CTE, Temp Tables, Sub Queries, Table Types, and Data Types

ISM 6218

Due on October 1st

The Avengers Team

"We will avenge every problem on our way"

Aitemir Yeskenov (Team Lead)

Nagarjuna Kanneganti

Sai Suraj Argula

Vinay Kumar Reddy Baradi

Table of Contents

Business Process Supported

For this assignment, we decided to use Hospital database. The database has a variety of tables and relationships, which allow us to run experiments so as to analyze what and how we can efficiently run the queries. The main tables that we worked on are **bed**, **ward**, **nurse**, and **patient**. We experimented with common table expression, temporary tables, table variables, correlated and uncorrelated subqueries, use of cursor-based approach, and use of such operators as EXISTS, [NOT], IN, ANY. We also created our own table type and data type and compared execution plans.

Requirements Described

You must create a user story and derive specific requirements to run each of these series of experiments:

Experiment Series 1:

- CTE, Temp Table, Table Variable
- Compare Execution Plan

Experiment Series 2:

- Correlated Subquery, Uncorrelated Subquery, CTE
- Compare with Cursor
- Compare Execution Plan

Experiment Series 3:

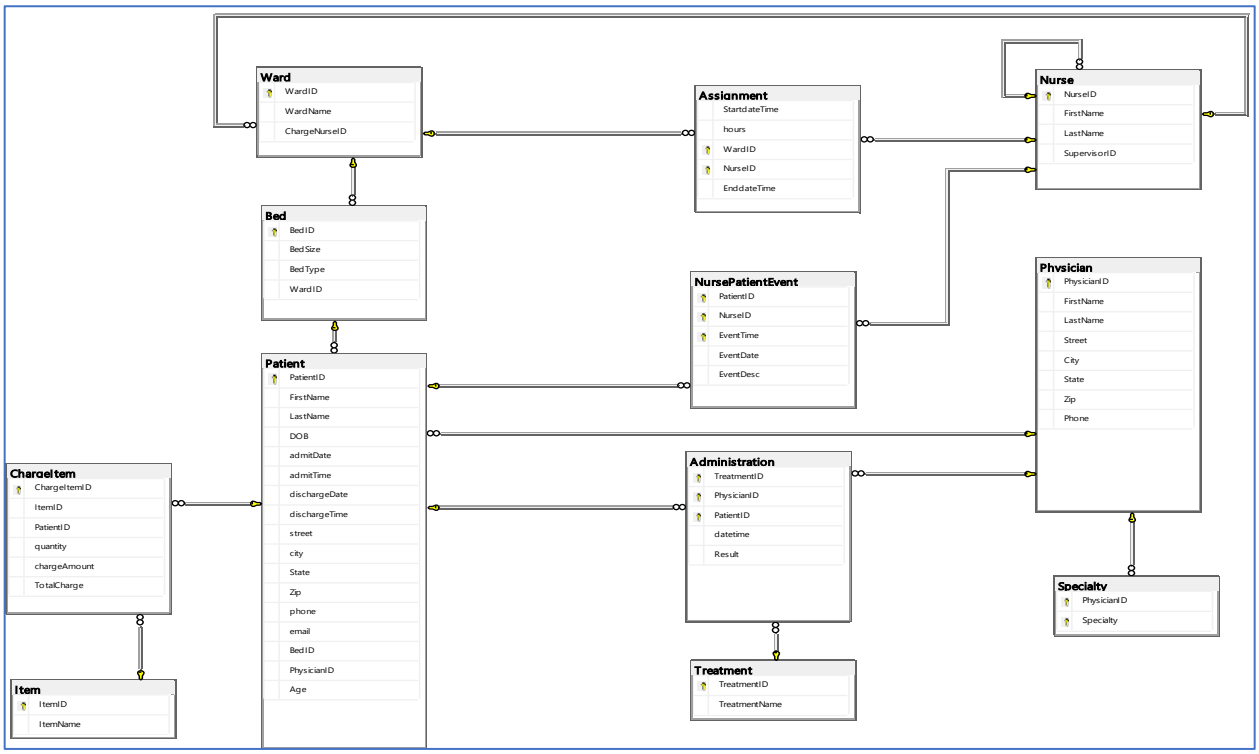
- Use EXITS, [NOT], IN, ANY
- Compare Results and Execution Plan

Experiment Series 4:

- Table Type and Data Type Exercise

Delivery should be formatted as your Database Lab Notebook.

Hospital DB diagram:



Experiment Series 1:

CTE example:

User story:

We want to take a look at the beds that have been assigned to wards. In other words, we need a list of beds that are being used at this point. Despite the fact that we can easily run a standalone join operation to get the results or create a stored procedure, we decided to use the feature of Common Table Expression. We want to use CTE since we are sure that we won't be interested in such type of query in the future - it is a one-time task that we have for now. CTE is a temporary solution, its object will not be saved after the session is finished. Here's the code we wrote:

```
with cteUsedBeds (BedId, BedSize, BedType, WardId)
as
(
Select BedId, BedSize, BedType, b.WardId
from Bed as b
inner join Ward as w
on b.WardId = w.WardID
)
select * from cteUsedBeds;
```

150 %

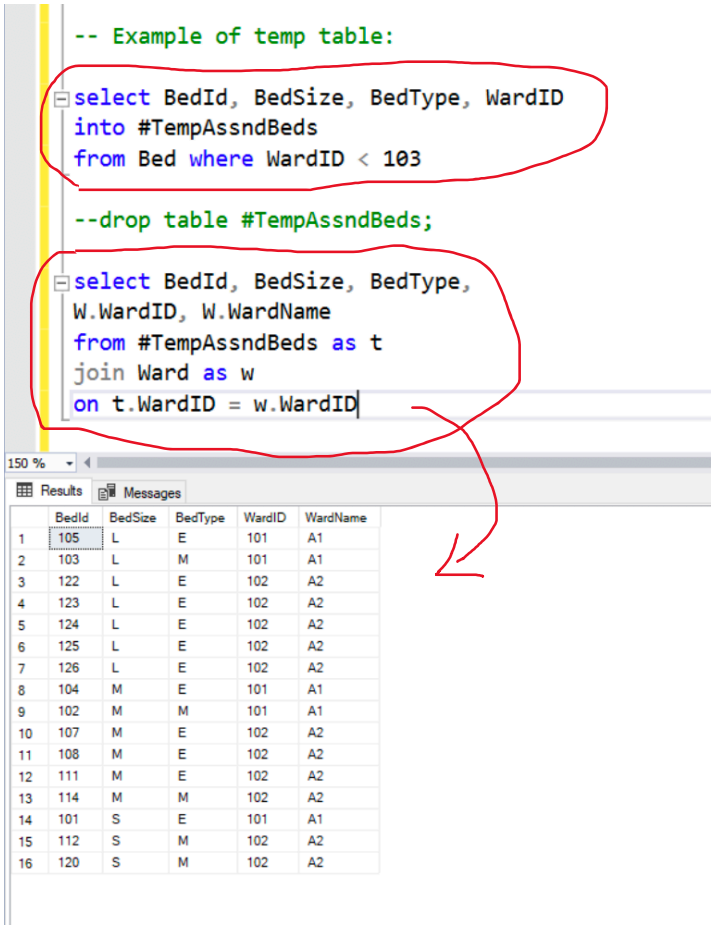
Results | Messages

	BedId	BedSize	BedType	WardId
1	105	L	E	101
2	103	L	M	101
3	110	L	E	103
4	122	L	E	102
5	123	L	E	102
6	124	L	E	102
7	125	L	E	102
8	126	L	E	102
9	104	M	E	101
10	102	M	M	101
11	107	M	E	102
12	108	M	E	102
13	111	M	E	102
14	114	M	M	102
15	101	S	E	101
16	112	S	M	102
17	118	S	M	105
18	120	S	M	102

Temp Table example:

User story:

While analyzing the data and looking at different tables, we decided that we want a quick snapshot of beds that are assigned to the first two wards (101 and 102) . Since the data is not going to be changed in any way, we decided to create a temporary table that would only have beds from the first two wards. We ran a join operation with the temp table as well as the ward table to see the identification of the beds, size and type, and also information about what ward a particular bed is assigned to (**WarID**, **WardName**):



```
-- Example of temp table:

select BedId, BedSize, BedType, WardID
into #TempAssndBeds
from Bed where WardID < 103

--drop table #TempAssndBeds;

select BedId, BedSize, BedType,
W.WardID, W.WardName
from #TempAssndBeds as t
join Ward as w
on t.WardID = w.WardID
```

150 %

Results | Messages

	BedId	BedSize	BedType	WardID	WardName
1	105	L	E	101	A1
2	103	L	M	101	A1
3	122	L	E	102	A2
4	123	L	E	102	A2
5	124	L	E	102	A2
6	125	L	E	102	A2
7	126	L	E	102	A2
8	104	M	E	101	A1
9	102	M	M	101	A1
10	107	M	E	102	A2
11	108	M	E	102	A2
12	111	M	E	102	A2
13	114	M	M	102	A2
14	101	S	E	101	A1
15	112	S	M	102	A2
16	120	S	M	102	A2

After the snapshot data was retrieved by the query we dropped the table since we would not use it any time in the future – **DROP TABLE #TempAssnBeds**

Table Variable example:

User story:

While analyzing the data about nurses and their assignments, we decided that we want to get a quick snapshot for a report on five nurses (101, 102, 103, 104, 105) that we have in the database. In particular, we are interested in the basic information about the nurses and assignment information of nurses and whether any nurse is missing the assignment. Since this is a one-time requirement for a specific result set, we decided to store the data temporarily without creation of any view or a persistent table. In order to do that, we decided to create a table variable with necessary columns from **Nurse** table as **@TableVNurseAssigned**. We set **NurseID**, **FirstName**, and **LastName** as the table variable's columns and inserted the values form the original Nurse table with the operator <=105 (to retrieve only first five nurses).

```
declare @TableVNurseAssigned table
(
    NurseID smallint,
    FirstName varchar(10),
    LastName varchar(10)
)
insert into @TableVNurseAssigned
select NurseID, FirstName, LastName from Nurse
where NurseID <=105;
```

After that we wrote a left join query and joined the **Assignment** table with the following columns – **WardID, hours, StartdateTime, EnddateTime**:

```
select t.NurseID, t.FirstName,
t.LastName, a.hours,
a.WardID, a.StartdateTime, a.EnddateTime
from @TableVNurseAssigned as t left join Assignment as a on
t.NurseID = a.NurseID;
```

Note that it is important that we execute everything at once since this is how the Table Variables work:

```
declare @TableVNurseAssigned table
(
    NurseID smallint,
    FirstName varchar(10),
    LastName varchar(10)
)
insert into @TableVNurseAssigned
select NurseID, FirstName, LastName from Nurse
where NurseID <=105;

select t.NurseID, t.FirstName,
t.LastName, a.hours,
a.WardID, a.StartdateTime, a.EnddateTime
from @TableVNurseAssigned as t left join Assignment as a on
t.NurseID = a.NurseID;
```

150 %

Results | Messages

	NurseID	FirstName	LastName	hours	WardID	StartdateTime	EnddateTime
1	104	Betty	Ford	NULL	NULL	NULL	NULL
2	102	Bill	Smith	5	101	2017-03-02 00:00:00.000	2017-03-02 22:00:00.000
3	102	Bill	Smith	6	102	2017-02-02 00:00:00.000	NULL
4	102	Bill	Smith	4	103	2018-10-22 00:00:00.000	NULL
5	103	Eric	Alexander	3	102	2018-10-12 00:00:00.000	NULL
6	105	John	Davis	3	101	2018-10-22 00:00:00.000	NULL
7	105	John	Davis	3	102	2017-05-03 00:00:00.000	NULL
8	105	John	Davis	8	103	2017-03-04 00:00:00.000	NULL
9	101	Sally	Jones	8	101	2017-01-01 00:00:00.000	2017-01-01 15:00:00.000

The received result set perfectly shows us information about the nurses and if there is any nurse that hasn't been assigned yet. According to the result set, nurse with id 104 does not have any assignment. Also, the table won't be stored anywhere on the DB after the session is over, which is exactly what we wanted.

CTE VS TEMP TABLE VS TABLE VARIABLE – EXECUTION COMPARISON:

In order to adequately compare execution plan between three approaches we decided to use the same user story. The chosen user story is the one we used for the first CTE example –

"We want to take a look at the beds that have been assigned to wards. In other words, we need a list of beds that are being used at this point."

We made sure that all the approaches give us identical result set:

CTE:

```
-- CTE Example:

with cteUsedBeds (BedId, BedSize, BedType, WardId)
as
(
Select BedId, BedSize, BedType, b.WardId
from Bed as b
inner join Ward as w
on b.WardId = w.WardID
)
select * from cteUsedBeds;
```

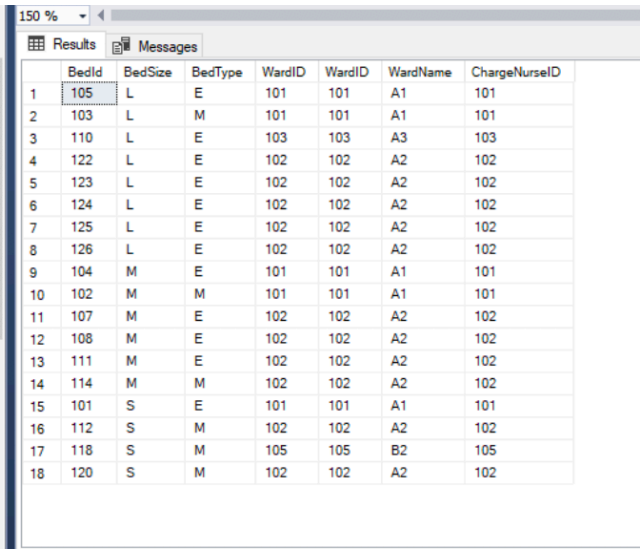
Temp Table:

```
-- Temp Table Example:

select BedId, BedSize, BedType, WardID
into #TempUsedBeds
from Bed

select * from #TempUsedBeds as t
inner join Ward as w
on t.WardID = w.WardID

drop table #TempUsedBeds;
```

Table Variable:

```
declare @tableVUsedBeds table
(
    BedId smallint,
    BedSize char(2),
    BedType char(1),
    WardId smallint
)
insert into @tableVUsedBeds
select BedId, BedSize, BedType, WardID from Bed

select t.bedId, t.BedSize, t.BedType, w.WardID
from @tableVUsedBeds as t inner join Ward as w on
t.WardID = w.WardID;
```

This is the result that all the three approaches give us:



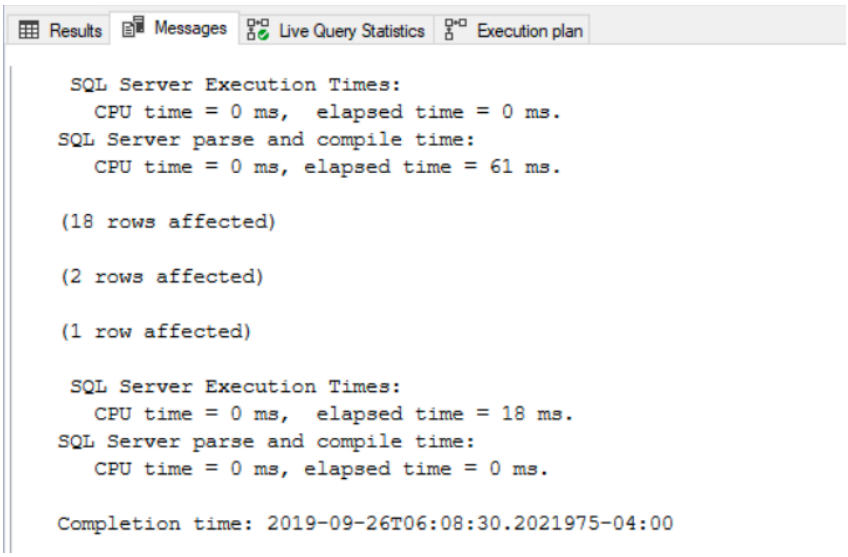
150 %

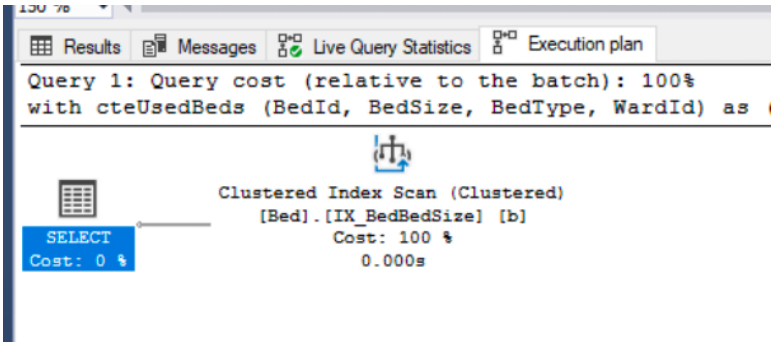
Results | Messages

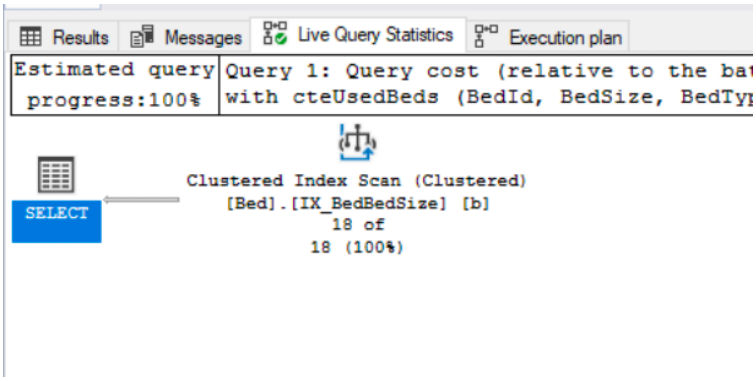
	BedId	BedSize	BedType	WardID	WardID	WardName	ChargeNurseID
1	105	L	E	101	101	A1	101
2	103	L	M	101	101	A1	101
3	110	L	E	103	103	A3	103
4	122	L	E	102	102	A2	102
5	123	L	E	102	102	A2	102
6	124	L	E	102	102	A2	102
7	125	L	E	102	102	A2	102
8	126	L	E	102	102	A2	102
9	104	M	E	101	101	A1	101
10	102	M	M	101	101	A1	101
11	107	M	E	102	102	A2	102
12	108	M	E	102	102	A2	102
13	111	M	E	102	102	A2	102
14	114	M	M	102	102	A2	102
15	101	S	E	101	101	A1	101
16	112	S	M	102	102	A2	102
17	118	S	M	105	105	B2	105
18	120	S	M	102	102	A2	102

Execution comparison:

CTE:


Results
Messages
Live Query Statistics
Execution plan
SQL Server Execution Times:
CPU time = 0 ms, elapsed time = 0 ms.
SQL Server parse and compile time:
CPU time = 0 ms, elapsed time = 61 ms.
(18 rows affected)
(2 rows affected)
(1 row affected)
SQL Server Execution Times:
CPU time = 0 ms, elapsed time = 18 ms.
SQL Server parse and compile time:
CPU time = 0 ms, elapsed time = 0 ms.
Completion time: 2019-09-26T06:08:30.2021975-04:00

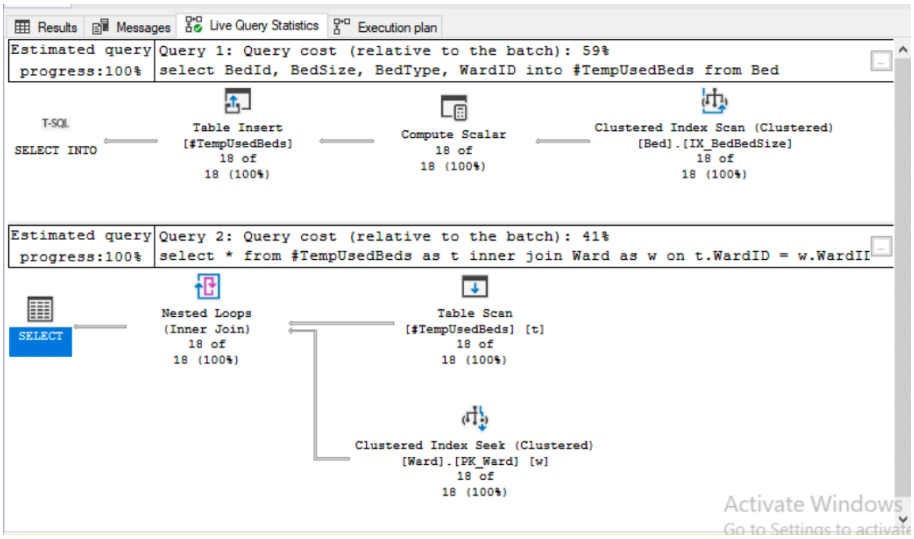

Results
Messages
Live Query Statistics
Execution plan
Query 1: Query cost (relative to the batch): 100%
with cteUsedBeds (BedId, BedSize, BedType, WardId) as
Clustered Index Scan (Clustered)
[Bed].[IX_BedBedSize] [b]
Cost: 100 %
0.000s
SELECT
Cost: 0 %


Results
Messages
Live Query Statistics
Execution plan
Estimated query progress:100%
Query 1: Query cost (relative to the bat
with cteUsedBeds (BedId, BedSize, BedTyp
Clustered Index Scan (Clustered)
[Bed].[IX_BedBedSize] [b]
18 of
18 (100%)
SELECT

Temp Table:

```
 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 1 ms.

(18 rows affected)

(4 rows affected)

(1 row affected)

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 64 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 2 ms.

(18 rows affected)

(4 rows affected)
```

```
 (1 row affected)

  SQL Server Execution Times:
    CPU time = 0 ms,  elapsed time = 268 ms.
SQL Server parse and compile time:
    CPU time = 0 ms, elapsed time = 0 ms.

Completion time: 2019-09-26T06:13:01.2904789-04:00
```



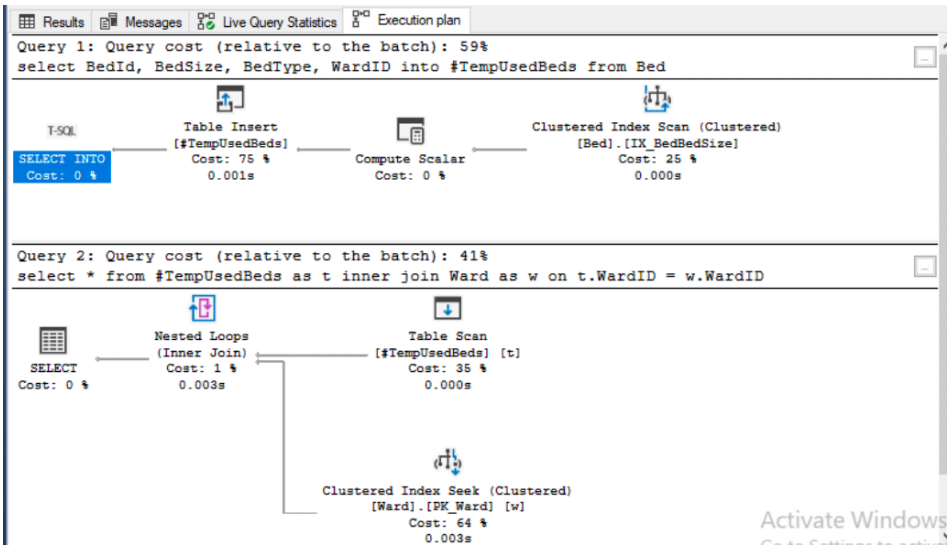
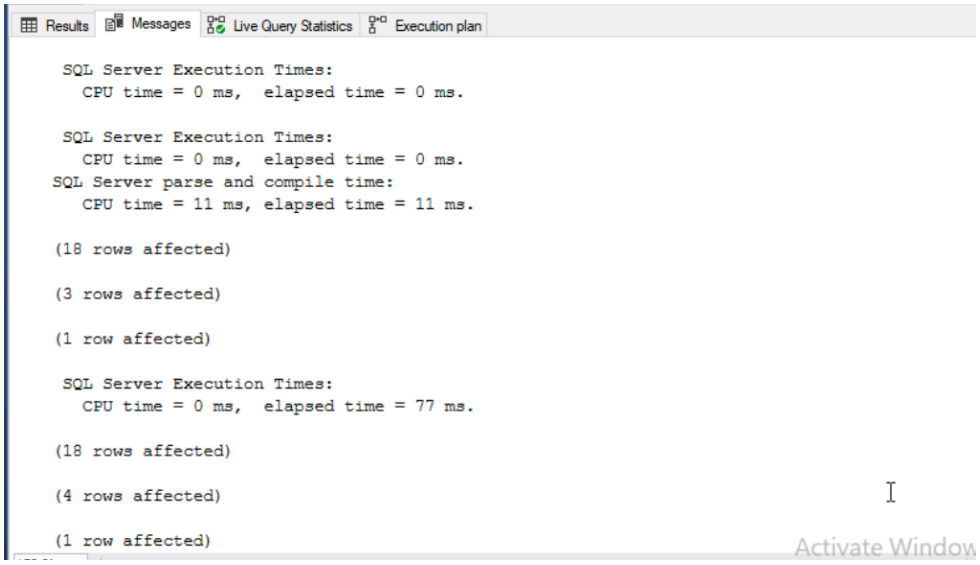
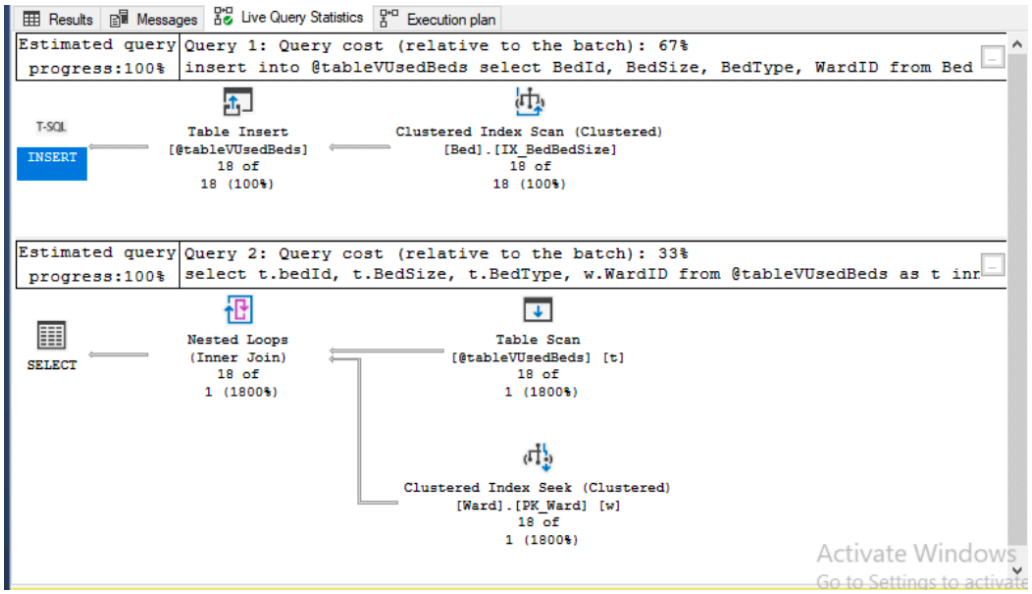
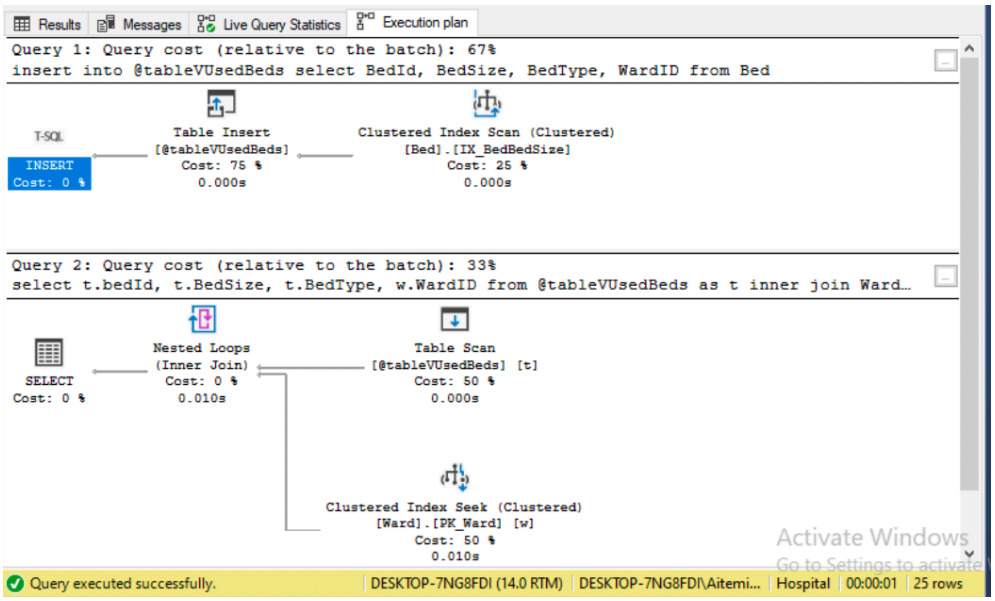


Table Variable:




```
 SQL Server Execution Times:
   CPU time = 15 ms,  elapsed time = 232 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

Completion time: 2019-09-26T06:15:26.7926262-04:00
```





Execution comparison results:

CTE – elapsed time 61ms, COST 0.000s

Temp Table – elapsed time 1ms, COST table insert - 0.001s, nested loops - 0.003s

Table Variable – elapsed time 11 ms, COST table insert - 0.00s, nested loops - 0.010s

As you can see, according to our statistics, CTE turned out to be the slowest in terms of performance, whereas Temp Table showed the best result. – elapsed time of 1ms and COST of table insert of 0.001s and nested loops of 0.003s.

Experiment Series 2 and 3:

The experiments 2 and 3 are about subqueries. Since our user stories include operators such as EXISTS/ ANY / IN and are correlated/uncorrelated, we decided to satisfy requirements of the 2nd and 3rd experiments in one (permission received from Professor on Slack).

Correlated and Uncorrelated Subquery:

Correlated Subquery user story (EXISTS operator):

One of the recent changes in the database was the assignment of a new supervisor from the list of working nurses. We want to identify that nurse who became a supervisor. In order to do that, we decided to run a correlated subquery with EXISTS operator:

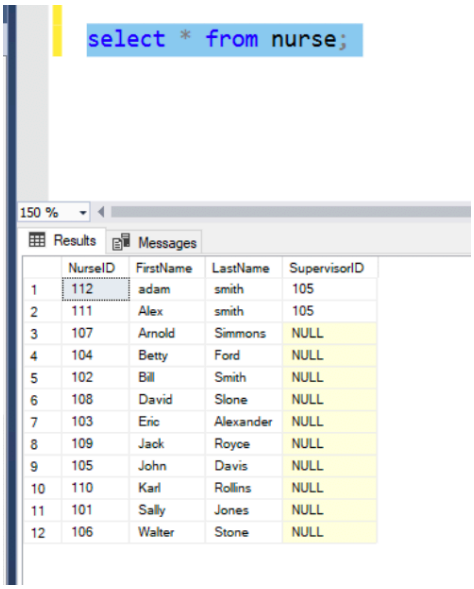
```
-- EXISTS
-- identify the supervisor
select NurseID, FirstName, LastName from Nurse a
where exists (
select nurseID from Nurse b
where b.SupervisorID = a.NurseID
)
```

50 %

Results | Messages

	NurseID	FirstName	LastName
1	105	John	Davis

We used **nurse** table twice (used aliases: a and b) and also used EXISTS operator to check if the data with the condition that we specified exists. This query is correlated since we check if the **SupervisorID** equals the **NurseID**. As you can see we retrieved the nurse that is also a supervisor. We can also check it by running select statement to get the entire list of the nurses:



	NurseID	FirstName	LastName	SupervisorID
1	112	adam	smith	105
2	111	Alex	smith	105
3	107	Arnold	Simmons	NULL
4	104	Betty	Ford	NULL
5	102	Bill	Smith	NULL
6	108	David	Slone	NULL
7	103	Eric	Alexander	NULL
8	109	Jack	Royce	NULL
9	105	John	Davis	NULL
10	110	Karl	Rollins	NULL
11	101	Sally	Jones	NULL
12	106	Walter	Stone	NULL

After getting the result set, we confirmed that 105 is indeed the supervisor nurse.

Correlated Subquery user story (NOT EXISTS operator):

We decided that we wanted values opposite to our initial idea – nurses that are regular nurses without supervisor level. To do that we simply put **NOT EXISTS** instead of **EXISTS** and got the result set as:

```
select NurseID, FirstName, LastName from Nurse a
where not exists (
select nurseID from Nurse b
where b.SupervisorID = a.NurseID
)

select * from nurse;
```

150 %

Results | Messages

	NurseID	FirstName	LastName
1	112	adam	smith
2	111	Alex	smith
3	107	Arnold	Simmons
4	104	Betty	Ford
5	102	Bill	Smith
6	108	David	Slone
7	103	Eric	Alexander
8	109	Jack	Royce
9	110	Karl	Rollins
10	101	Sally	Jones
11	106	Walter	Stone

As you can see we retrieved everything except for the nurse with the supervisor level.

Uncorrelated Subquery user story (IN operator):

We want to identify nurses that work less than 6 hours. The data about the hours they work is stored in the **Assignment** table. In order to get this data, we want to run a subquery that includes our **nurse** table and **assignment** table. The condition of the inner query is to find the nurses that work less than 6 hours. Note that the clause of outer query after "where" shows the use of the inner query – the inner query retrieves IDs that should match with the IDs of the outer query. We also use **IN** operator so as to match the exact IDs:

```
-- IN
-- identify nurses that work less than or exactly 6 hours
select NurseId, FirstName, LastName
from Nurse Where NurseId In
(
select NurseId from Assignment as a Where a.hours <= 6
)
```

150 %

Results | Messages

	NurseId	FirstName	LastName
1	102	Bill	Smith
2	103	Eric	Alexander
3	105	John	Davis

As you can see we got the result set with the nurses who work exactly 6 or less than 6 hours. Again, this is **not a correlated query**. We use the operator **IN** to find the matching values of two tables without connecting the PK and FK.

Subquery with ANY operator user story:

Alternatively, we can use "= ANY" operator to get the same output:

```
select NurseId, FirstName, LastName
from Nurse Where NurseId = ANY
(
select NurseId from Assignment as a Where a.hours <= 6
)
```

150 %

Results | Messages

	NurseId	FirstName	LastName
1	102	Bill	Smith
2	103	Eric	Alexander
3	105	John	Davis

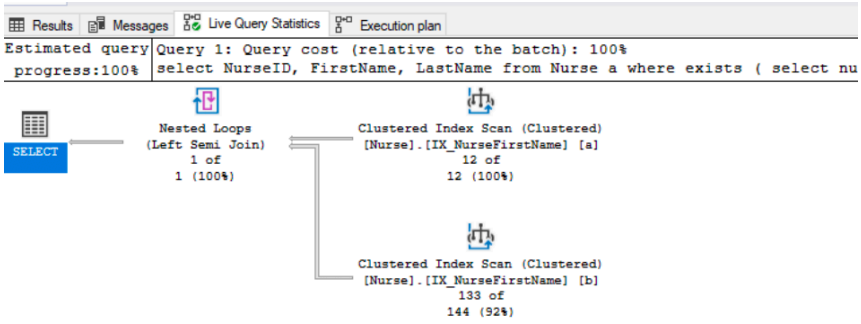
ANY sometimes can be used for the same purpose as **IN**. However, this does not work in all the cases. For instance, if we used **"<> ANY**", the output would be different.

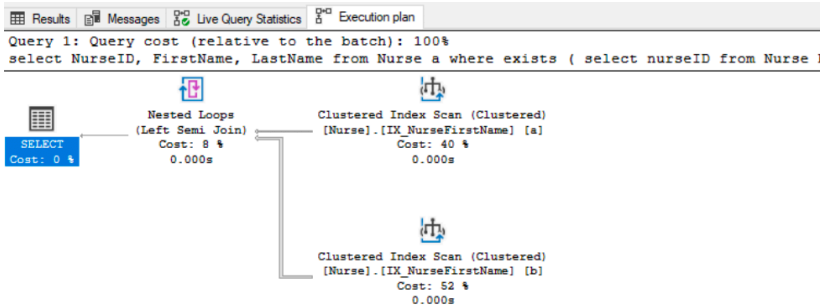
EXECUTION PLAN COMPARISON:

Correlated subquery with **EXISTS** operator:

```
-- EXISTS
-- identify the supervisor
select NurseID, FirstName, LastName from Nurse a
where exists (
select nurseID from Nurse b
where b.SupervisorID = a.NurseID
)
```

```
 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 4 ms.

(1 row affected)

(4 rows affected)

(1 row affected)

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 104 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

Completion time: 2019-09-28T18:06:32.6804870-04:00
```

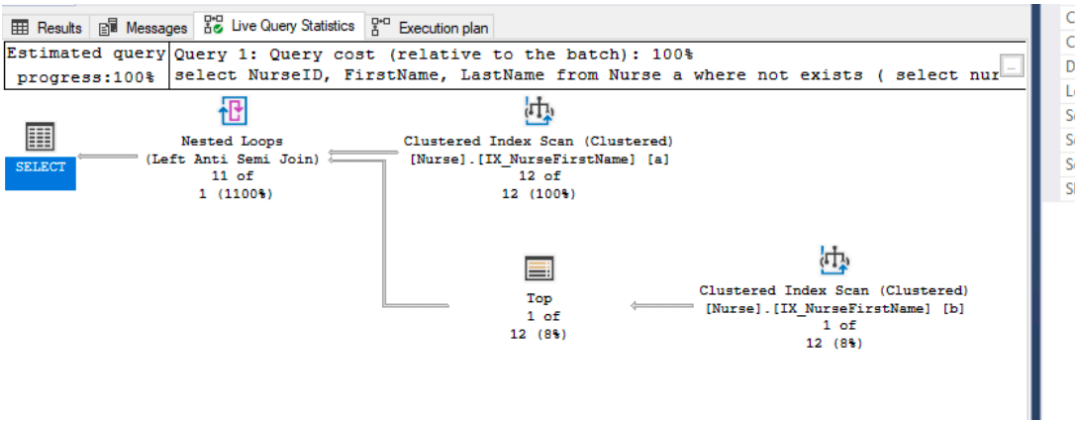


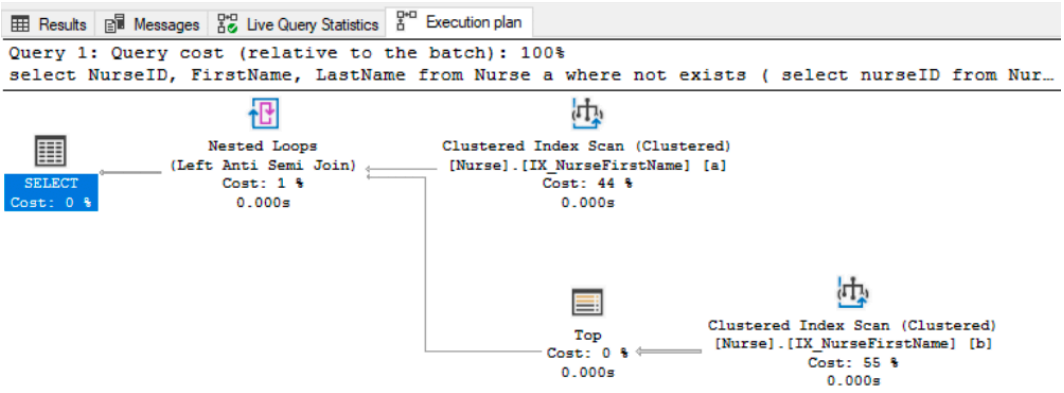


Correlated subquery with **NOT EXISTS** operator:

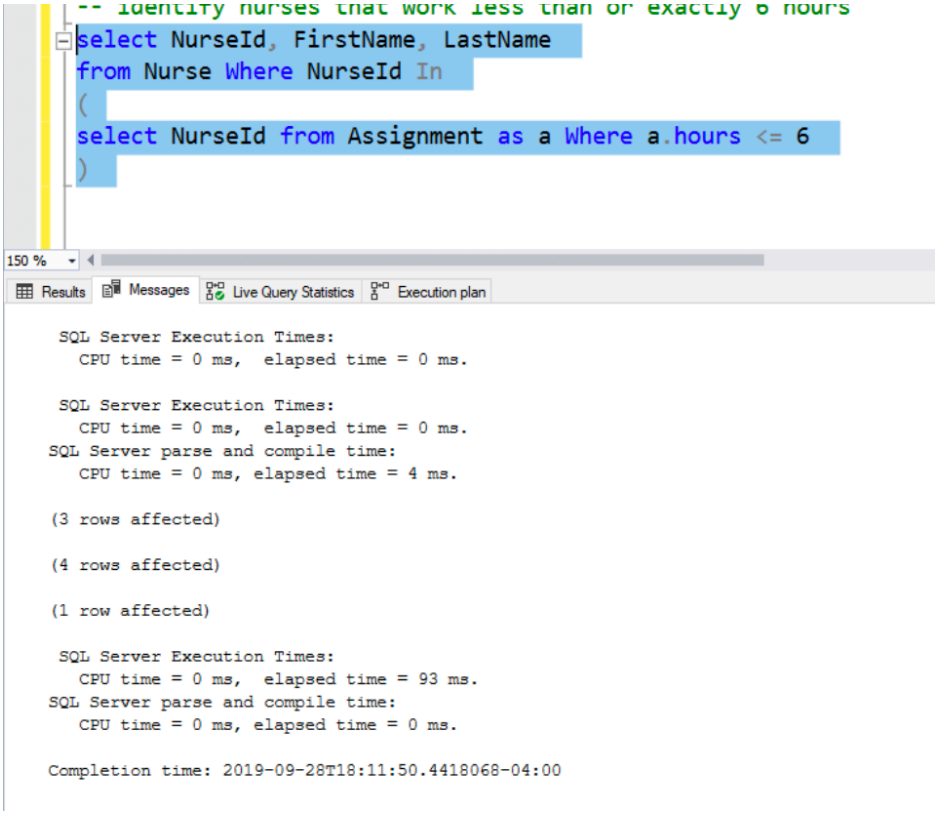
```
-- identify the supervisor
select NurseID, FirstName, LastName from Nurse a
where not exists (
select nurseID from Nurse b
where b.SupervisorID = a.NurseID
)
```

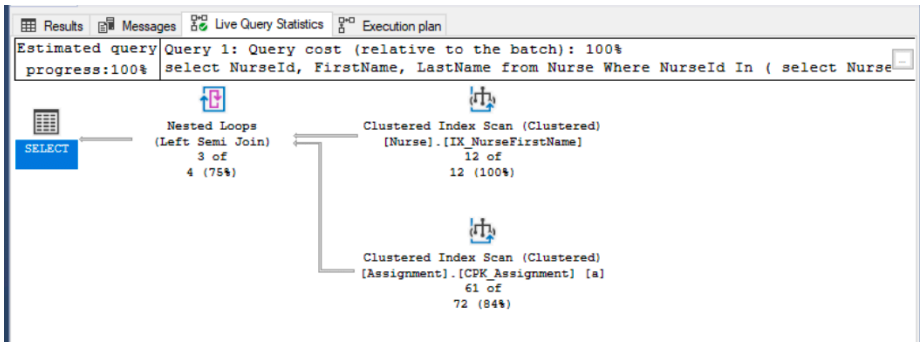
```
Results  Messages  Live Query Statistics  Execution plan

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 2 ms.

(11 rows affected)

(5 rows affected)

(1 row affected)

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 92 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

Completion time: 2019-09-28T18:09:40.5233167-04:00
```

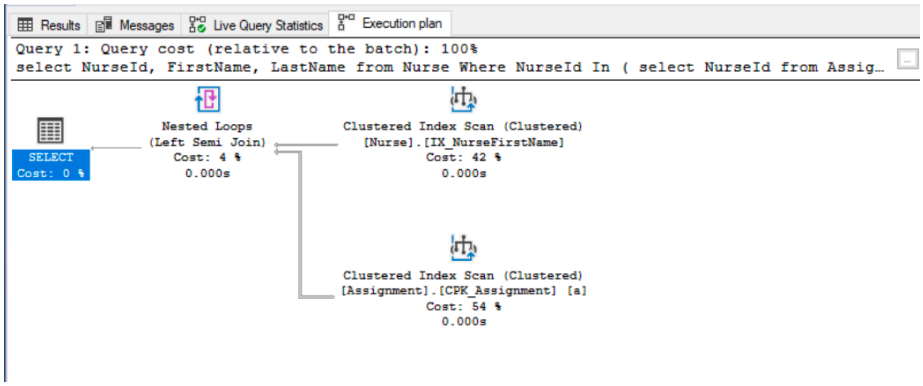




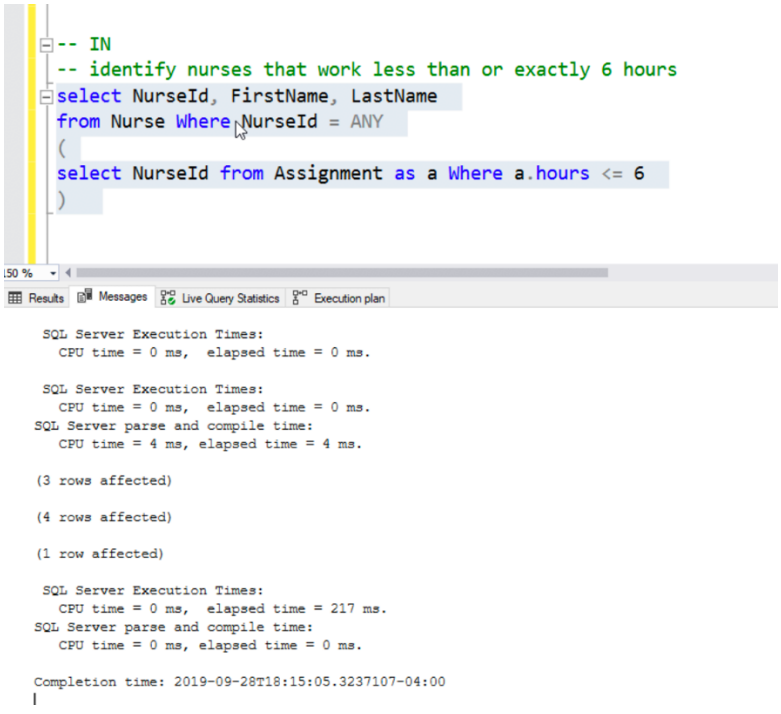
Uncorrelated subquery with **IN** operator:







Uncorrelated subquery with **ANY** operator:

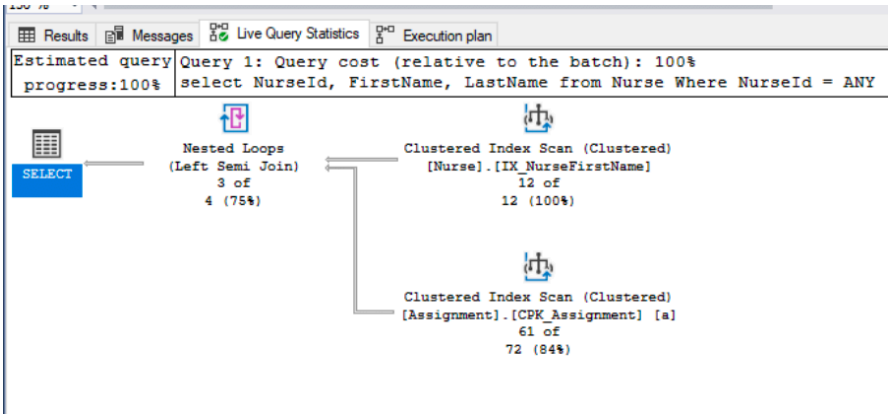


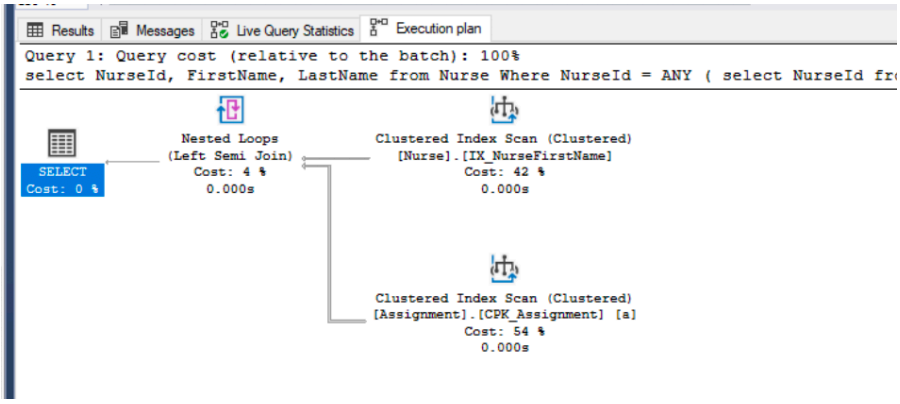
```
-- IN
-- identify nurses that work less than or exactly 6 hours
select NurseId, FirstName, LastName
from Nurse Where NurseId = ANY
(
select NurseId from Assignment as a Where a.hours <= 6
)
```

150 %

Results | Messages | Live Query Statistics | Execution plan

```
 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
SQL Server parse and compile time:
   CPU time = 4 ms, elapsed time = 4 ms.

(3 rows affected)

(4 rows affected)

(1 row affected)

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 217 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

Completion time: 2019-09-28T18:15:05.3237107-04:00
```





CORRELATED SUBQUERY WITH EXISTS – elapsed time 4ms, COST 0.000s

CORRELATED SUBQUERY WITH NOT EXISTS – elapsed time 2ms, COST 0.000s

UNCORRELATED SUBQUERY WITH IN – elapsed time 4ms, COST 0.000s

UNCORRELATED SUBQUERY WITH ANY – elapsed time 4ms, COST 0.000s

As you can see, the correlated subquery with **NOT EXISTS** showed the best indicator – 2ms with the COST of 0.000s. The remaining three competitors correlated subquery with exists and two uncorrelated subqueries all show 4ms as the elapsed time in performance.

Cursor User Story:

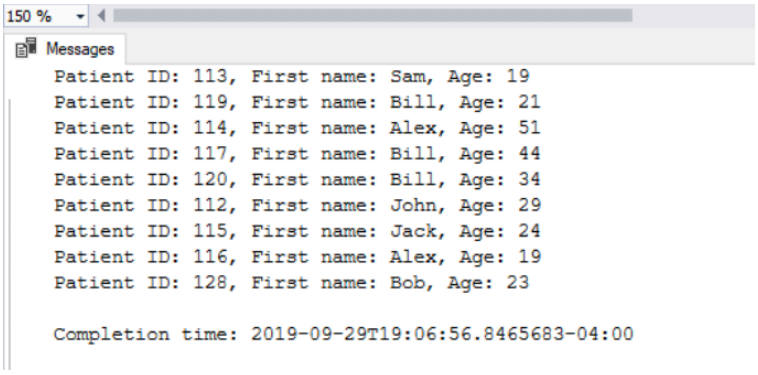
Our team ran into reporting requirements where it is necessary to implement the cursor so as to generate a report with patient id, first name, and age of all the patients that we have in the database. We know that there are some referential integrity issues in our database and that is why we want to use the cursor-based approach instead of regular select query. Here is the syntax:

```
DECLARE
    @FirstName varchar(10),
    @patientId smallint,
    @age int;

DECLARE cursor_patient CURSOR
FOR SELECT
        FirstName,
        patientId,
        age
    FROM
        patient;

OPEN cursor_patient;

FETCH NEXT FROM cursor_patient INTO
    @FirstName,
    @patientId,
    @age;
```

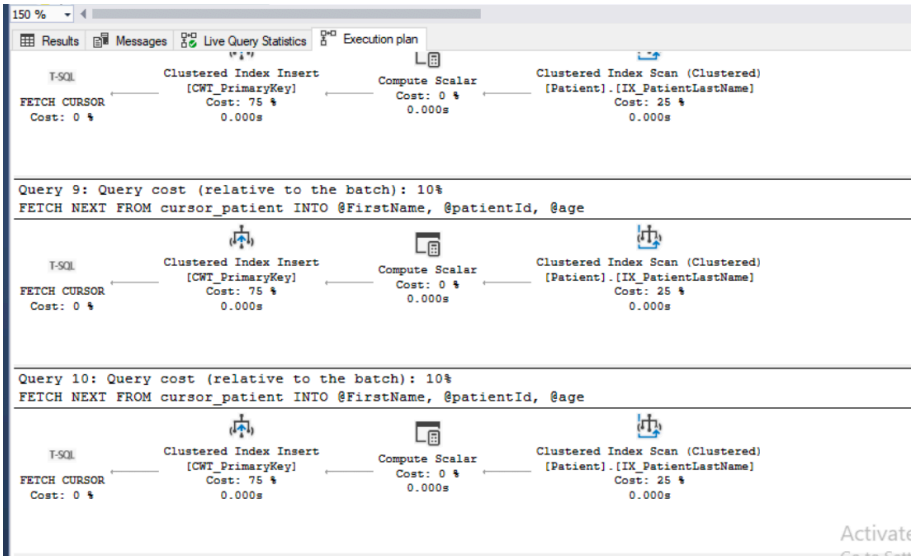
```
WHILE @@FETCH_STATUS = 0
    BEGIN
        PRINT 'Patient ID: ' + CAST (@patientId as varchar) + ', First name: ' +
        --PRINT @age;
        FETCH NEXT FROM cursor_patient INTO
        @FirstName,
        @patientId,
        @age
    END;

CLOSE cursor_patient;

DEALLOCATE cursor_patient;
```

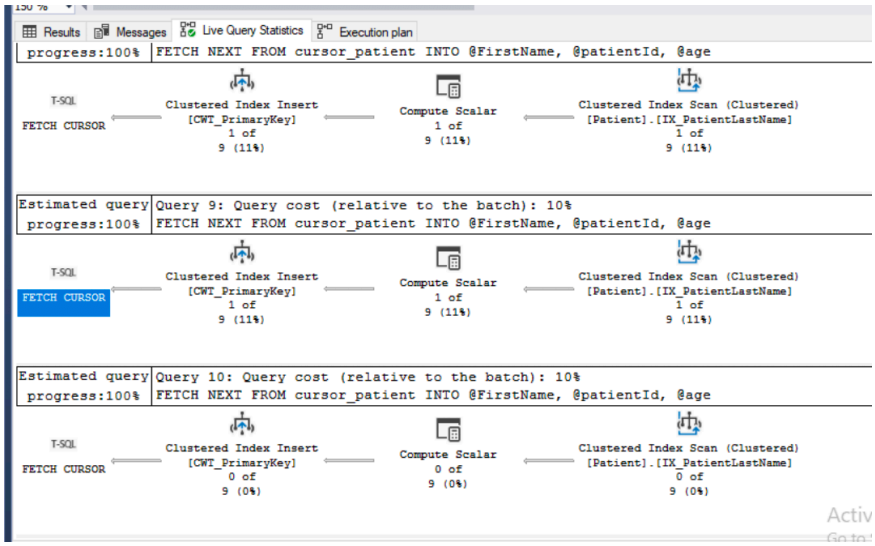
Note that we declared the parameters based on the values of columns we want to fetch (**SELECT FirstName, patientId, Age FROM patient**). It is worth to mention that we have to open and close the cursor as well as deallocate after we achieved what we originally wanted. We also decided to print messages with meaningful string comments such as “Patient ID: ” for each record.

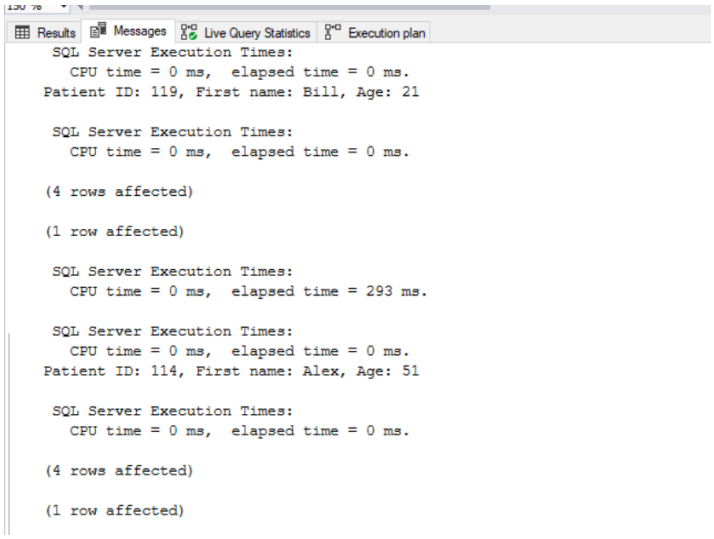
Output received:



Execution statistics:







It would be difficult to compare the cursor approach execution with rest of the queries since every record has its own execution time, live query statistics, and the execution plan. However, on the screens above, you can see some samples from the performance of the cursor.

Experiment 4:

User Defined Table User Story:

We decided to create a table type to create tables from that type. The idea is that there might be different positions across the hospital (physician, nurse, patient, etc.) and we want to create a

template for the creation of the new tables. We noticed that every position typically includes such columns as id, first and last names, city, state, zip, and phone number.

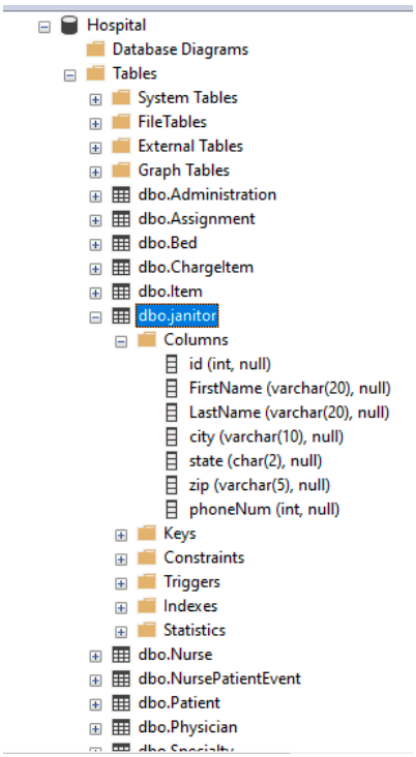
We wanted to create a janitor table. To do that we created positionTableType table type:

```
CREATE TYPE positionTableType AS TABLE
(id int,
FirstName varchar(20),
LastName varchar(20),
city varchar(10) null,
state char(2) null,
zip varchar(5),
phoneNum int
)
```

We created **Janitor** table based on this type:

```
-- create table from a type:
declare @table as positionTableType
select * into janitor
from @table
```

We now have the **Janitor** table:



User Defined Data Type User Story:

We noticed that phone numbers are quite a difficult data type to deal with. In order to make data entry process for this attribute simpler, we decided to create a custom data type. Here is the syntax for the creation of **phoneNum** data type:

```
-- user-defined data type

sp_addtype 'phoneNum', 'varchar(10)';
```

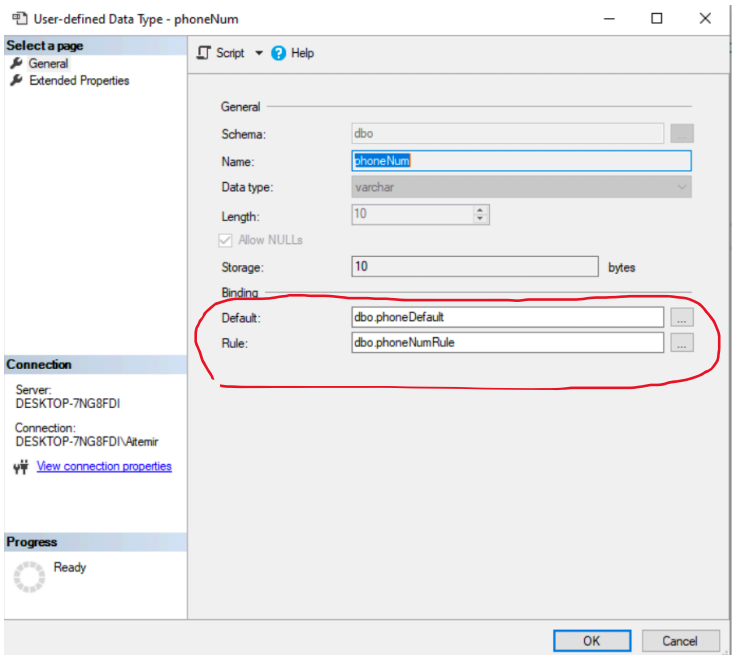
We created one rule and one default to make sure we follow the company's standards. The standard is to have no more than 10 characters. Since in certain cases we might have shorter phone numbers, we allow to have less than 10:

```
uery23.sql - D...G8FDI\Aitemir (60))   SQLQuery22.sql - D...G8FDI\Aitemir (52))*
USE [Hospital]
GO

/****** Object:  Rule [phoneNumRule]    Scr
CREATE RULE [dbo].[phoneNumRule]
AS
(LEN (@phoneNum) <=10)
GO
```

Another standard is to have "**UnknownNum**" in the db in case the value is not provided:

```
USE [Hospital]
GO

/****** Object:  Default [phoneDefault]    Script Date: 9/29/2019 9
CREATE DEFAULT [dbo].[phoneDefault]
AS
'UnknownNum'
GO
```

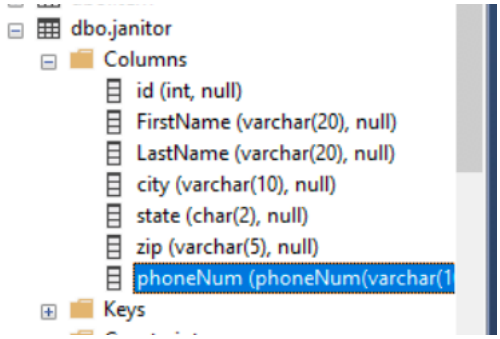
Both default and rule are bound to our **phoneNum** data type:



We decided that we definitely need to add this datatype to our newly created janitor table. To do that, we had to drop the original column and add a new one with the custom datatype:

```
alter table janitor
drop phoneNum;

alter table janitor
add phoneNum phoneNum
```

Finally, we have a column with the user defined data type in the table created from the user defined table type:



To reassure that everything works fine, we added a record without any value – the default check got triggered and we got this output: