Advanced Join Operations and Hints

ISM 6218

Due on September 17

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 want to use ACME 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 Customer, OrderLine, and Price. We experimented with equi and non-equi joins, hints joins, composite joins, cross, and remote joins.

Requirements Described

part 1 - Join operations:

- Run an experiment comparing an equi-join and a non-equi join.
- Support a user story demonstrating the difference.
- Run an experiment comparing a join and a composite join.
- Support a user story demonstrating the difference.
- Create and execute a user story demonstrating a cross join.
- Run a full join to find missing match values.

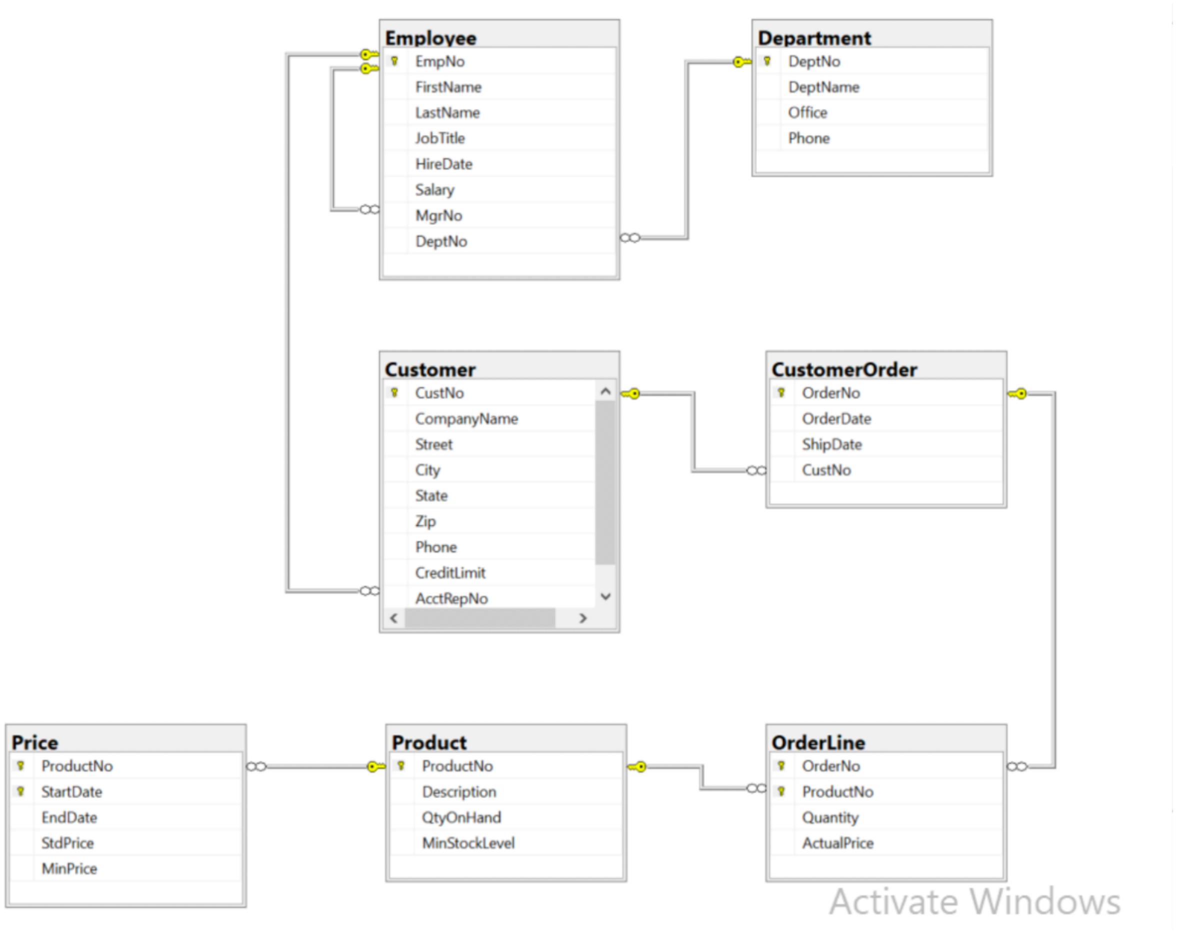
part 2 - Hints

- Run a series of experiments with an inner and an outer join using loop, hash, merge.
- Run an experiment with a join query where one table is on the azure server. Run the join using remote and not using remote.

part 3 - reporting

- Compare execution plans and write a descriptive narrative explaining the performance differences.

ACME DB diagram:



As for the previous ACME DB experiments, this time we are using dummy data that we inserted through **generatedata** tool.

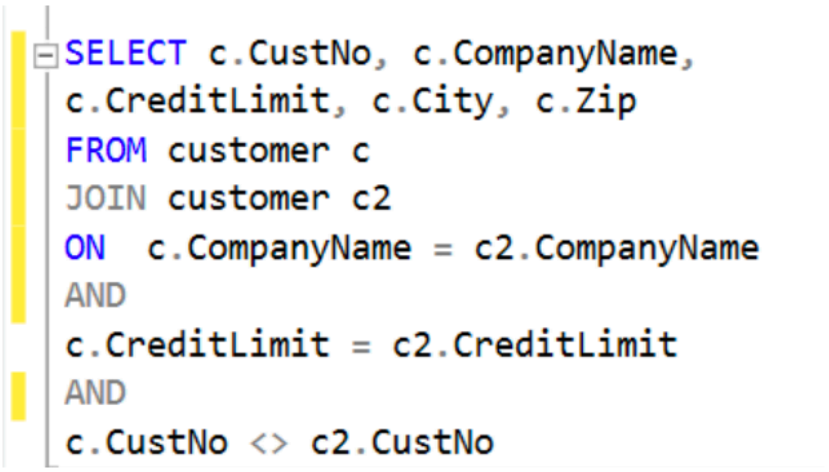
Part I

1) Equi vs non-equi join

User story:

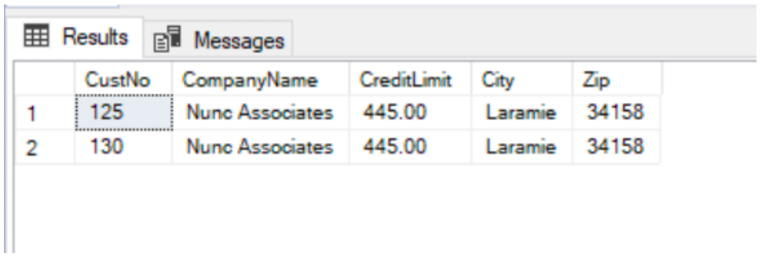
We want to make sure that after a tremendous data insertion to the customer table we did not create duplicate records by accident. In other words, we want to identify the duplicated records. To do that, we decided to run a JOIN operation on customer table that has conditional operators “>” and “<”. We know that combination of company name and credit limit is unique. Therefore, if there is a different **Customer Number** on more than one record with the same name and credit limit, it is most likely to be a duplicate:

Non-equi join:



```
SELECT c.CustNo, c.CompanyName,
c.CreditLimit, c.City, c.Zip
FROM customer c
JOIN customer c2
ON  c.CompanyName = c2.CompanyName
AND
c.CreditLimit = c2.CreditLimit
AND
c.CustNo <> c2.CustNo
```

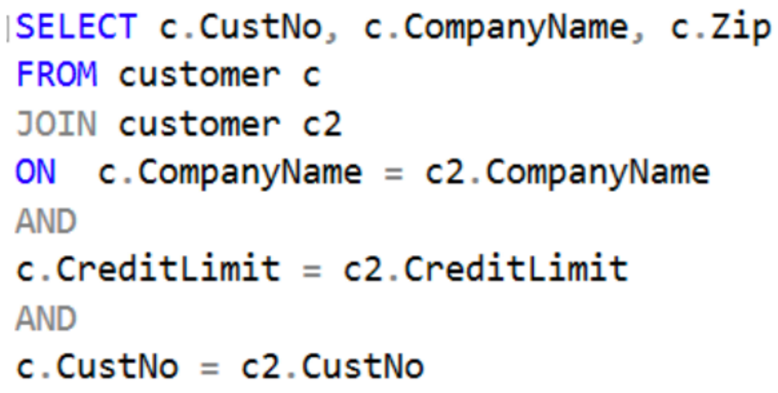
Output received:



Results Messages

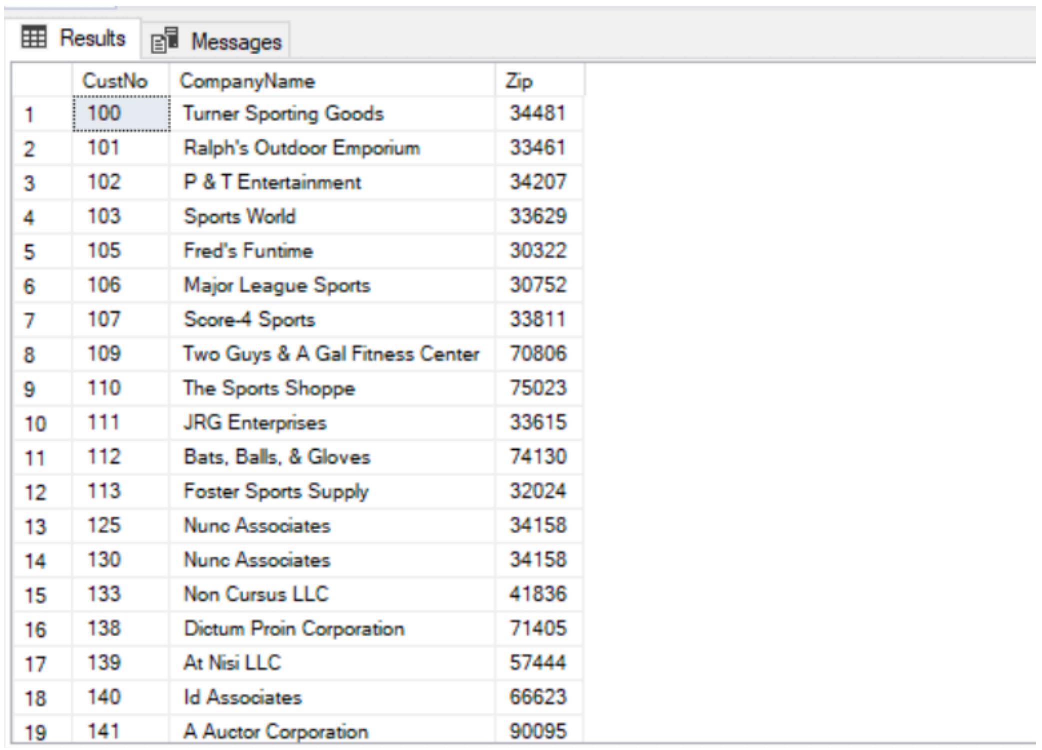
	CustNo	CompanyName	CreditLimit	City	Zip
1	125	Nunc Associates	445.00	Laramie	34158
2	130	Nunc Associates	445.00	Laramie	34158

Note that we also used composite join to make sure that CompanyName and CreditLimit match. For the sake of experiment we also ran equi join and just replaced "< >" with "=". That simply gave us all the records of the table since we run the join the table to itself:



```sql
SELECT c.CustNo, c.CompanyName, c.Zip
FROM customer c
JOIN customer c2
ON  c.CompanyName = c2.CompanyName
AND
c.CreditLimit = c2.CreditLimit
AND
c.CustNo = c2.CustNo
```

Output received



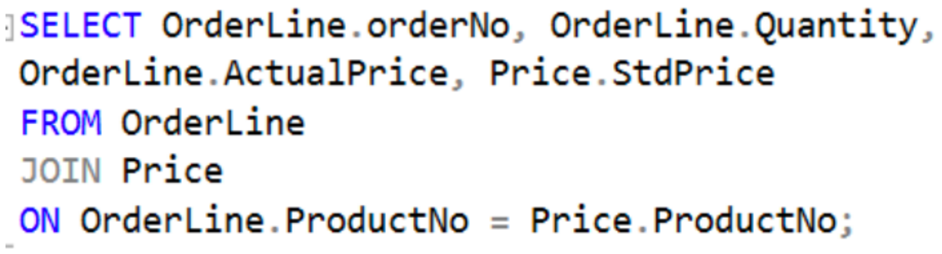
	CustNo	CompanyName	Zip
1	100	Turner Sporting Goods	34481
2	101	Ralph's Outdoor Emporium	33461
3	102	P & T Entertainment	34207
4	103	Sports World	33629
5	105	Fred's Funtime	30322
6	106	Major League Sports	30752
7	107	Score-4 Sports	33811
8	109	Two Guys & A Gal Fitness Center	70806
9	110	The Sports Shoppe	75023
10	111	JRG Enterprises	33615
11	112	Bats, Balls, & Gloves	74130
12	113	Foster Sports Supply	32024
13	125	Nunc Associates	34158
14	130	Nunc Associates	34158
15	133	Non Cursus LLC	41836
16	138	Dictum Proin Corporation	71405
17	139	At Nisi LLC	57444
18	140	Id Associates	66623
19	141	A Auctor Corporation	90095

1015 records retrieved as expected (this is the total number of records in the table).

2) Join vs composite join

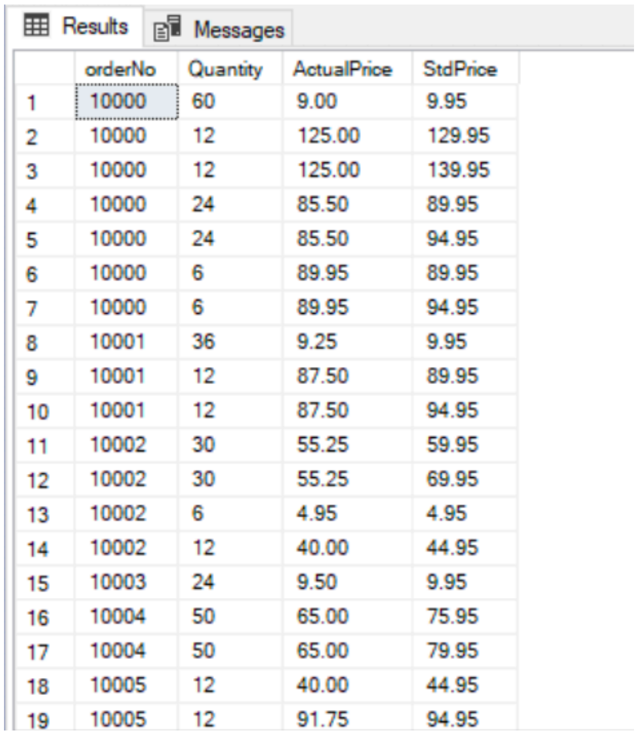
User story:

We want to retrieve **OrderLine** items for all the products and see the difference between actual price and the standard price specified in the price table. In order to do this, we run a join query based on **ProductNo** and select columns we are interested in (**OrderNo, Quantity, ActualPrice, StdPrice**):



```
SELECT OrderLine.orderNo, OrderLine.Quantity,
OrderLine.ActualPrice, Price.StdPrice
FROM OrderLine
JOIN Price
ON OrderLine.ProductNo = Price.ProductNo;
```

Output received:

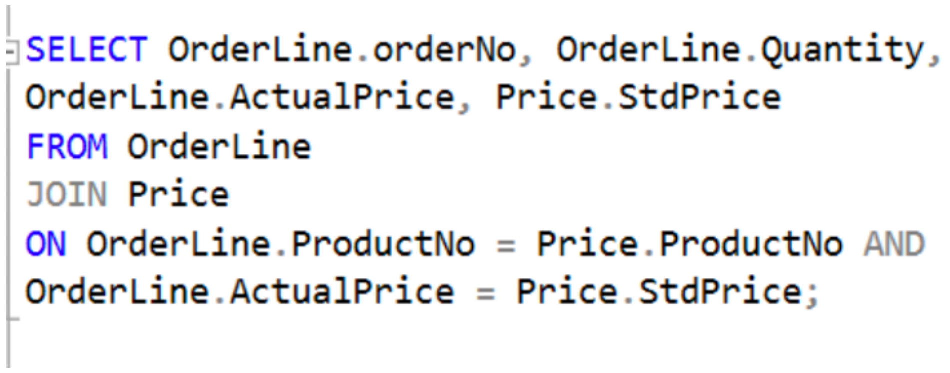


Results | Messages

	orderNo	Quantity	ActualPrice	StdPrice
1	10000	60	9.00	9.95
2	10000	12	125.00	129.95
3	10000	12	125.00	139.95
4	10000	24	85.50	89.95
5	10000	24	85.50	94.95
6	10000	6	89.95	89.95
7	10000	6	89.95	94.95
8	10001	36	9.25	9.95
9	10001	12	87.50	89.95
10	10001	12	87.50	94.95
11	10002	30	55.25	59.95
12	10002	30	55.25	69.95
13	10002	6	4.95	4.95
14	10002	12	40.00	44.95
15	10003	24	9.50	9.95
16	10004	50	65.00	75.95
17	10004	50	65.00	79.95
18	10005	12	40.00	44.95
19	10005	12	91.75	94.95

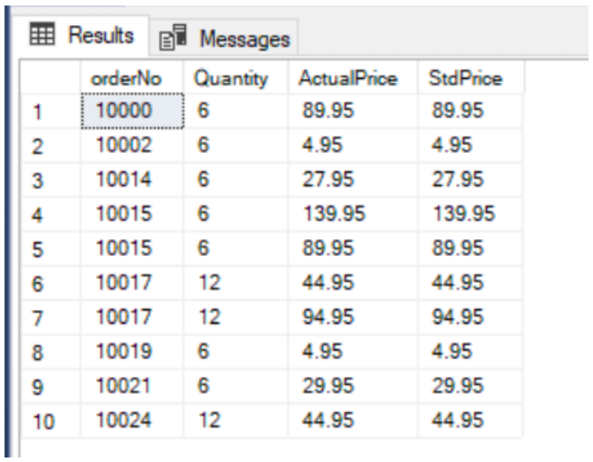
As you can see, now we have a list of items with actual and standard prices.

After reviewing the output, we decided that we want to see products, which actual price matches with the standard price from the **Price** table. To do that, we used composite join and have two conditions (**OrderLine.ProductNo = Price.ProductNo AND Orderline.ActualPrice = Price.StdPrice**):



```
SELECT OrderLine.orderNo, OrderLine.Quantity,
OrderLine.ActualPrice, Price.StdPrice
FROM OrderLine
JOIN Price
ON OrderLine.ProductNo = Price.ProductNo AND
OrderLine.ActualPrice = Price.StdPrice;
```

Output received:



Results | Messages

	orderNo	Quantity	ActualPrice	StdPrice
1	10000	6	89.95	89.95
2	10002	6	4.95	4.95
3	10014	6	27.95	27.95
4	10015	6	139.95	139.95
5	10015	6	89.95	89.95
6	10017	12	44.95	44.95
7	10017	12	94.95	94.95
8	10019	6	4.95	4.95
9	10021	6	29.95	29.95
10	10024	12	44.95	44.95

As a result, we successfully retrieved all the instances when the actual price and standard price are the same.

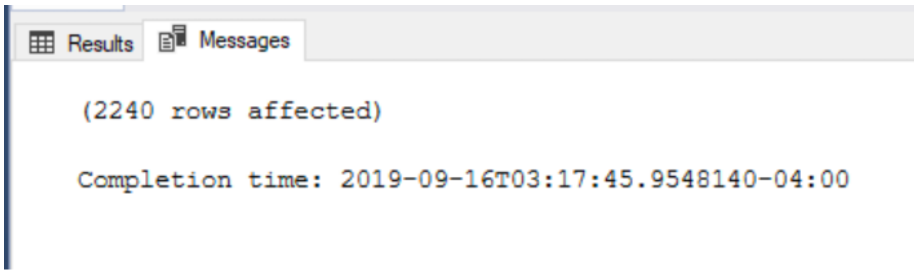
3) Cross join

User story:

As part of our series of experiments we would like to see all the possible combinations of product and its actual price as well as standard price. This gives us a broad perspective on what we can potentially have in our product and prices strategy. In order to see the combinations of actual and standard prices for the products that we have, we should run a **cross join query:**

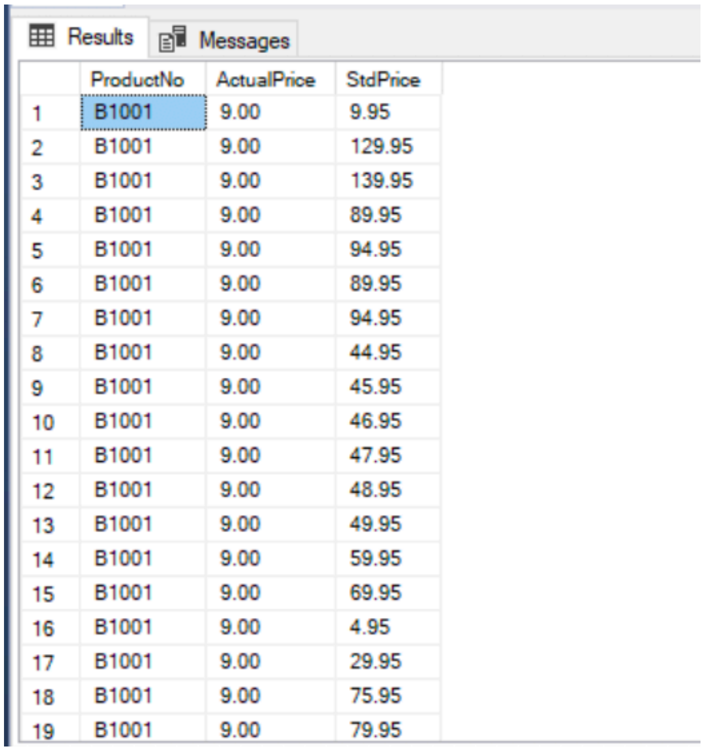
```
SELECT OrderLine.ProductNo, OrderLine.ActualPrice, Price.StdPrice
FROM OrderLine
CROSS JOIN Price
order by ProductNo
```

Output received:



```
(2240 rows affected)

Completion time: 2019-09-16T03:17:45.9548140-04:00
```



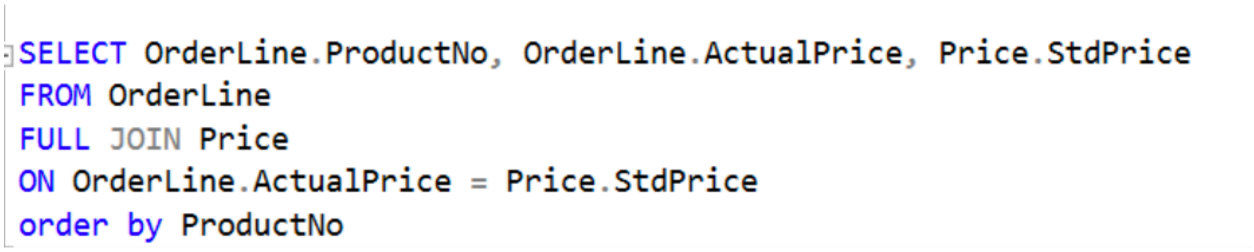
	ProductNo	ActualPrice	StdPrice
1	B1001	9.00	9.95
2	B1001	9.00	129.95
3	B1001	9.00	139.95
4	B1001	9.00	89.95
5	B1001	9.00	94.95
6	B1001	9.00	89.95
7	B1001	9.00	94.95
8	B1001	9.00	44.95
9	B1001	9.00	45.95
10	B1001	9.00	46.95
11	B1001	9.00	47.95
12	B1001	9.00	48.95
13	B1001	9.00	49.95
14	B1001	9.00	59.95
15	B1001	9.00	69.95
16	B1001	9.00	4.95
17	B1001	9.00	29.95
18	B1001	9.00	75.95
19	B1001	9.00	79.95

Note that the result gives many records, which makes sense – we produced all the possible combinations of **ProductNo, ActualPrice,** and the **StdPrice**.

4) Full Join

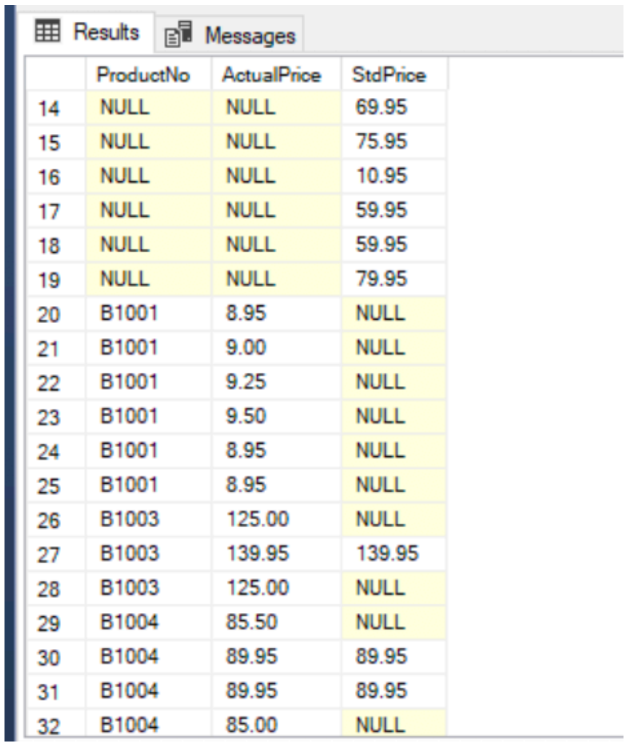
User story:

Our next task was to identify if products have both the actual price and standard price or not. To do that, we ran a **full join** query and easily found the missing match values:



```
SELECT OrderLine.ProductNo, OrderLine.ActualPrice, Price.StdPrice
FROM OrderLine
FULL JOIN Price
ON OrderLine.ActualPrice = Price.StdPrice
order by ProductNo
```

Output received:



Results Messages

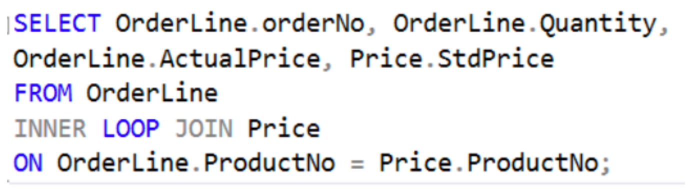
	ProductNo	ActualPrice	StdPrice
14	NULL	NULL	69.95
15	NULL	NULL	75.95
16	NULL	NULL	10.95
17	NULL	NULL	59.95
18	NULL	NULL	59.95
19	NULL	NULL	79.95
20	B1001	8.95	NULL
21	B1001	9.00	NULL
22	B1001	9.25	NULL
23	B1001	9.50	NULL
24	B1001	8.95	NULL
25	B1001	8.95	NULL
26	B1003	125.00	NULL
27	B1003	139.95	139.95
28	B1003	125.00	NULL
29	B1004	85.50	NULL
30	B1004	89.95	89.95
31	B1004	89.95	89.95
32	B1004	85.00	NULL

The output clearly shows which products have a standard and actual price and which ones do not. Also, it shows that some of the standard price items from the **Price** table are not linked to any products (see the records with NULL in **ProductNo**).

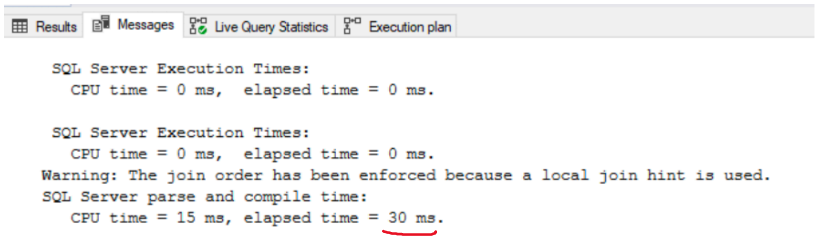
Part II

1) Series of Experiments with an inner and an outer joins. Use of Loop, Hash, Merge.

We decided to run different experiments using join hints. First one is the inner with the **LOOP** hint:

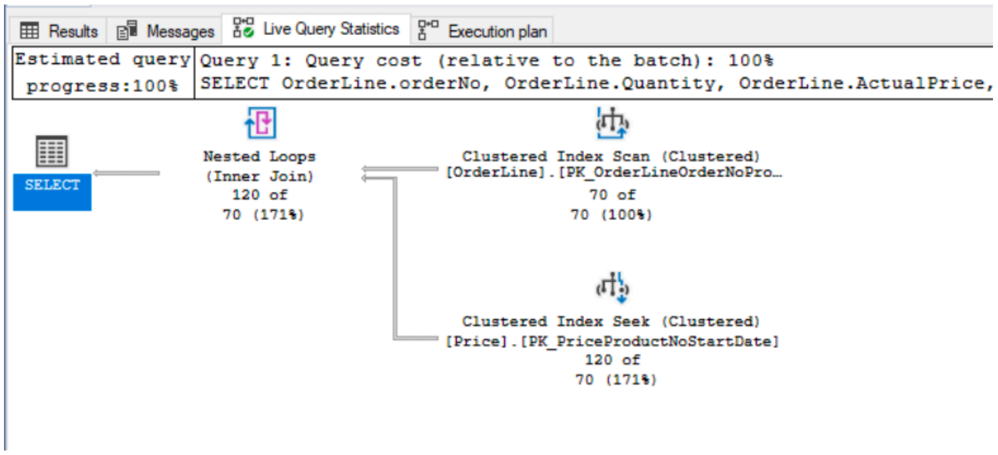


```
SELECT OrderLine.orderNo, OrderLine.Quantity,
OrderLine.ActualPrice, Price.StdPrice
FROM OrderLine
INNER LOOP JOIN Price
ON OrderLine.ProductNo = Price.ProductNo;
```

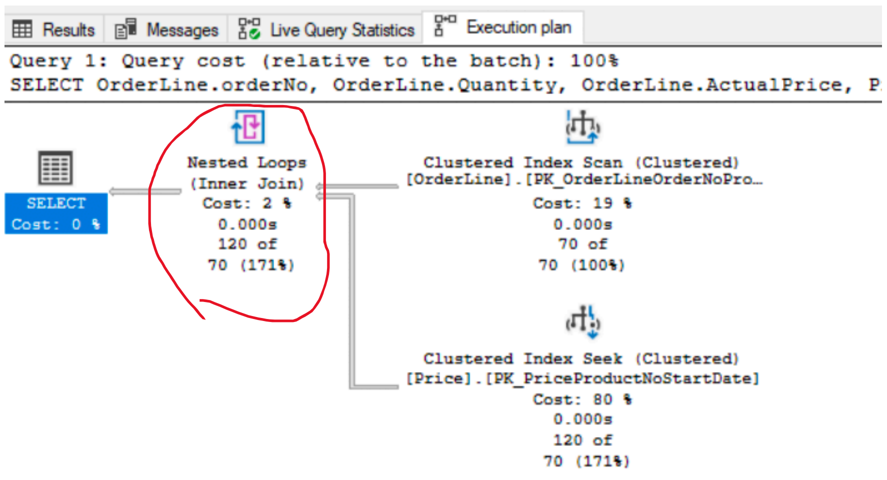


```
 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
Warning: The join order has been enforced because a local join hint is used.
SQL Server parse and compile time:
   CPU time = 15 ms, elapsed time = 30 ms.
```


Results
Messages
Live Query Statistics
Execution plan
SELECT
Nested Loops
(Inner Join)
120 of
70 (171%)
Clustered Index Scan (Clustered)
[OrderLine].[PK_OrderLineOrderNoPro...
70 of
70 (100%)
Clustered Index Seek (Clustered)
[Price].[PK_PriceProductNoStartDate]
120 of
70 (171%)

```
Estimated query
 progress:100%
Query 1: Query cost (relative to the batch): 100%
SELECT OrderLine.orderNo, OrderLine.Quantity, OrderLine.ActualPrice,
```



```
Query 1: Query cost (relative to the batch): 100%
SELECT OrderLine.orderNo, OrderLine.Quantity, OrderLine.ActualPrice, P
```

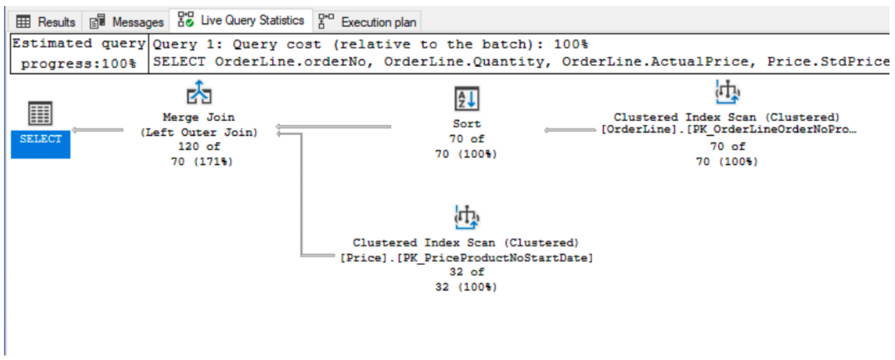
SELECT
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 2 %
0.000s
120 of
70 (171%)
Clustered Index Scan (Clustered)
[OrderLine].[PK_OrderLineOrderNoPro…
Cost: 19 %
0.000s
70 of
70 (100%)
Clustered Index Seek (Clustered)
[Price].[PK_PriceProductNoStartDate]
Cost: 80 %
0.000s
120 of
70 (171%)

Second operation that we ran as experiment is the **LEFT MERGE JOIN**:

```
SELECT OrderLine.orderNo, OrderLine.Quantity,
OrderLine.ActualPrice, Price.StdPrice
FROM OrderLine
LEFT MERGE JOIN Price
ON OrderLine.ProductNo = Price.ProductNo;
```

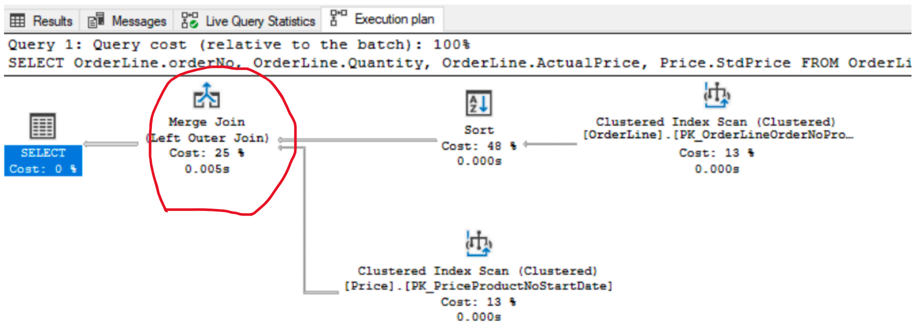
```
 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
Warning: The join order has been enforced because a local join hint :
SQL Server parse and compile time:
   CPU time = 38 ms, elapsed time = 38 ms.
```



```
Estimated query  Query 1: Query cost (relative to the batch): 100%
 progress:100%   SELECT OrderLine.orderNo, OrderLine.Quantity, OrderLine.ActualPrice, Price.StdPrice
```

SELECT
Merge Join
(Left Outer Join)
120 of
70 (171%)
Sort
70 of
70 (100%)
Clustered Index Scan (Clustered)
[OrderLine].[PK_OrderLineOrderNoPro…
70 of
70 (100%)
Clustered Index Scan (Clustered)
[Price].[PK_PriceProductNoStartDate]
32 of
32 (100%)



```
Query 1: Query cost (relative to the batch): 100%
SELECT OrderLine.orderNo, OrderLine.Quantity, OrderLine.ActualPrice, Price.StdPrice FROM OrderLi
```

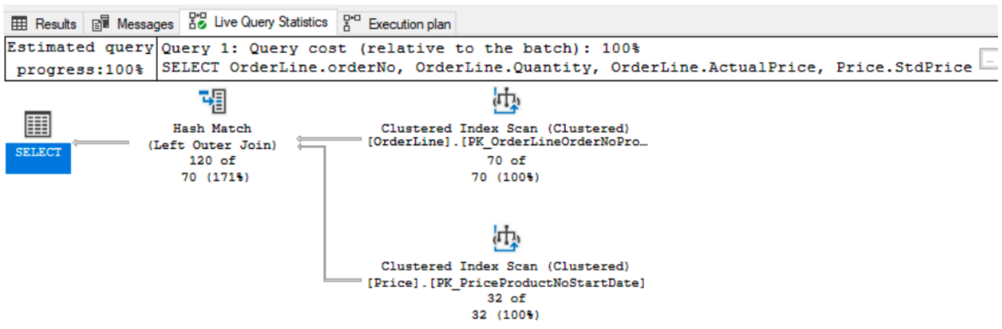
SELECT
Cost: 0 %
Merge Join
(Left Outer Join)
Cost: 25 %
0.005s
Sort
Cost: 48 %
0.000s
Clustered Index Scan (Clustered)
[OrderLine].[PK_OrderLineOrderNoPro...
Cost: 13 %
0.000s
Clustered Index Scan (Clustered)
[Price].[PK_PriceProductNoStartDate]
Cost: 13 %
0.000s

Second operation that we ran as experiment is the **LEFT HASH JOIN**:

```
SELECT OrderLine.orderNo, OrderLine.Quantity,
OrderLine.ActualPrice, Price.StdPrice
FROM OrderLine
LEFT HASH JOIN Price
ON OrderLine.ProductNo = Price.ProductNo;
```

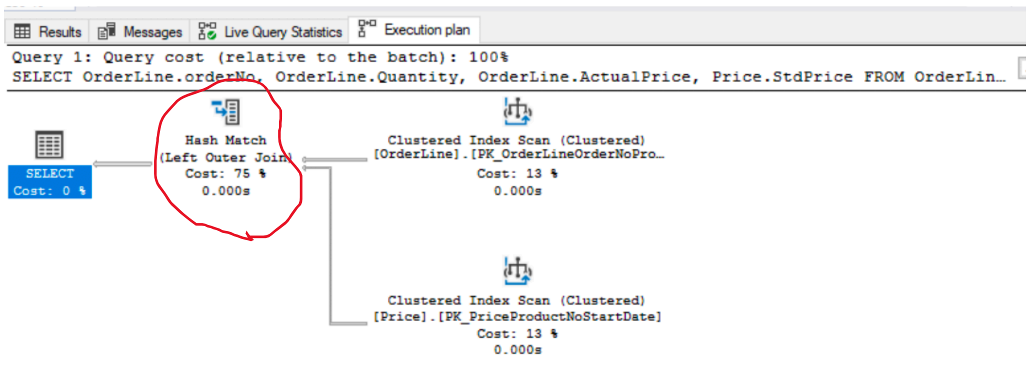
```
 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
Warning: The join order has been enforced because a local join hint
SQL Server parse and compile time:
   CPU time = 15 ms, elapsed time = 21 ms.
```


Results
Messages
Live Query Statistics
Execution plan

```
Estimated query  Query 1: Query cost (relative to the batch): 100%
 progress:100%   SELECT OrderLine.orderNo, OrderLine.Quantity, OrderLine.ActualPrice, Price.StdPrice
```

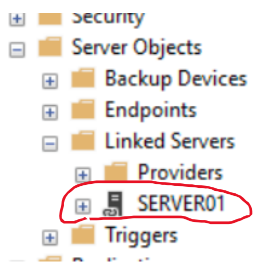
SELECT
Hash Match
(Left Outer Join)
120 of
70 (171%)
Clustered Index Scan (Clustered)
[OrderLine].[PK_OrderLineOrderNoPro...
70 of
70 (100%)
Clustered Index Scan (Clustered)
[Price].[PK_PriceProductNoStartDate]
32 of
32 (100%)



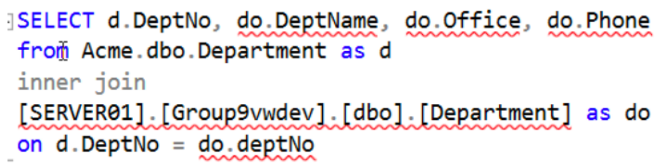
```
Query 1: Query cost (relative to the batch): 100%
SELECT OrderLine.orderNo, OrderLine.Quantity, OrderLine.ActualPrice, Price.StdPrice FROM OrderLin…
```

2) Regular Join vs Remote Join experiment:

We have a local table Department that has only five records and Department table from database on Azure that has one hundred records. In order to link two tables from different servers (local and azure) through the join, we have to add a linked server in SSMS:

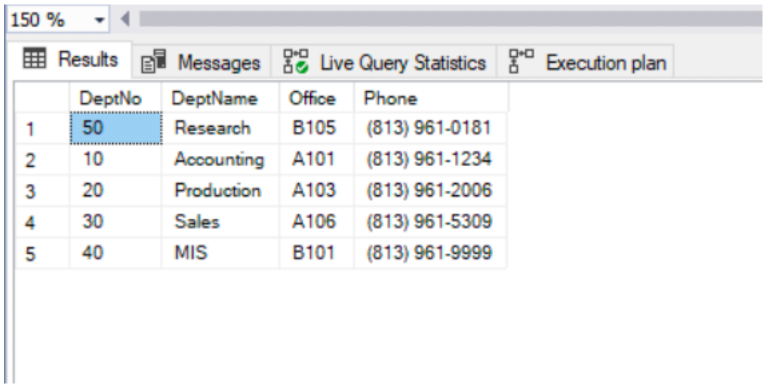


We created the linked server and ready to work with tables from the two servers. So, we want to see if the values match: for that we retrieve department name, office, and phone number based on the department NO (PK):



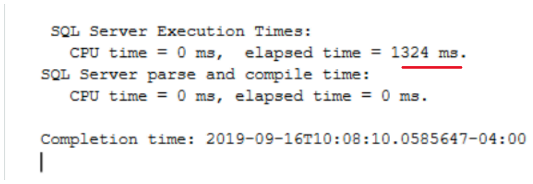
```
SELECT d.DeptNo, do.DeptName, do.Office, do.Phone
from Acme.dbo.Department as d
inner join
[SERVER01].[Group9vwdev].[dbo].[Department] as do
on d.DeptNo = do.deptNo
```

Output received:



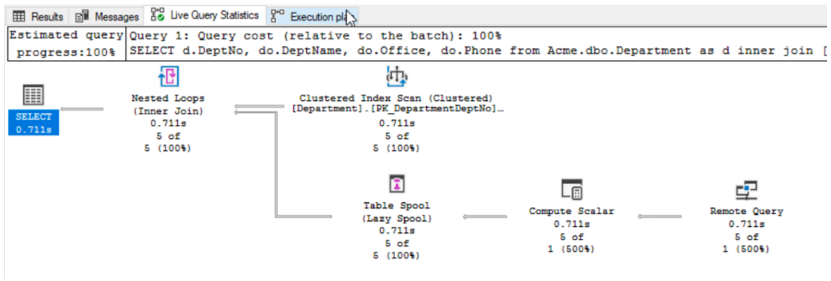
	DeptNo	DeptName	Office	Phone
1	50	Research	B105	(813) 961-0181
2	10	Accounting	A101	(813) 961-1234
3	20	Production	A103	(813) 961-2006
4	30	Sales	A106	(813) 961-5309
5	40	MIS	B101	(813) 961-9999

So, we see that the records match and the number of records is total five. Let's take a look at the performance statistics:

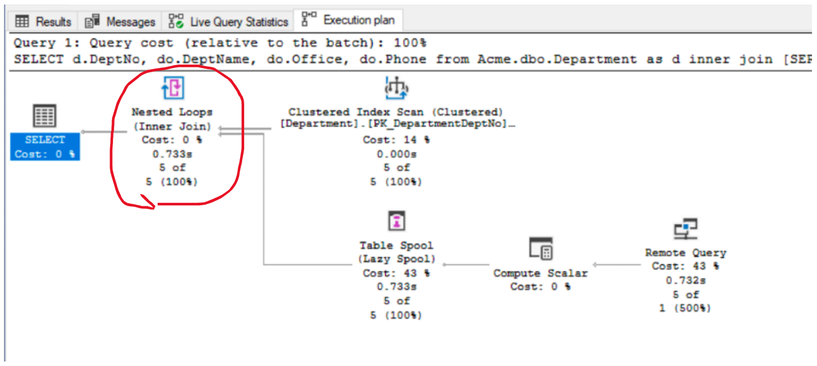


```
 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 1324 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

Completion time: 2019-09-16T10:08:10.0585647-04:00
```

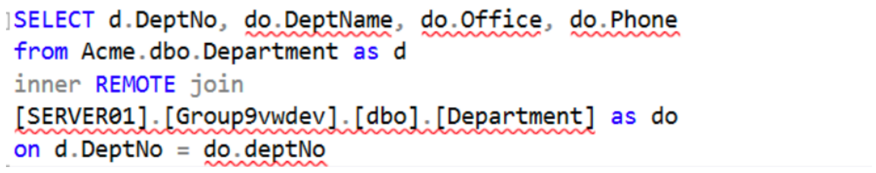


```
Estimated query progress:100%
Query 1: Query cost (relative to the batch): 100%
SELECT d.DeptNo, do.DeptName, do.Office, do.Phone from Acme.dbo.Department as d inner join [
```

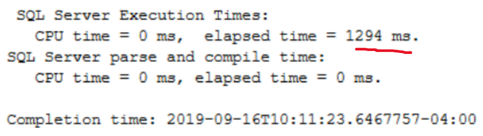


```
Query 1: Query cost (relative to the batch): 100%
SELECT d.DeptNo, do.DeptName, do.Office, do.Phone from Acme.dbo.Department as d inner join [SEF
```

It is recommended to practice using REMOTE whenever we join tables from different servers. It is better to follow the industry standards, especially when the right table (table from Azure db in our case) has way more records:

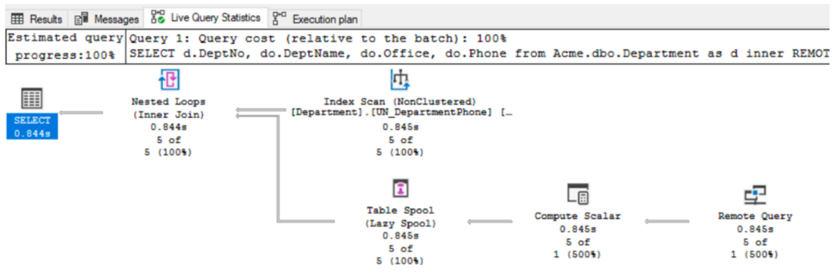


```
SELECT d.DeptNo, do.DeptName, do.Office, do.Phone
from Acme.dbo.Department as d
inner REMOTE join
[SERVER01].[Group9vwdev].[dbo].[Department] as do
on d.DeptNo = do.deptNo
```

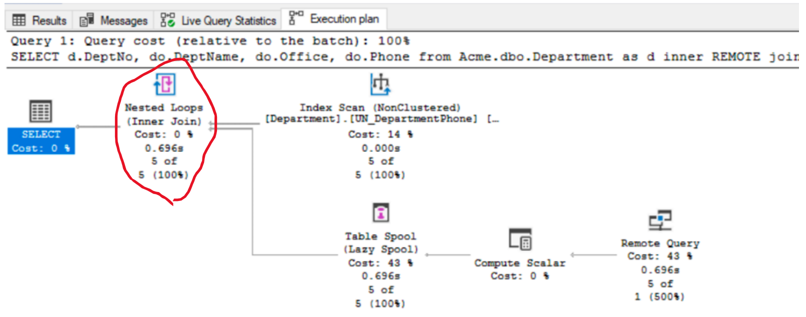


```
 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 1294 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

Completion time: 2019-09-16T10:11:23.6467757-04:00
```



```
Estimated query  Query 1: Query cost (relative to the batch): 100%
 progress:100%   SELECT d.DeptNo, do.DeptName, do.Office, do.Phone from Acme.dbo.Department as d inner REMOT
```



```
Query 1: Query cost (relative to the batch): 100%
SELECT d.DeptNo, do.DeptName, do.Office, do.Phone from Acme.dbo.Department as d inner REMOTE joir
```

The difference in performance is clearly defined in PART III under Remote vs regular join section on the next page.

Part III

Experiment Report:

Loop, Merge, and Hash join performance comparison results:

According to the experiment with comparison of Loop, Merge, and Hash joins, the performance depends not only on what sort of hint join you are using, but also the volume of the data. In our case, the number of records was relatively small – 120 items. It is important to note that every operation retrieves same number of records (120). Let's take a look at the performance below:

Inner loop join – elapsed time **30ms**, COST Nested loops: **0.000s**

Left merge join – elapsed time **38ms**, COST Merge: **0.005s**

Left hash join – elapsed time **21ms**, COST Hash match: **0.000s**

As you can see, the hash join turned out to be the fastest among all the hint joins that we ran.

Regular join vs remote join performance comparison results:

According to the experiment with comparison, both non-remote and remote approaches are quite fast and retrieve same records within milliseconds. Nevertheless, if you pay attention to the performance (completion time), using REMOTE shows a slightly better indicator: **1324MS** (non-remote) vs **1294MS** (remote). The "cost" under Nested Loops in the execution plan shows **0.733S** for regular join whereas remote join shows only **0.696S.**

After execution of the operations that we ran as part of this assignments, we have a better understanding of the query performance. First of all, with the use of hint joins, we achieve efficiency. For instance, the hash join is known to execute in two stages: build and probe. During the build, it goes through all the rows on the equijoin keys and creates a hash-table in memory. At the probe stage, it is going through the rows from the second input, hashes those rows on the same equijoin keys and looks for matching values in the hash table. In our experiment the hash hint join approach showed the best result in terms of performance. The merge join is known to be the most effective among the join operators. It always has two inputs: right and left and it supports all the logical join operations (inner join, left, right, full outer join, etc).The loop join is known to be very effective whenever we have small outer input and the inner input is sorted and large. In our case the inner loop join was on the second place and it took only 30ms to run the operation. Overall, from our experiment we learned that the use of hints really depends on type of join operation you are trying to run and also the structure of the join itself. In case with REMOTE join operation, we learned that it is way more efficient to use REMOTE instead of regular join since most of the time the outside linked database has more records. In such cases REMOTE operations helps with the performance: it was very clear that the time it takes to perform was faster than in the regular join (**1294MS vs 1324MS**). Now, imagine that the number of records in the Azure database was more than 100,000? The difference in time would even more significant. All in all, the REMOTE join experiment showed us that it is more efficient to use it whenever we have a right table outside of the local DB server.