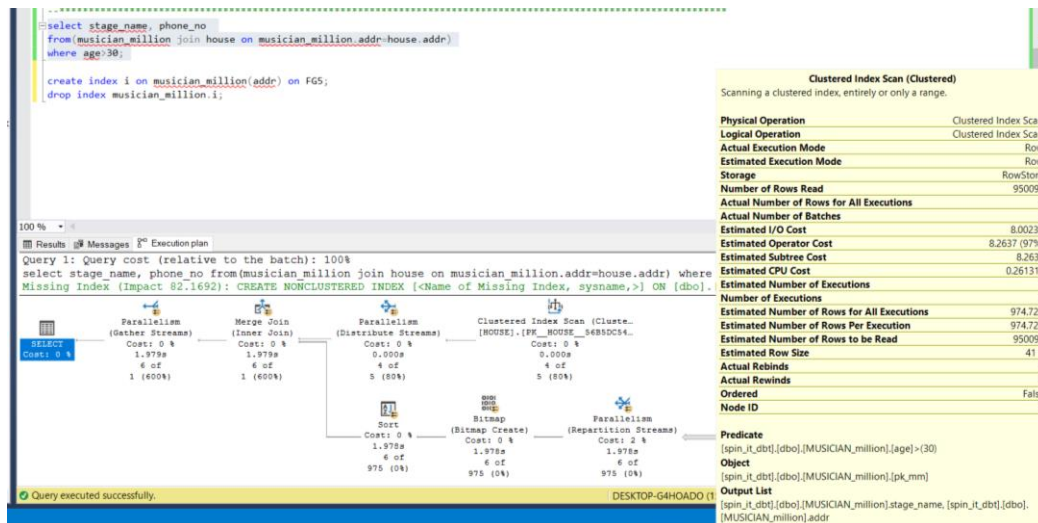
the cost gets significantly reduced (to 0.003)

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.

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

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

100 %

Results Messages Execution plan

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

select stage\_name, phone\_no from(musician\_million join house on musician

The execution plan diagram shows a query with the following structure:

- SELECT** (Cost: 0 %) - 10 of 10 (60%)
- Nested Loops (Inner Join)** (Cost: 0 %) - 10 of 10 (100%)
  - Key Lookup (Clustered)** (MUSICIAN\_million].[pk\_mm] Cost: 80 %) - 6 of 10 (60%)
  - Nested Loops (Inner Join)** (Cost: 0 %) - 10 of 10 (100%)
    - Index Scan (NonClustered)** (HOUSE].[ID\_HOUSE] Cost: 9) - 5 of 5 (100%)
    - Index Seek (NonClustered)** (MUSICIAN\_million].[MUSICIAN\_million].[i] Cost: 11) - 10 of 10 (100%)

Query executed successfully.

Index Seek (NonClustered)	
Scan a particular range of rows from a nonclustered index.	
Physical Operation	Index Seek
Logical Operation	Index Seek
Actual Execution Mode	Row
Estimated Execution Mode	Row
Storage	RowStore
Number of Rows Read	10
Actual Number of Rows for All Executions	10
Actual Number of Batches	0
Estimated Operator Cost	0.003921 (11%)
Estimated I/O Cost	0.003125
Estimated Subtree Cost	0.003921
Estimated CPU Cost	0.0001592
Estimated Number of Executions	5
Number of Executions	5
Estimated Number of Rows for All Executions	10
Estimated Number of Rows to be Read	2
Estimated Number of Rows Per Execution	2
Estimated Row Size	27 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	3
Object	[spin_it_dbt].[dbo].[MUSICIAN_million].[i]
Output List	[spin_it_dbt].[dbo].[MUSICIAN_million].musician_id
Seek Predicates	Seek Keys[1]: Prefix: [spin_it_dbt].[dbo].[MUSICIAN_million].addr = Scalar Operator([spin_it_dbt].[dbo].[HOUSE].[addr])