DBT ASSIGNMENT-2

NAME: ASHWIN KRISHNA P

SRN: PES1201801465

SEM: 5 , SECTION: F

For this assignment all the operations are performed on objects of the database AIRDT

Analysing query optimization techniques

Queries containing **Order by**, **Group by** clause:

To retrieve ratio of gender of passengers from PASSENGER2 table.

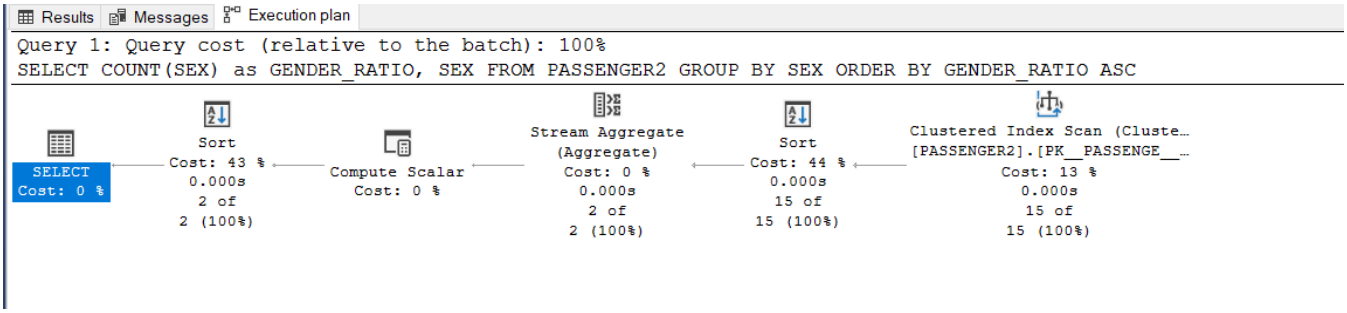
```
--ORDER BY, GROUP BY--
SELECT COUNT(SEX) as GENDER_RATIO, SEX
FROM PASSENGER2
GROUP BY SEX
ORDER BY GENDER_RATIO ASC
```

98 %

Results | Messages | Execution plan

	GENDER_RATIO	SEX
1	7	M
2	8	F

And we can see the execution plan where we see how the aggregate function, sorting and grouping takes place.



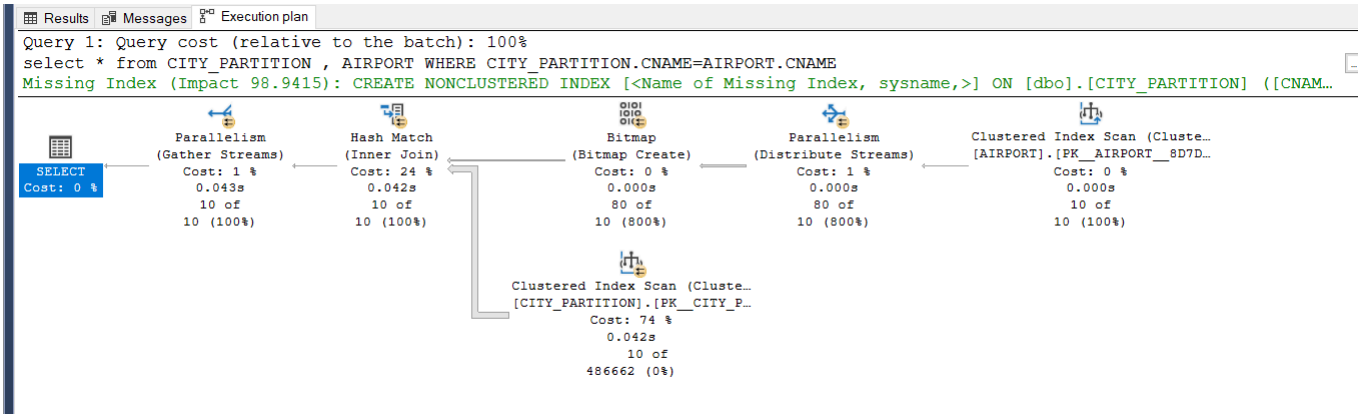
Queries with **JOIN**:

```
--JOIN--
select * from CITY_PARTITION JOIN AIRPORT ON CITY_PARTITION.CNAME=AIRPORT.CNAME

select * from CITY_PARTITION , AIRPORT WHERE CITY_PARTITION.CNAME=AIRPORT.CNAME
```

Both these queries give same result and same execution plan.

Results | Messages | Execution plan

	CITYCODE	CNAME	STATE	COUNTRY	AP_NAME	STATE	COUNTRY	CNAME	CITYCODE
1	121112	Houston	Texas	United States	George Bush Intercontinental Airport	Texas	United States	Houston	121112
2	111211	Tampa	Florida	United States	Tampa International Airport	Florida	United States	Tampa	111211
3	100000	Louisville	Kentucky	United States	Louisville International Airport	Kentucky	United States	Louisville	100000
4	120190	San Francisco	California	United States	San Francisco International Airport	California	United States	San Francisco	120190
5	121232	Frankfurt	Hesse	Germany	Frankfurt Airport	Hesse	Germany	Frankfurt	121232
6	132222	New York City	New York	United States	John F. Kennedy International Airport	New York	United States	New York City	132222
7	190908	Fort Worth	Texas	United States	Dallas/Fort Worth International Airport	Texas	United States	Fort Worth	190908
8	487787	Delhi	Delhi	India	Indira GandhiInternational Airport	Delhi	India	Delhi	487787
9	581187	Chandigarh	Chandigarh	India	Chandigarh International Airport	Chandigarh	India	Chandigarh	581187
10	230532	Mumbai	Maharashtra	India	Chhatrapati Shivaji International Airport	Maharashtra	India	Mumbai	230532



Here we can see how the inner join takes place and also, we notice the high I/O cost below.

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	486662
Actual Number of Rows for All Executions	10
Actual Number of Batches	0
Estimated I/O Cost	2.36477
Estimated Operator Cost	2.4988 (74%)
Estimated Subtree Cost	2.4988
Estimated CPU Cost	0.134028
Estimated Number of Executions	1
Number of Executions	8
Estimated Number of Rows for All Executions	486662
Estimated Number of Rows Per Execution	486662
Estimated Number of Rows to be Read	486662
Estimated Row Size	54 B
Actual Rebinds	0
Actual Rewinds	0
Partitioned	True
Actual Partition Count	2
Ordered	False
Node ID	5

Predicate
PROBE([Bitmap1005],[AIRDT].[dbo].[CITY_PARTITION].[CNAME])
Object
[AIRDT].[dbo].[CITY_PARTITION].[PK__CITY_PAR__D50A2A65DD14A0D1]
Output List
[AIRDT].[dbo].[CITY_PARTITION].CITYCODE, [AIRDT].[dbo].[CITY_PARTITION].CNAME, [AIRDT].[dbo].[CITY_PARTITION].STATE, [AIRDT].[dbo].[CITY_PARTITION].COUNTRY

So I created a non-clustered index on CNAME and then I could notice the significant reduction in the I/O cost which improves the performance.

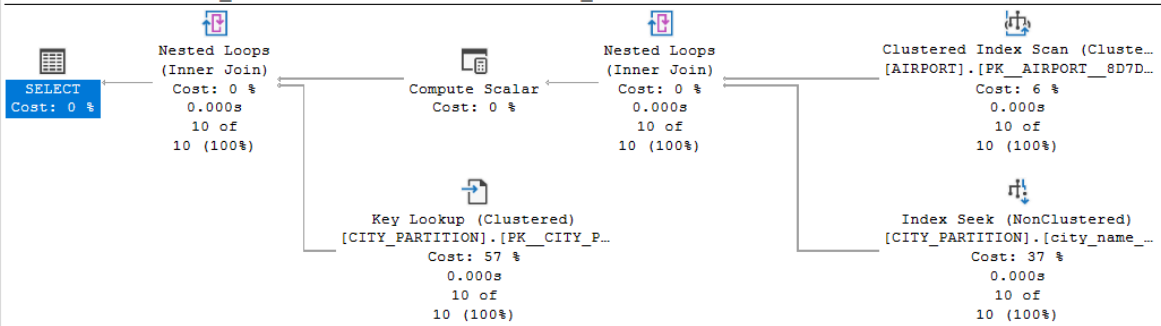
And we can say that creating a non-clustered index has really helped in improving the performance.

```
--JOIN--
--after creating index--
create index city_name_idx on CITY_PARTITION(CNAME) ON FGROUP2;

select * from CITY_PARTITION JOIN AIRPORT ON CITY_PARTITION.CNAME=AIRPORT.CNAME

select * from CITY_PARTITION , AIRPORT WHERE CITY_PARTITION.CNAME=AIRPORT.CNAME
```

98 %

Results | Messages | Execution plan

```
Query 1: Query cost (relative to the batch): 100%
select * from CITY_PARTITION JOIN AIRPORT ON CITY_PARTITION.CNAME=AIRPORT.CNAME
```



Index Seek (NonClustered)

Scan a particular range of rows from a nonclustered index.

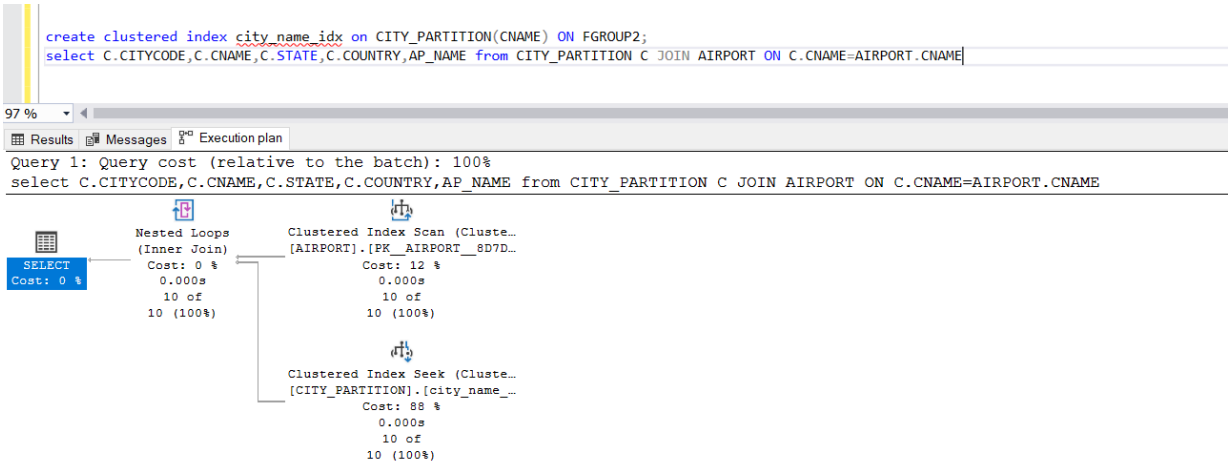
Physical Operation	Index Seek
Logical Operation	Index Seek
Actual Execution Mode	Row
Estimated Execution Mode	Row
Storage	RowStore
Number of Rows Read	10
Actual Number of Rows for All Executions	10
Actual Number of Batches	0
Estimated Operator Cost	0.0195249 (37%)
Estimated I/O Cost	0.003125
Estimated Subtree Cost	0.0195249
Estimated CPU Cost	0.0001581
Estimated Number of Executions	10
Number of Executions	10
Estimated Number of Rows for All Executions	10
Estimated Number of Rows to be Read	1
Estimated Number of Rows Per Execution	1
Estimated Row Size	28 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	4

Object
[AIRDT].[dbo].[CITY_PARTITION].[city_name_idx]

Output List
[AIRDT].[dbo].[CITY_PARTITION].CITYCODE, [AIRDT].[dbo].[CITY_PARTITION].CNAME

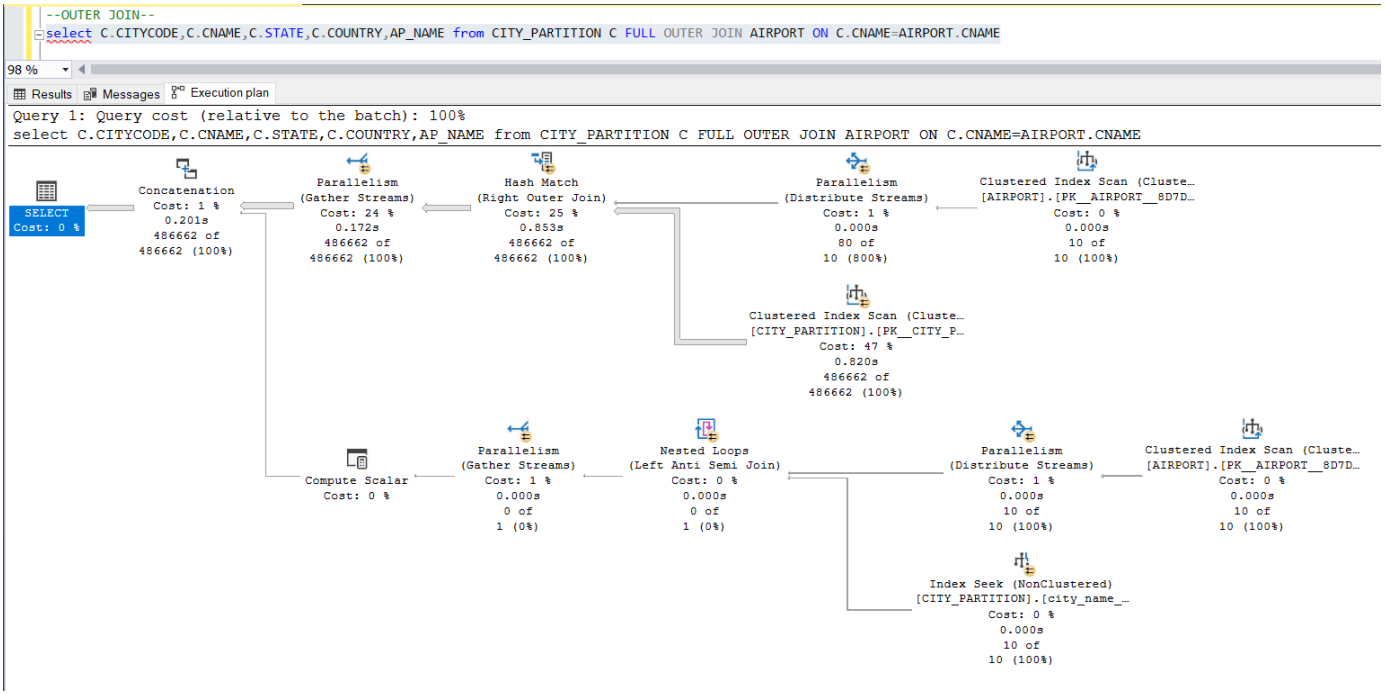
Seek Predicates
Seek Keys[1]: Prefix: [AIRDT].[dbo].[CITY_PARTITION].CNAME = Scalar Operator([AIRDT].[dbo].[AIRPORT].[CNAME])

Now I drop the non-clustered index, drop Foreign key from AIRPORT table, and I drop primary key on CITY_PARTITION just to see the performance of a clustered index. I create a clustered index, run the same query and we can see very simple execution plan generated! And cost is reduced same as the non-clustered index.



So clustered index is better for operations where we have JOINs.

Query with **Full Outer Join**:-



We can observe the complex execution plan.

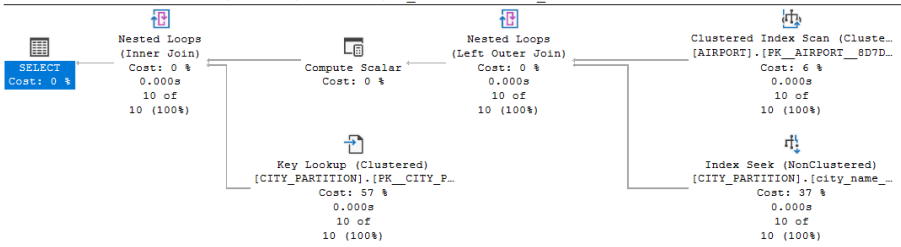
But there are many NULL values in the AP_NAME column and I specify IS NOT NULL in where condition to not retrieve records with AP_NAME as NULL and immediately, we can notice the simpler execution plan.

```
--OUTER JOIN--
select C.CITYCODE,C.CNAME,C.STATE,C.COUNTRY,AP_NAME from CITY_PARTITION C FULL OUTER JOIN AIRPORT ON C.CNAME=AIRPORT.CNAME
where AIRPORT.AP_NAME IS NOT NULL
```

98 %

Results | Messages | Execution plan

Query 1: Query cost (relative to the batch): 100%
select C.CITYCODE,C.CNAME,C.STATE,C.COUNTRY,AP_NAME from CITY_PARTITION C FULL OUTER JOIN AIRPORT ON C.CNAME=AIRPORT.CNAME where AIRPORT.AP_NAME IS NOT NULL



Use case-1:-

Consider below queries, both give same output table.

```
--To retrieve CNAME,AP_NAME located in India except Delhi and Maharashtra--
 --1--
select C.CNAME,A.AP_NAME
 from CITY_PARTITION C, AIRPORT A
 where C.COUNTRY='India' and C.CITYCODE=A.CITYCODE and C.STATE NOT IN ('Delhi','Maharashtra');

 --2--
select C.CNAME,AP.AP_NAME
 from CITY_PARTITION C, (select * from AIRPORT) AP
 where C.COUNTRY='India' and C.CITYCODE=AP.CITYCODE and C.STATE NOT IN ('Delhi','Maharashtra');
```

Results | Messages | Execution plan

	CNAME	AP_NAME
1	Chandigarh	Chandigarh International Airport

Execution plan of the first query is simple and predictable. In second query I am retrieving whole table AIRPORT as subquery , But the execution plan is interesting.

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	10
Actual Number of Rows for All Executions	10
Actual Number of Batches	0
Estimated Operator Cost	0.003293 (10%)
Estimated I/O Cost	0.003125
Estimated CPU Cost	0.000168
Estimated Subtree Cost	0.003293
Number of Executions	1
Estimated Number of Executions	1
Estimated Number of Rows to be Read	10
Estimated Number of Rows Per Execution	10
Estimated Row Size	70 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Node ID	1

Object
[AIRDT].[dbo].[AIRPORT].[PK__AIRPORT__8D7DB8040861628A]
Output List
[AIRDT].[dbo].[AIRPORT].AP_NAME, [AIRDT].[dbo].[AIRPORT].CITYCODE

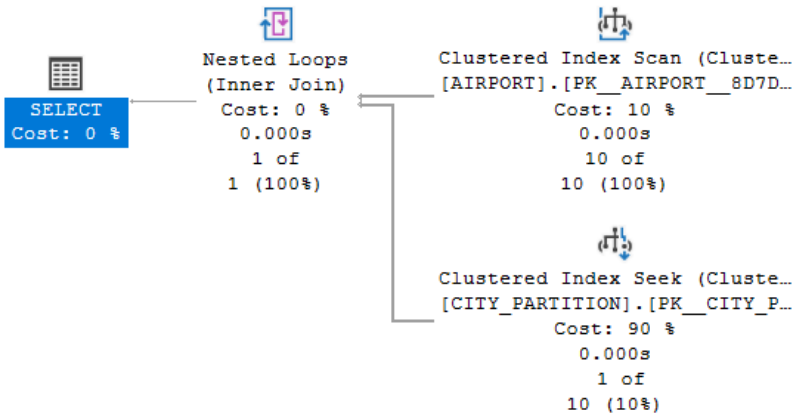
In the output list it is giving only the required column AP_NAME and CITYCODE instead of selecting all the columns even though there is (select * from AIRPORT) as sub query. This is really a smart execution by the query optimizer.

This is how the selection criteria is pushed to appropriate level in the parse tree by the query optimizer.

We have used NOT IN, same query can be written using IN, result is the same.

```
--3--
select C.CNAME,AP.AP_NAME
from (select A.AP_NAME,A.CITYCODE,A.COUNTRY from AIRPORT A) AP, CITY_PARTITION C
where C.COUNTRY='India' and C.CITYCODE=AP.CITYCODE and C.STATE IN ('Chandigarh');
```

For above qiery1,2 and 3 the plan looks something like:



Now we have seen use of **NOT IN**, and the same operation can be performed by making use of **MINUS** operator.

In sql server EXCEPT works same as the MINUS operation, so the above queries can also be rewritten as:

```
--4--
-- EXCEPT works same as set operator MINUS in SQL SERVER --
(select C.CNAME,A.AP_NAME
from CITY_PARTITION C, AIRPORT A
where C.COUNTRY='India' and C.CITYCODE=A.CITYCODE)
EXCEPT
(select C.CNAME,AP.AP_NAME
from (select A.AP_NAME,A.CITYCODE,A.COUNTRY from AIRPORT A) AP, CITY_PARTITION C
where C.COUNTRY='India' and C.CITYCODE=AP.CITYCODE and C.STATE IN ('Delhi','Maharashtra'))
```

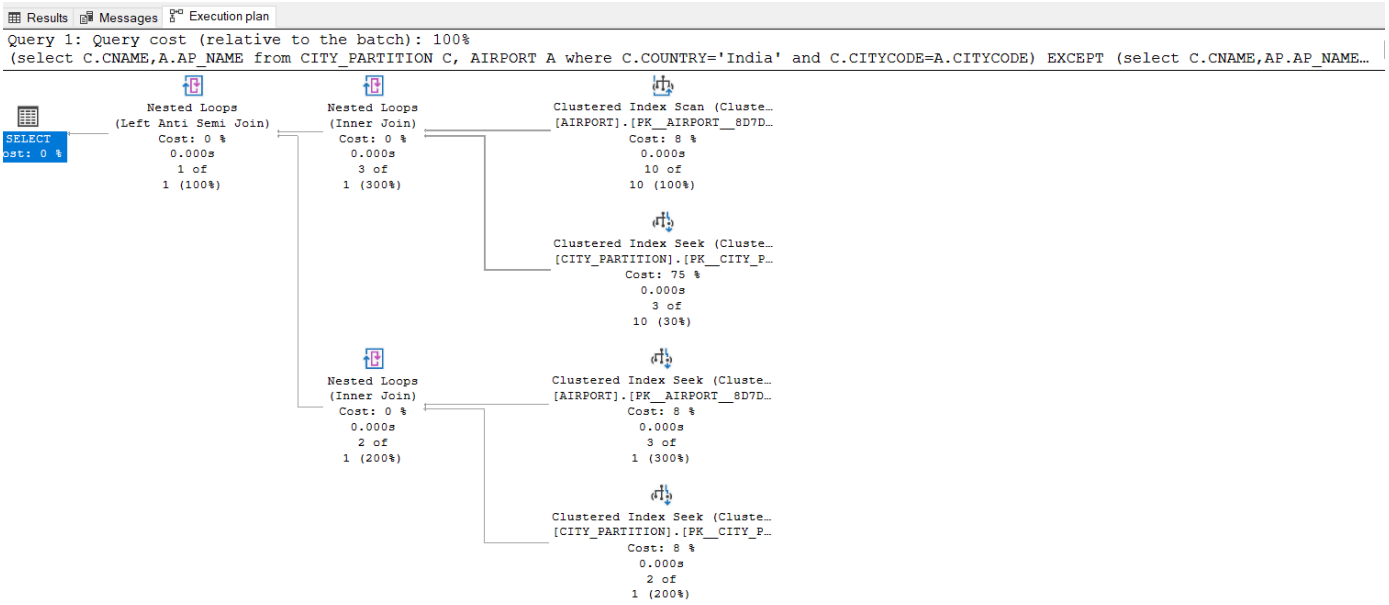
98 %

Results | Messages | Execution plan

	CNAME	AP_NAME
1	Chandigarh	Chandigarh International Airport

In this case we have to read the base tables twice and hence there are 2 nested loop joins.

And the execution plan is:



So in terms of simpler plan, we can say that this query using EXCEPT is not much efficient.

Below 4 methods give same result table:

```
--To retrieve CNAME,AP_NAME located in India except Delhi and Maharashtra--
--1--
select C.CNAME,A.AP_NAME
from CITY_PARTITION C, AIRPORT A
where C.COUNTRY='India' and C.CITYCODE=A.CITYCODE and C.STATE NOT IN ('Delhi','Maharashtra');

--2--
select C.CNAME,AP.AP_NAME
from CITY_PARTITION C, (select * from AIRPORT) AP
where C.COUNTRY='India' and C.CITYCODE=AP.CITYCODE and C.STATE NOT IN ('Delhi','Maharashtra');

--3--
select C.CNAME,AP.AP_NAME
from (select A.AP_NAME,A.CITYCODE,A.COUNTRY from AIRPORT A) AP, CITY_PARTITION C
where C.COUNTRY='India' and C.CITYCODE=AP.CITYCODE and C.STATE IN ('Chandigarh');

--4--
-- EXCEPT works same as set operator MINUS in SQL SERVER --
(select C.CNAME,A.AP_NAME
from CITY_PARTITION C, AIRPORT A
where C.COUNTRY='India' and C.CITYCODE=A.CITYCODE)
EXCEPT
(select C.CNAME,AP.AP_NAME
from (select A.AP_NAME,A.CITYCODE,A.COUNTRY from AIRPORT A) AP, CITY_PARTITION C
where C.COUNTRY='India' and C.CITYCODE=AP.CITYCODE and C.STATE IN ('Delhi','Maharashtra'))
```

98 %

Results | Messages | Execution plan

	CNAME	AP_NAME
1	Chandigarh	Chandigarh International Airport

So we have seen different methods of writing same query, which gives same result but when we analyse the execution plan, we can clearly tell how the query optimizer does its job and we can say which method is more efficient.

Use case-2:-

Query with **UNION** operator:

Here both the queries, query 1 and query 2 gives the same result. In both queries there is a UNION operation to UNION 2 sub queries with joins.

Both are written in different manner, but result is the same.

```sql
--union--
--1--
--SELECT(R) where condition U SELECT(S) where conditin
select C.CNAME,C.CITYCODE,A.AP_NAME from CITY_PARTITION C left join AIRPORT A on C.CITYCODE=A.CITYCODE where A.AP_NAME IS NOT NULL
union
select A.CNAME,A.CITYCODE,A.AP_NAME from CITY_PARTITION C full outer join AIRPORT A on C.CITYCODE=A.CITYCODE where A.AP_NAME IS NOT NULL

-- SELECT(R U S) where condition
select * from
(select C.CNAME,C.CITYCODE,A.AP_NAME from CITY_PARTITION C left join AIRPORT A on C.CITYCODE=A.CITYCODE
union
select A.CNAME,A.CITYCODE,A.AP_NAME from CITY_PARTITION C full outer join AIRPORT A on C.CITYCODE=A.CITYCODE ) T
where T.AP_NAME IS NOT NULL
```

97 %

Results | Messages | Execution plan

	CNAME	CITYCODE	AP_NAME
1	Chandigarh	581187	Chandigarh International Airport
2	Delhi	487787	Indira GandhiInternational Airport
3	Fort Worth	190908	Dallas/Fort Worth International Airport
4	Frankfurt	121232	Frankfurt Airport
5	Houston	121112	George Bush Intercontinental Airport
6	Louisville	100000	Louisville International Airport
7	Mumbai	230532	Chhatrapati Shivaji International Airport
8	New York City	132222	John F. Kennedy International Airport
9	San Francisco	120190	San Francisco International Airport
10	Tampa	111211	Tampa International Airport

The first query is written as, $SELECT_{condition}(R)$ U $SELECT_{condition}(S)$

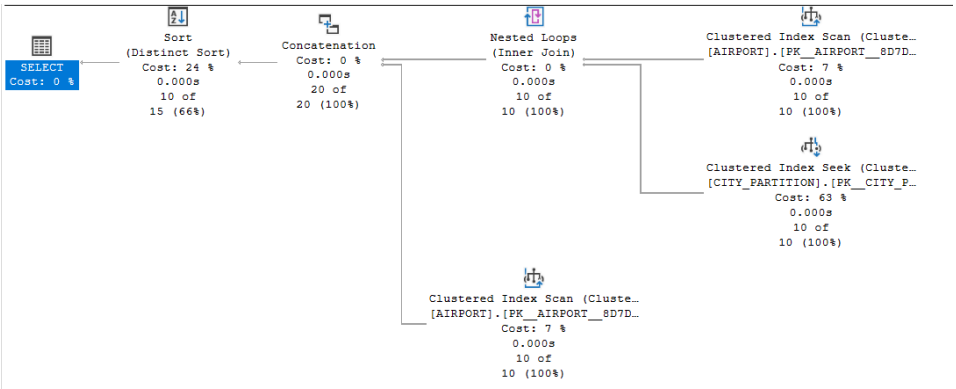
And second query is in the form of $SELECT_{condition}(R \cup S)$

Important rule for efficient query processing is to push the selection criteria down the tree as far as it goes.

By looking at the result we confirm the **algebraic law of query optimization(for selection)**, that is:

$SELECT_{condition}(R \cup S) = SELECT_{condition}(R)$ U $SELECT_{condition}(S)$

The execution plan looks like:



Use case-3:-

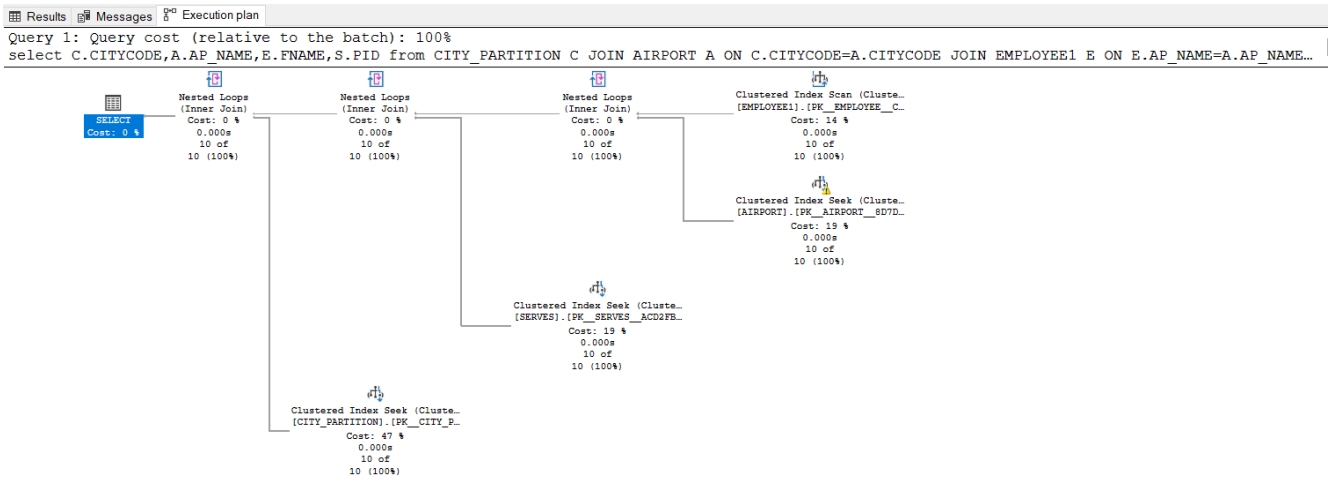
Joining multiple tables:

Here I am joining 4 tables CITY_PARTITION, AIRPORT, EMPLOYEE1 and SERVES

```sql
--multiple table join--
--1--
select C.CITYCODE,A.AP_NAME,E.FNAME,S.PID from
CITY_PARTITION C JOIN AIRPORT A ON C.CITYCODE=A.CITYCODE
JOIN EMPLOYEE1 E ON E.AP_NAME=A.AP_NAME
JOIN SERVES S ON S.SSN=E.SSN

--2--
select C.CITYCODE,A.AP_NAME,E.FNAME,S.PID from
EMPLOYEE1 E JOIN AIRPORT A ON E.AP_NAME=A.AP_NAME
JOIN CITY_PARTITION C ON C.CITYCODE=A.CITYCODE
JOIN SERVES S ON S.SSN=E.SSN
```

Here the order of joining is different in these 2 queries.

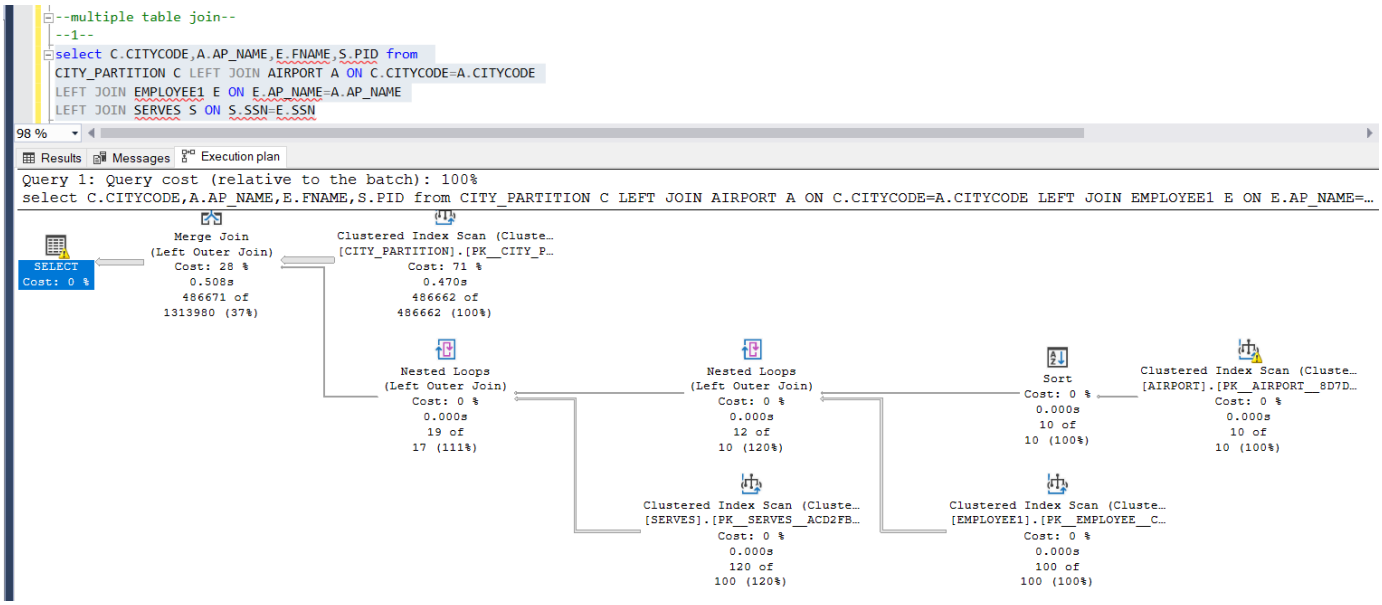
Both these queries give same result and the same execution plan.



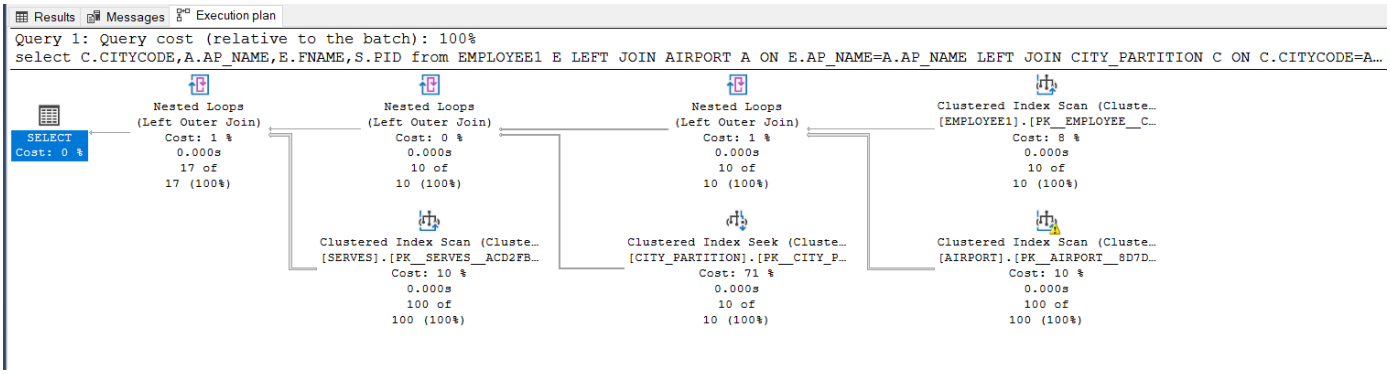
So we confirm that **Inner join is commutative.** i.e: table1 join table2 = table2 join table1

And **I did not notice any change in the performance because of the join order**.

Now let us see joining 4 tables using **left join**:



Writing same query in different join order gives different results:



So we confirm that **Left/right joins are not commutative.**

i.e: table1 left join table2 != table2 right join table1 . Results are different so I'm not analysing the execution plan further.

Understandings from UNIT-2:

- During query processing , in Relational algebra level we do transformations and try to find good transformations. In detailed query plan level we estimate the cost, generate and compare different plans so that the best plan can be opted for future.
- As we have seen important rule for efficient query processing is to push the selection criteria down the tree as far as it goes. A smart query optimizer takes more care about selections and makes sure that it retrieves only the required attributes even if there are sub queries which simply does 'select * from table' .
- Appropriate transformations of Algebraic expressions results in performance improvement . But no transformation is always good, All these transformations are task specific.
- For equi-join, where relations are not sorted and no indexes exist, hash based join technique is usually best.
- Creating appropriate index(s) improves the performance significantly. We have seen that creating a clustered index on join is usually good.
- So I have learnt many concepts of query processing, query optimization, different algorithms for doing this and many techniques about how the sql server does this job.

THANK YOU.