P4 Assignment: DW Performance

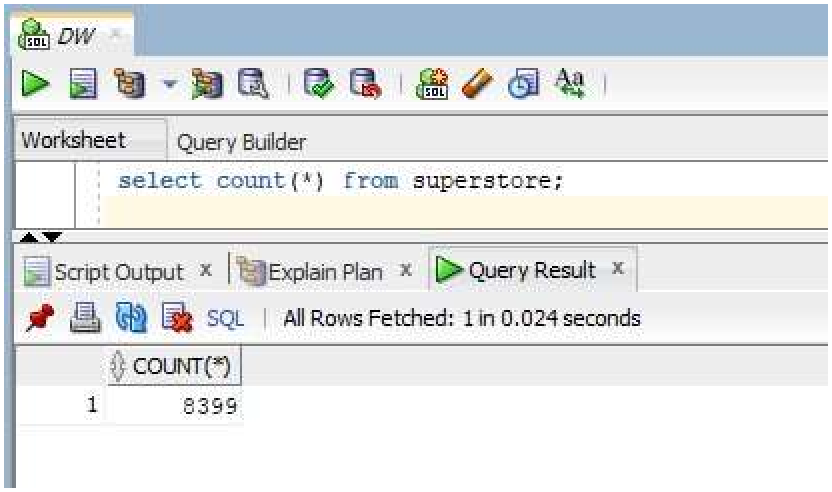
ISM 6208: Data Warehousing

Group 8 -

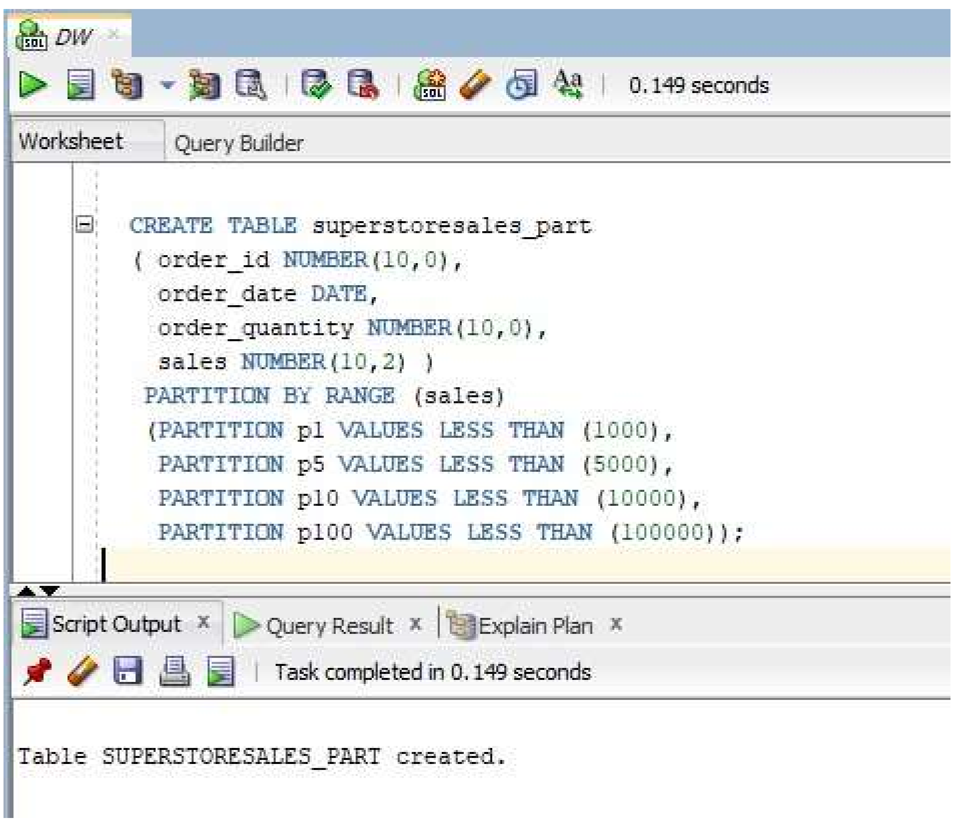
GOUTHAM BEERAM
RITIK GUPTA
PRASAD ACHARYA
ANKUR SRIVASTAVA
RISHABH MITTAL

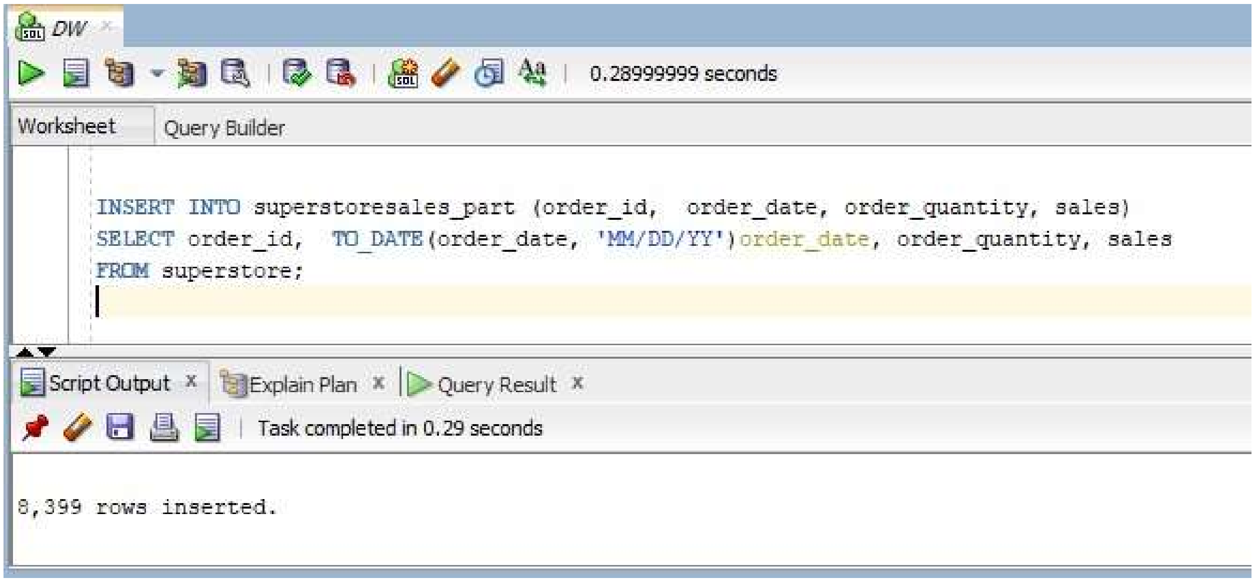
Idea 1: Simple Partitioning

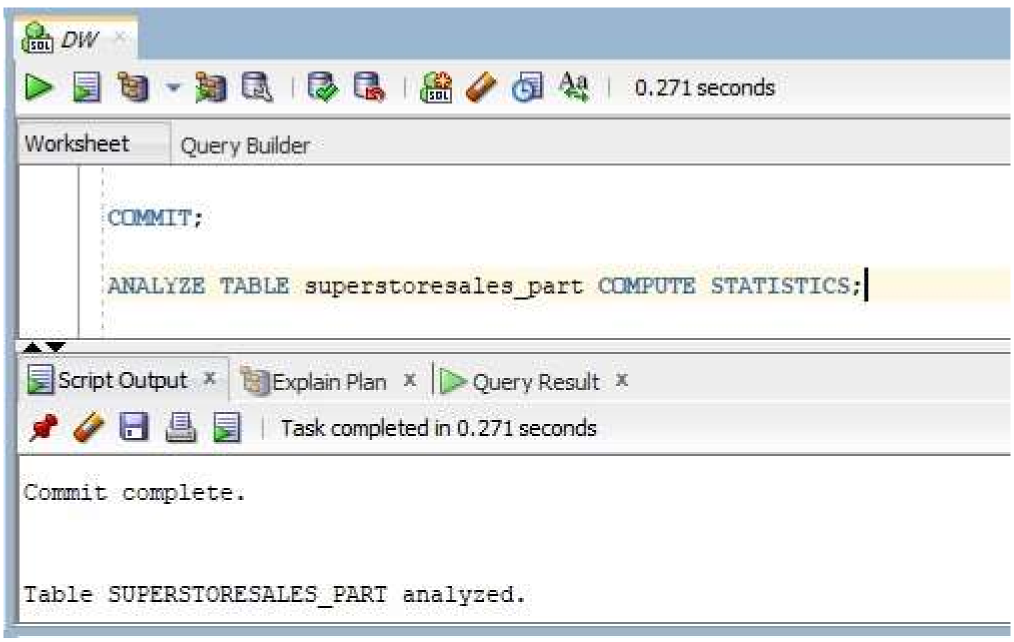
Superstore table contains 8399 rows.



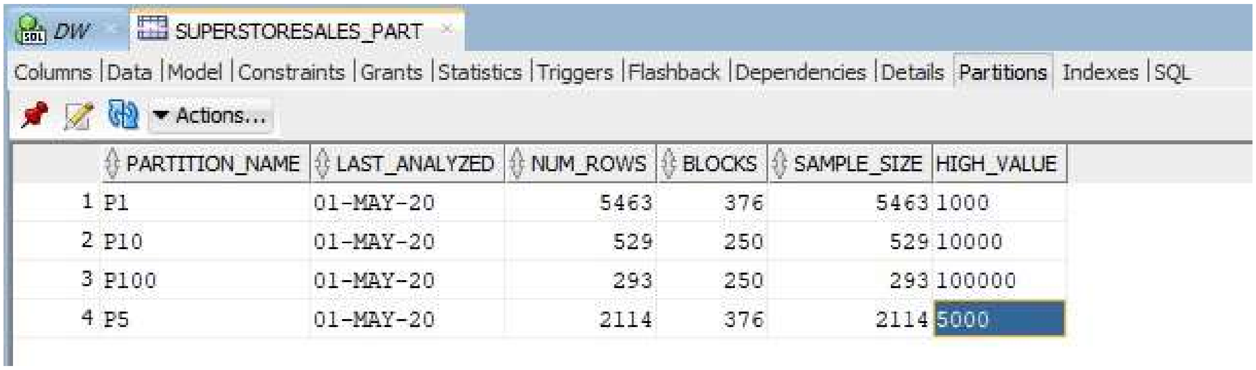
Below script extracts a portion of that data by sales to create a partitioned version of the table. This sets ups an opportunity to assess query against partitioned and non-partitioned versions of the table.



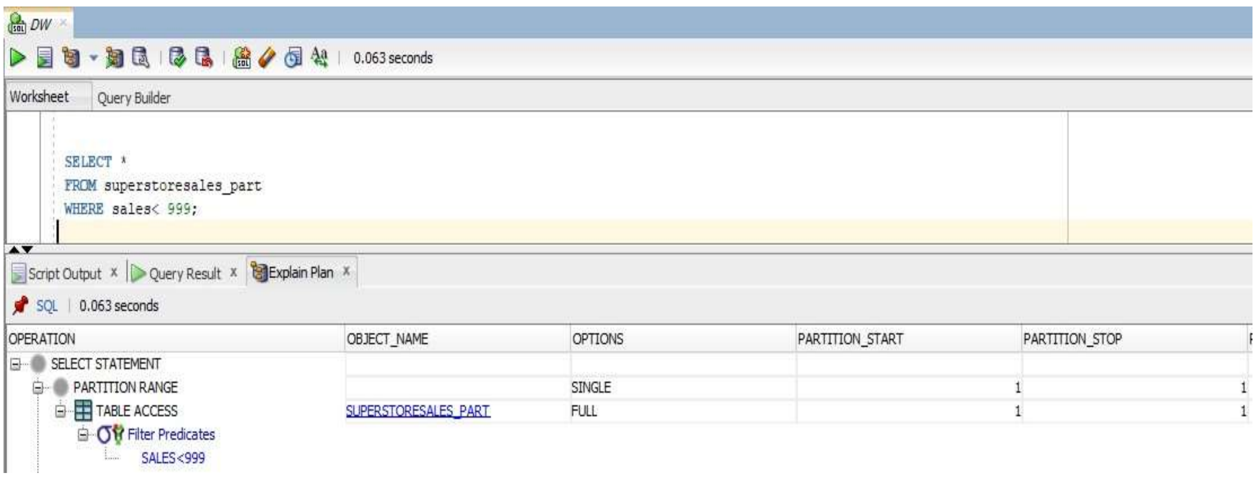




After creating a partitioned table and populating it with data, used the SQL Developer "Partitions" tab to check the structure. Noticed that there are 4 partitions in this example with distribution of rows as shown below.



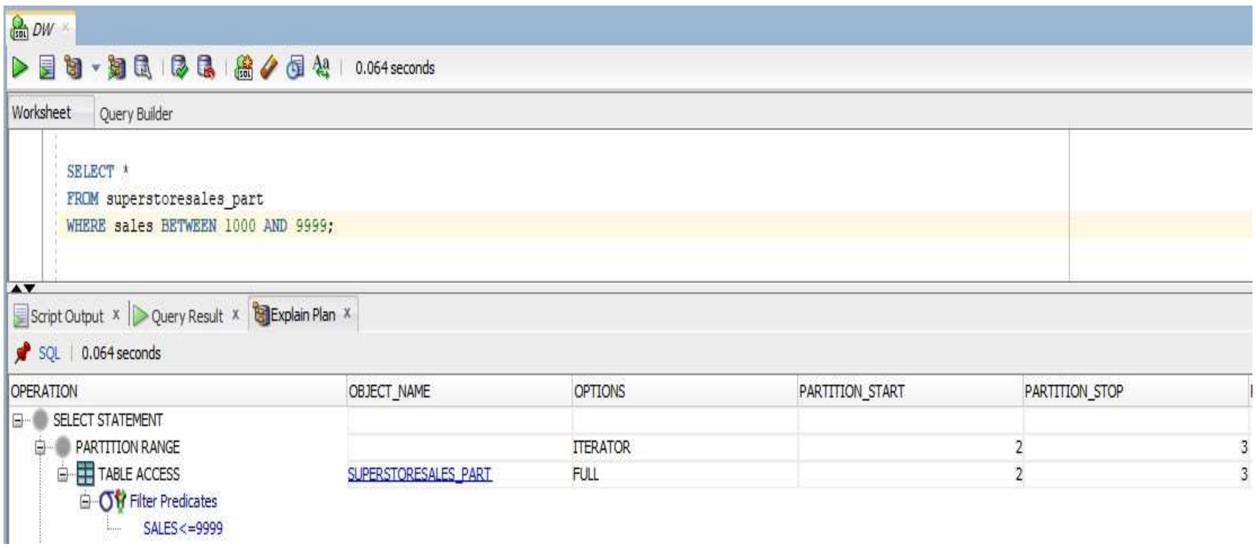
Run a query against the partitioned table. Since sales is the partitioning attribute the optimizer can eliminate 3 of the 4 partitions right at the start, looking at only the sales<999 data, hence the PARTITION RANGE (SINGLE) operation in the plan. Query took 0.063 seconds.



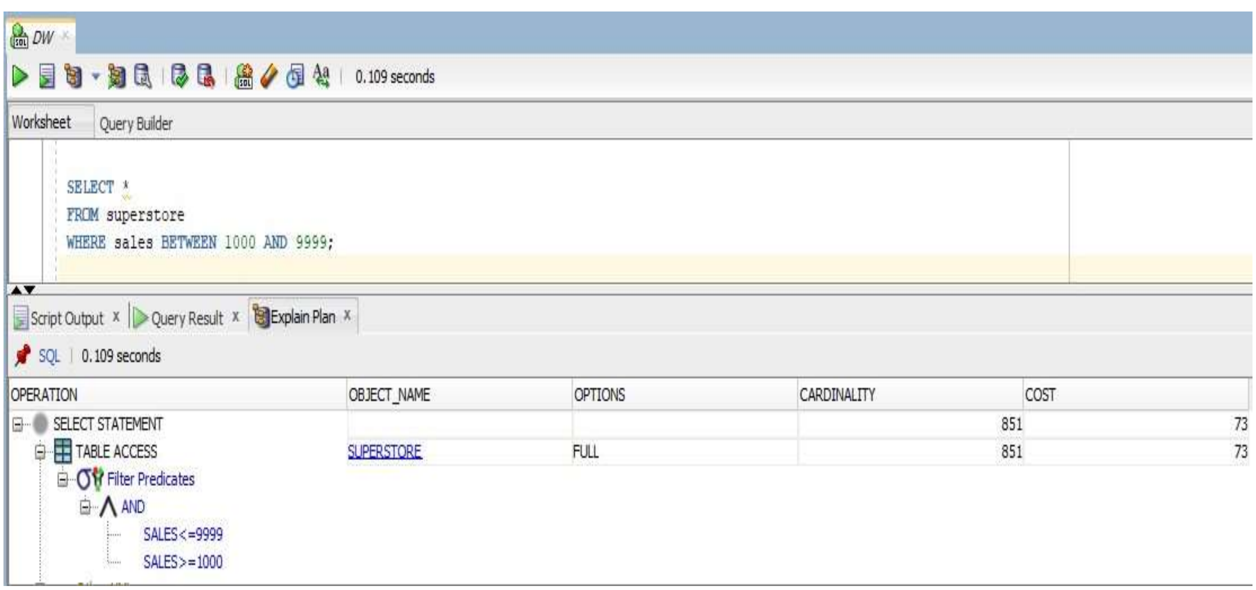
Now, Run this query against the non-partitioned version of the table. The same query took 0.116 seconds.

Run 2nd query against the partitioned table. Since sales is the partitioning attribute the optimizer looking at only the sales between 1000 and 9999 data, hence the PARTITION RANGE starts at 2 and stops at 3 in the plan. Query took 0.064 seconds.

In this query, the range is expanded just a bit, but it crosses a partition boundary. You can see that multiple ranges are being accessed in the PARTITION RANGE (ITERATOR) step. At the righthand side of the execution plan, you should also see the PARTITION_START and PARTITION_STOP, which lists the partition identification numbers that define the range.

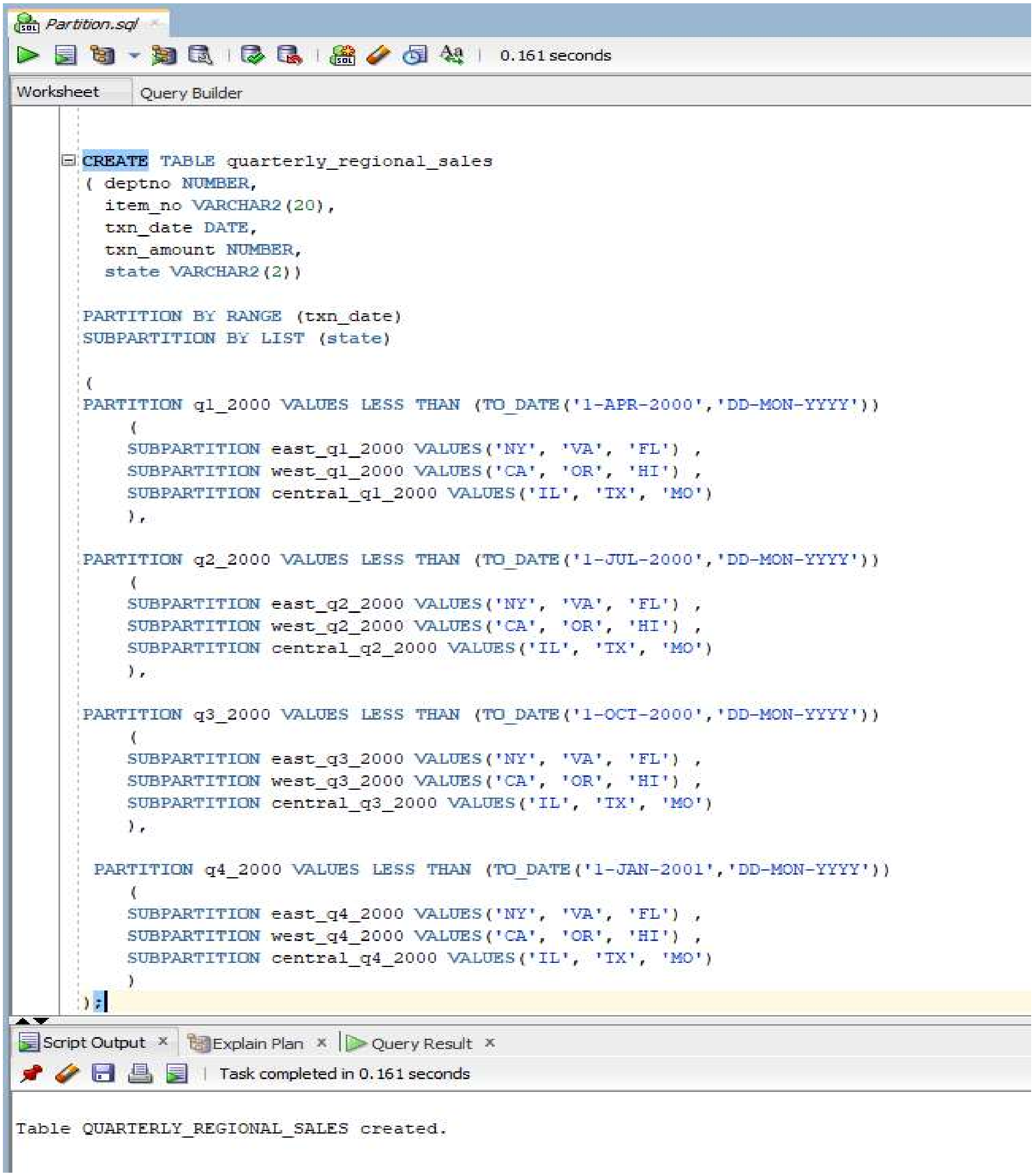


Now, Run this query against the non-partitioned version of the table. The same query took 0.109 seconds.



Idea 2: Composite Partitioning

In below example I have Created a Composite Range-List Partitioned Table called quarterly_regional_sales combining time-based range partitioning with geographic-based list subpartitioning. Composite range-list partitioning partitions data using the range method, and within each partition, sub-partitions the data further using the list method.




```
CREATE TABLE quarterly_regional_sales
( deptno NUMBER,
  item_no VARCHAR2(20),
  txn_date DATE,
  txn_amount NUMBER,
  state VARCHAR2(2))

PARTITION BY RANGE (txn_date)
SUBPARTITION BY LIST (state)

(
PARTITION q1_2000 VALUES LESS THAN (TO_DATE('1-APR-2000','DD-MON-YYYY'))
    (
    SUBPARTITION east_q1_2000 VALUES('NY', 'VA', 'FL') ,
    SUBPARTITION west_q1_2000 VALUES('CA', 'OR', 'HI') ,
    SUBPARTITION central_q1_2000 VALUES('IL', 'TX', 'MO')
    ),

PARTITION q2_2000 VALUES LESS THAN (TO_DATE('1-JUL-2000','DD-MON-YYYY'))
    (
    SUBPARTITION east_q2_2000 VALUES('NY', 'VA', 'FL') ,
    SUBPARTITION west_q2_2000 VALUES('CA', 'OR', 'HI') ,
    SUBPARTITION central_q2_2000 VALUES('IL', 'TX', 'MO')
    ),

PARTITION q3_2000 VALUES LESS THAN (TO_DATE('1-OCT-2000','DD-MON-YYYY'))
    (
    SUBPARTITION east_q3_2000 VALUES('NY', 'VA', 'FL') ,
    SUBPARTITION west_q3_2000 VALUES('CA', 'OR', 'HI') ,
    SUBPARTITION central_q3_2000 VALUES('IL', 'TX', 'MO')
    ),

 PARTITION q4_2000 VALUES LESS THAN (TO_DATE('1-JAN-2001','DD-MON-YYYY'))
    (
    SUBPARTITION east_q4_2000 VALUES('NY', 'VA', 'FL') ,
    SUBPARTITION west_q4_2000 VALUES('CA', 'OR', 'HI') ,
    SUBPARTITION central_q4_2000 VALUES('IL', 'TX', 'MO')
    )
);
```



```
Table QUARTERLY_REGIONAL_SALES created.
```

Partition.sql | QUARTERLY_REGIONAL_SALES

Columns | Data | Model | Constraints | Grants | Statistics | Triggers | Flashback | Dependencies | Details | Partitions | Indexes | SQL

Actions...

	PARTITION_NAME	LAST_ANALYZED	NUM_ROWS	BLOCKS	SAMPLE_SIZE	HIGH_VALUE
1	Q4_2000	01-MAY-20	0	0	(null)	TO_DATE(' 2001-01-01 00:00:00', 'SYYYY-MM-DD HH24:MI:SS', 'NLS_CALENDAR=GREGORIAN')
2	Q3_2000	01-MAY-20	0	0	(null)	TO_DATE(' 2000-10-01 00:00:00', 'SYYYY-MM-DD HH24:MI:SS', 'NLS_CALENDAR=GREGORIAN')
3	Q2_2000	01-MAY-20	0	0	(null)	TO_DATE(' 2000-07-01 00:00:00', 'SYYYY-MM-DD HH24:MI:SS', 'NLS_CALENDAR=GREGORIAN')
4	Q1_2000	01-MAY-20	0	0	(null)	TO_DATE(' 2000-04-01 00:00:00', 'SYYYY-MM-DD HH24:MI:SS', 'NLS_CALENDAR=GREGORIAN')

Partition.sql

0.41800001 seconds

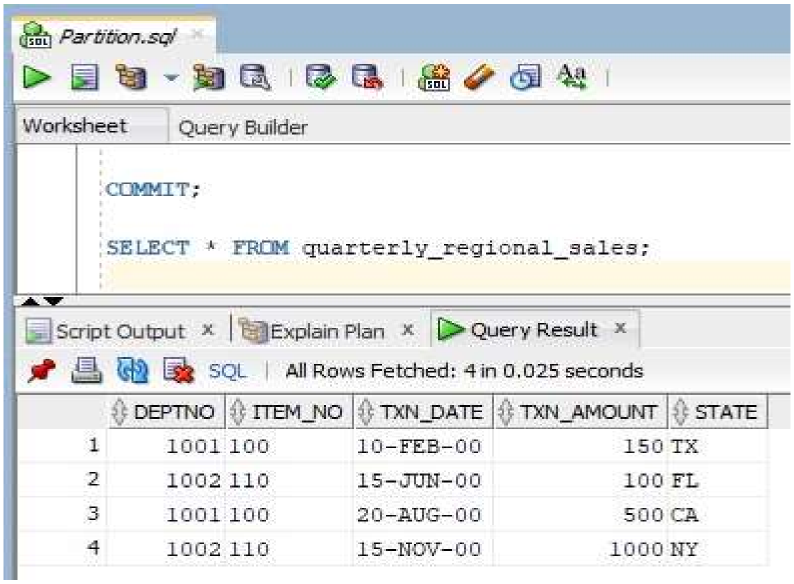
Worksheet | Query Builder

```sql
INSERT INTO quarterly_regional_sales VALUES (1001,100,'10-FEB-2000',150,'TX');
INSERT INTO quarterly_regional_sales VALUES (1002,110,'15-JUN-2000',100,'FL');
INSERT INTO quarterly_regional_sales VALUES (1001,100,'20-AUG-2000',500,'CA');
INSERT INTO quarterly_regional_sales VALUES (1002,110,'15-NOV-2000',1000,'NY');
```

Script Output | Explain Plan | Query Result

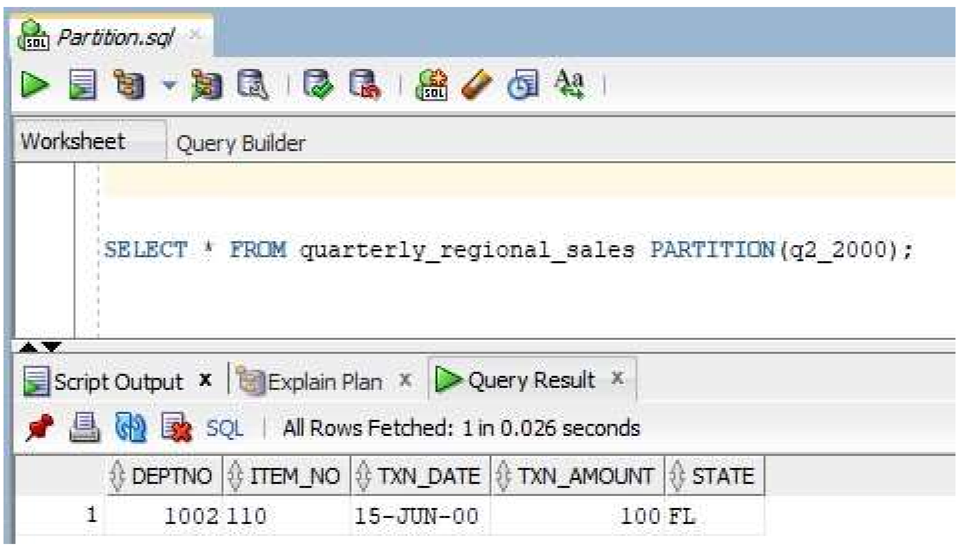
Task completed in 0.418 seconds

```
Table QUARTERLY_REGIONAL_SALES created.


Table QUARTERLY_REGIONAL_SALES analyzed.


1 row inserted.


1 row inserted.


1 row inserted.


1 row inserted.
```



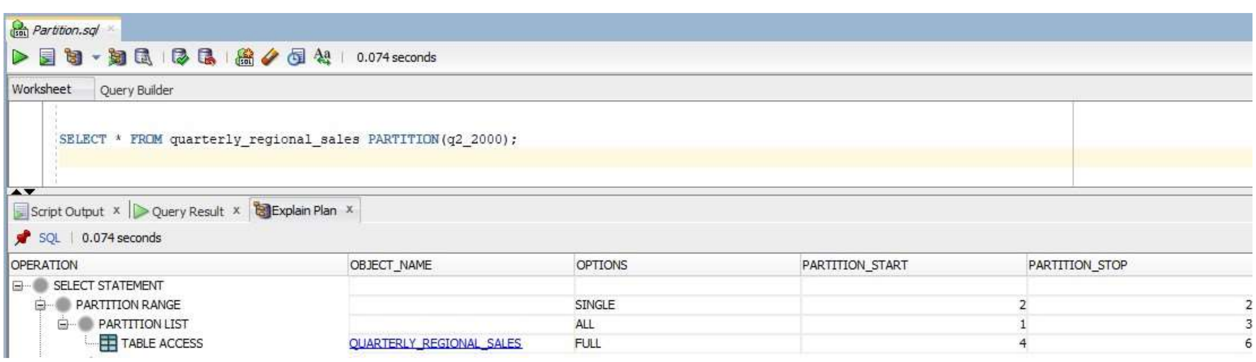

```sql
COMMIT;

SELECT * FROM quarterly_regional_sales;
```

	DEPTNO	ITEM_NO	TXN_DATE	TXN_AMOUNT	STATE
1	1001	100	10-FEB-00	150	TX
2	1002	110	15-JUN-00	100	FL
3	1001	100	20-AUG-00	500	CA
4	1002	110	15-NOV-00	1000	NY



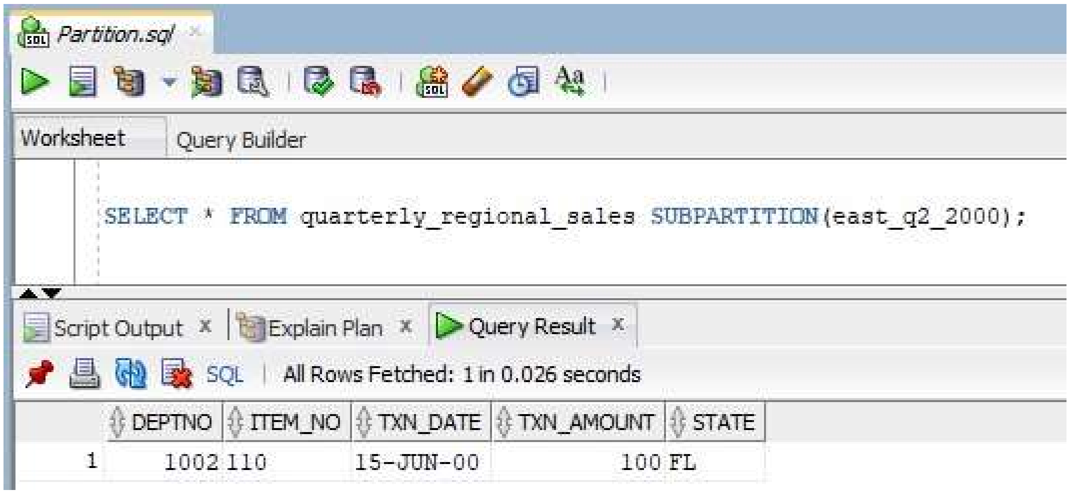

```sql
SELECT * FROM quarterly_regional_sales PARTITION(q2_2000);
```

	DEPTNO	ITEM_NO	TXN_DATE	TXN_AMOUNT	STATE
1	1002	110	15-JUN-00	100	FL




```sql
SELECT * FROM quarterly_regional_sales PARTITION(q2_2000);
```

OPERATION	OBJECT_NAME	OPTIONS	PARTITION_START	PARTITION_STOP
SELECT STATEMENT				
PARTITION RANGE		SINGLE	2	2
PARTITION LIST		ALL	1	3
TABLE ACCESS	QUARTERLY_REGIONAL_SALES	FULL	4	6


Partition.sql
Worksheet
Query Builder
SELECT * FROM quarterly_regional_sales SUBPARTITION(east_q2_2000);
Script Output
Explain Plan
Query Result
SQL
All Rows Fetched: 1 in 0.026 seconds
DEPTNO
ITEM_NO
TXN_DATE
TXN_AMOUNT
STATE
1
1002
110
15-JUN-00
100
FL


Partition.sql
0.064 seconds
Worksheet
Query Builder
SELECT * FROM quarterly_regional_sales SUBPARTITION(east_q2_2000);
Script Output
Query Result
Explain Plan
SQL
0.064 seconds
OPERATION
OBJECT_NAME
OPTIONS
PARTITION_START
PARTITION_STOP
SELECT STATEMENT
PARTITION COMBINED
ITERATOR
KEY
KEY
TABLE ACCESS
QUARTERLY_REGIONAL_SALES
FULL
4
4

Idea 3: Parallelism in Data Warehouses

PS: for convenience all below queries and execution plans are taken from SQL Server

Here is simple query that will demonstrate how parallelism works.
Using a join to combine the results of two entities to enable dashboard to display the results of entities.
First query without using parallelism.

```sql
SELECT 1 AS EntityID,'xxx' AS Entity,load_log_id,load_type,load_finish_date,rows_inserted,rows_updated,rows_deleted,load_status,
load_start_date,DATEPART(dw, load_start_date) AS DayOfWK,
DATEPART(HOUR, load_start_date) AS Hr,DATEDIFF(MINUTE, load_start_date, load_finish_date) AS MN,CAST(load_start_date AS DATE) AS LoadStartDt
FROM dbo.load_log
UNION
SELECT 2 AS EntityID,'yyy' AS Entity,load_log_id,load_type,load_finish_date,rows_inserted,rows_updated,rows_deleted,load_status,load_start_date,
DATEPART(dw, load_start_date) AS DayOfWK,DATEPART(HOUR, load_start_date) AS Hr,DATEDIFF(MINUTE, load_start_date, load_finish_date) AS MN,
CAST(load_start_date AS DATE) AS LoadStartDt
FROM PCC_RDB_NSP.dbo.load_log ORDER BY load_start_date DESC
```

Fig 1.1

Resulting execution plan

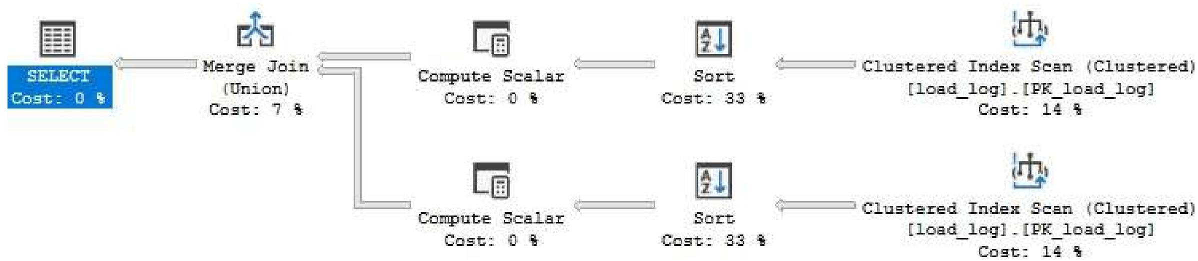


Fig 1.2

As you can see there is no parallelism is seen in the execution plan. The query took 4 secs to get results

00:00:04

Now, forcing the same query to use parallelism.

```sql
SELECT 1 AS EntityID,'xxx' AS Entity,load_log_id,load_type,load_finish_date,rows_inserted,rows_updated,rows_deleted,load_status,
load_start_date,DATEPART(dw, load_start_date) AS DayOfWK,
DATEPART(HOUR, load_start_date) AS Hr,DATEDIFF(MINUTE, load_start_date, load_finish_date) AS MN,CAST(load_start_date AS DATE) AS LoadStartD
FROM dbo.load_log
UNION
SELECT 2 AS EntityID,'yyy' AS Entity,load_log_id,load_type,load_finish_date,rows_inserted,rows_updated,rows_deleted,load_status,load_start_
DATEPART(dw, load_start_date) AS DayOfWK,DATEPART(HOUR, load_start_date) AS Hr,DATEDIFF(MINUTE, load_start_date, load_finish_date) AS MN,
CAST(load_start_date AS DATE) AS LoadStartDt
FROM [redacted].dbo.load_log ORDER BY load_start_date DESC
OPTION(USE HINT('ENABLE_PARALLEL_PLAN_PREFERENCE'))
```

Fig 1.3

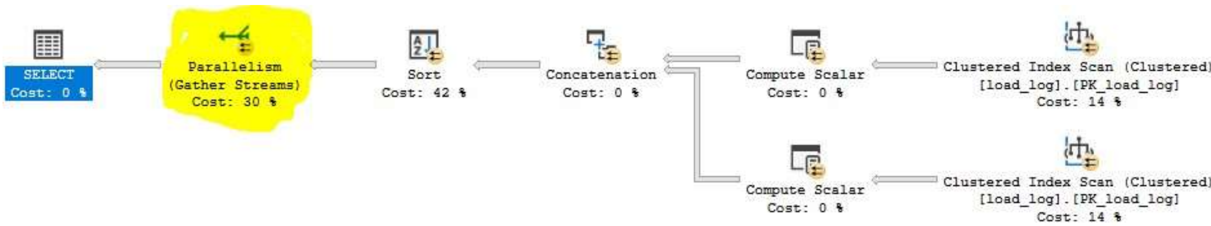


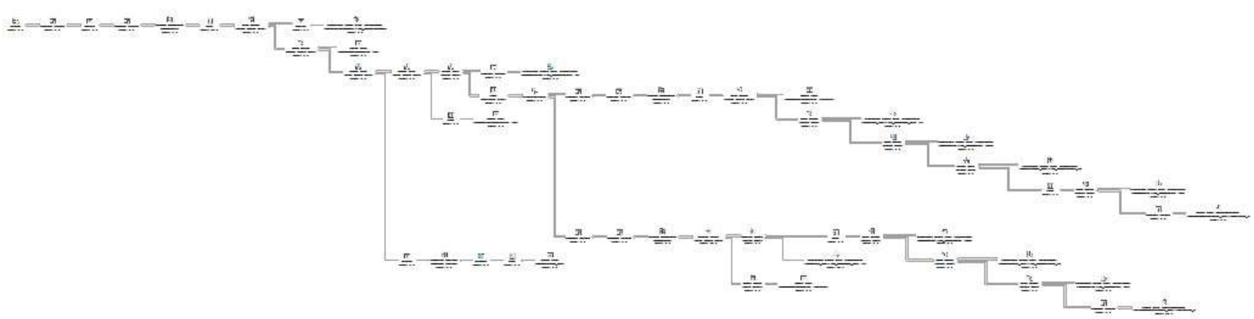
Fig 1.4

As you see the change in execution plan, parallelism made the query to execute in different way making the query to return results little faster.

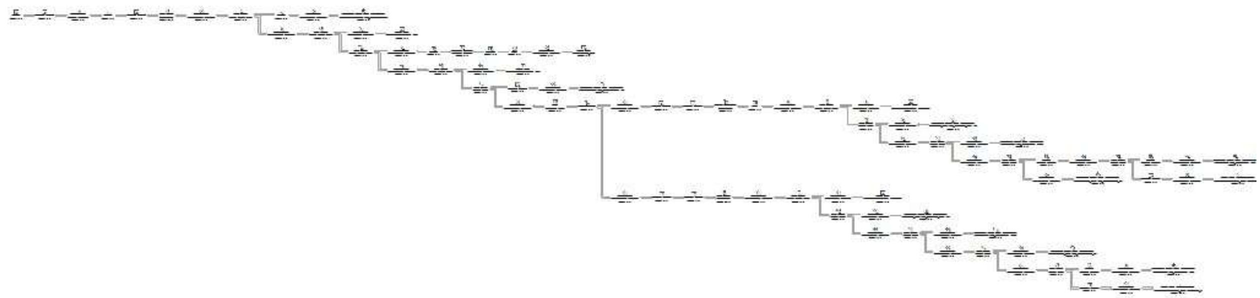
00:00:03

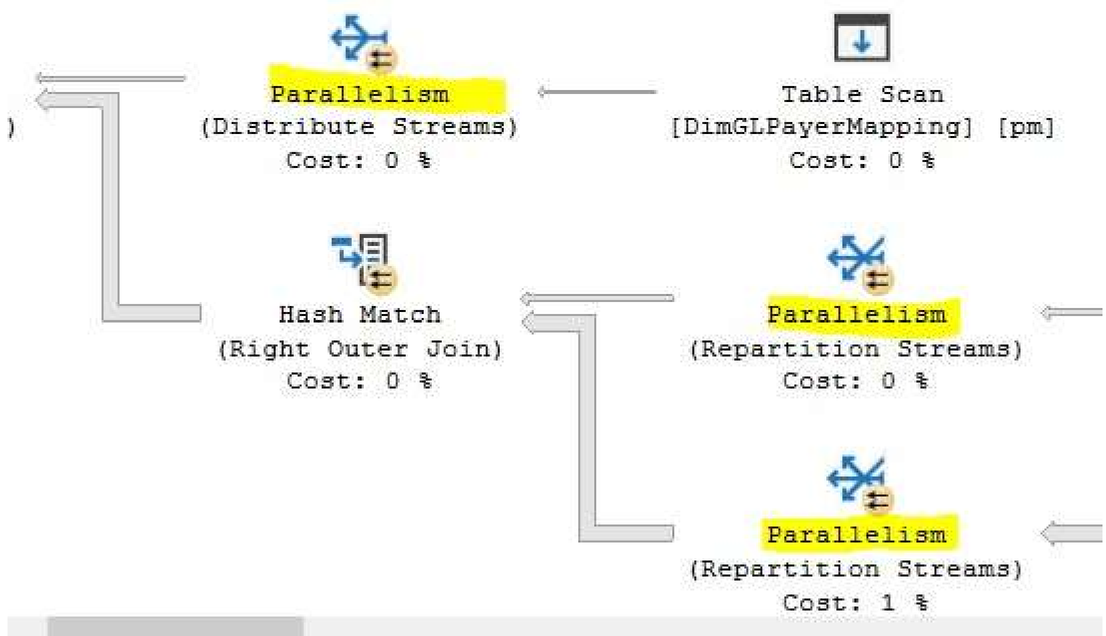
2. Another example of using parallelism

```
SELECT OpUnitID,OpsFacilityDescDisplay,PayerID,PeriodID,Payer,CareLevel,
case when len(CareLevel)<4 then NULL else LEFT(CareLevel, 1) end AS HIPPS1,
case when Len(CareLevel)<4 then null else RIGHT(LEFT(CareLevel, 2), 1)  end AS HIPPS2,
case when Len(careLevel)<4 then null else RIGHT(LEFT(CareLevel, 3), 1) end  AS HIPPS3,
case when Len(careLevel)<4 then null else RIGHT(LEFT(CareLevel, 4), 1) end AS HIPPS4,
AccountNumber,SubAccntNumber,ReportGroup,Days,TotalTransactions as Revenue,ResidentID,ServiceFromDate,ServiceToDate,
SummaryGroup1,SummaryGroup2,GLPayer,PMID
FROM vw_RevenueReconcilation
where PostedPeriodDate>='1/1/2019'
```



```
SELECT OpUnitID,OpsFacilityDescDisplay,PayerID,PeriodID,Payer,CareLevel,
case when len(CareLevel)<4 then NULL else LEFT(CareLevel, 1) end AS HIPPS1,
case when Len(CareLevel)<4 then null else RIGHT(LEFT(CareLevel, 2), 1)  end AS HIPPS2,
case when Len(careLevel)<4 then null else RIGHT(LEFT(CareLevel, 3), 1) end  AS HIPPS3,
case when Len(careLevel)<4 then null else RIGHT(LEFT(CareLevel, 4), 1) end AS HIPPS4,
AccountNumber,SubAccntNumber,ReportGroup,Days,TotalTransactions as Revenue,ResidentID,ServiceFromDate,ServiceToDate,
SummaryGroup1,SummaryGroup2,GLPayer,PMID
FROM vw_RevenueReconcilation
where PostedPeriodDate>='1/1/2019'
 OPTION(USE HINT('ENABLE_PARALLEL_PLAN_PREFERENCE'))
```





As you see above how parallelism makes a difference in the way the database engine executes it.

3.

```sql
 SELECT top 1 s.Date,r.eff_date_from [FROM],r.eff_date_to [THROUGH],DATEDIFF(dd, r.eff_date_from, r.eff_date_to) AS [Days],
[vpd_days],[medicare_days],OpUnitID,s.ResidentID,s.EntityID,PrimaryPayerID,r.pt_rate AS PT,ot_rate AS OT,slp_rate AS SLP,
ncm_rate AS nonCMI,nta_rate AS NTA,nursing_rate AS NUR,CareLevelCode + RugsModifier AS HIPPS,LEFT(CareLevelCode, 1) AS HIPPS_P1,
hp1.PaymentGroup PT_OT_Group,RIGHT(LEFT(CareLevelCode, 2), 1) AS HIPPS_P2,hp2.PaymentGroup SLP_Group,RIGHT(LEFT(CareLevelCode, 3), 1)
hp3.PaymentGroup Nursing_Group,RIGHT(LEFT(CareLevelCode, 4), 1) AS HIPPS_P4,hp4.PaymentGroup NTA_Group,(RugsModifier) AS HIPPS_P5,
hp5.PaymentGroup AI_Group
,LEFT(s.CareLevelCode, 1) as HIPPS1
,RIGHT(LEFT(s.CareLevelCode, 2), 1) as HIPPS2
,RIGHT(LEFT(s.CareLevelCode, 3), 1) as HIPPS3
,RIGHT(LEFT(s.CareLevelCode, 4), 1) as HIPPS4
, s.RugsModifier as RugsModifier
FROM FactResidentStatus_PCC s
INNER JOIN FactARRates r ON r.ResidentID = s.ResidentID AND s.Date BETWEEN eff_date_from AND eff_date_to
INNER JOIN [dbo].[HIPPSTranslation] hp1 ON hp1.HIPPSCharacter = LEFT(s.CareLevelCode, 1) AND hp1.Position = 1
INNER JOIN [dbo].[HIPPSTranslation] hp2 ON hp2.HIPPSCharacter = RIGHT(LEFT(s.CareLevelCode, 2), 1) AND hp2.Position = 2
INNER JOIN [dbo].[HIPPSTranslation] hp3 ON hp3.HIPPSCharacter = RIGHT(LEFT(s.CareLevelCode, 3), 1) AND hp3.Position = 3
INNER JOIN [dbo].[HIPPSTranslation] hp4 ON hp4.HIPPSCharacter = RIGHT(LEFT(s.CareLevelCode, 4), 1) AND hp4.Position = 4
left outer JOIN [dbo].[HIPPSTranslation] hp5 ON hp5.HIPPSCharacter = s.RugsModifier AND hp5.Position = 5
WHERE Date >= '10/1/2019'
```

Fig 3.1

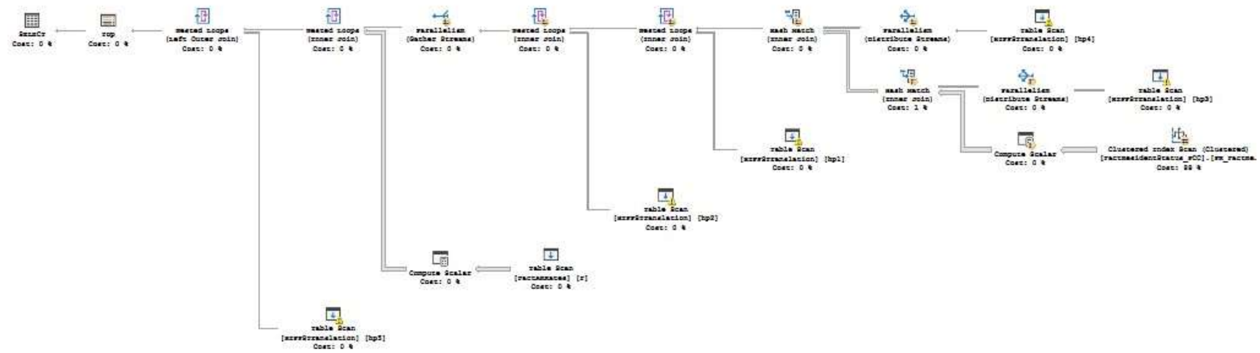


Fig 3.2

Execution without parallelism in figure 3.3

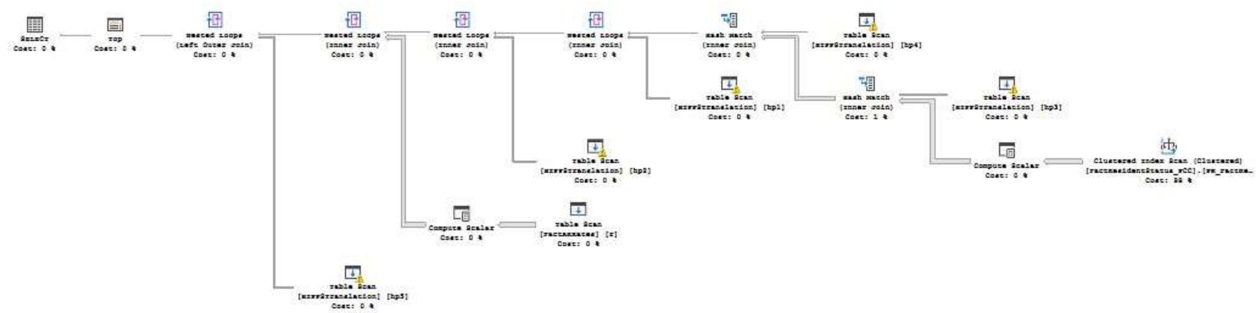
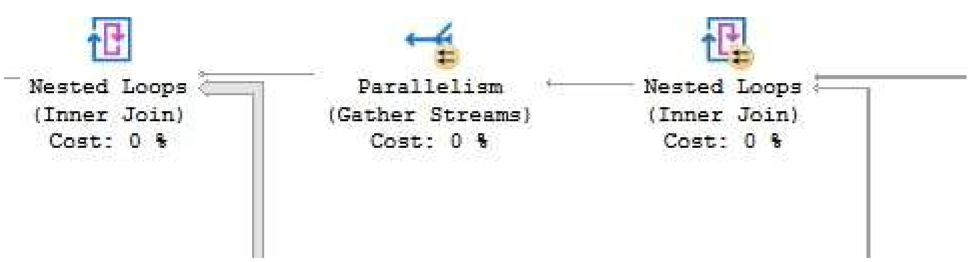


Fig 3.3



Idea 4: Aggregation

The idea behind this administrative task is to create an altogether different table of a group by operation. Such that when that table is called the cost of fetching the data is very low as it is direct fetching. However, When you perform same aggregation group by all the time it is computationally very expensive to perform that operation again and again.

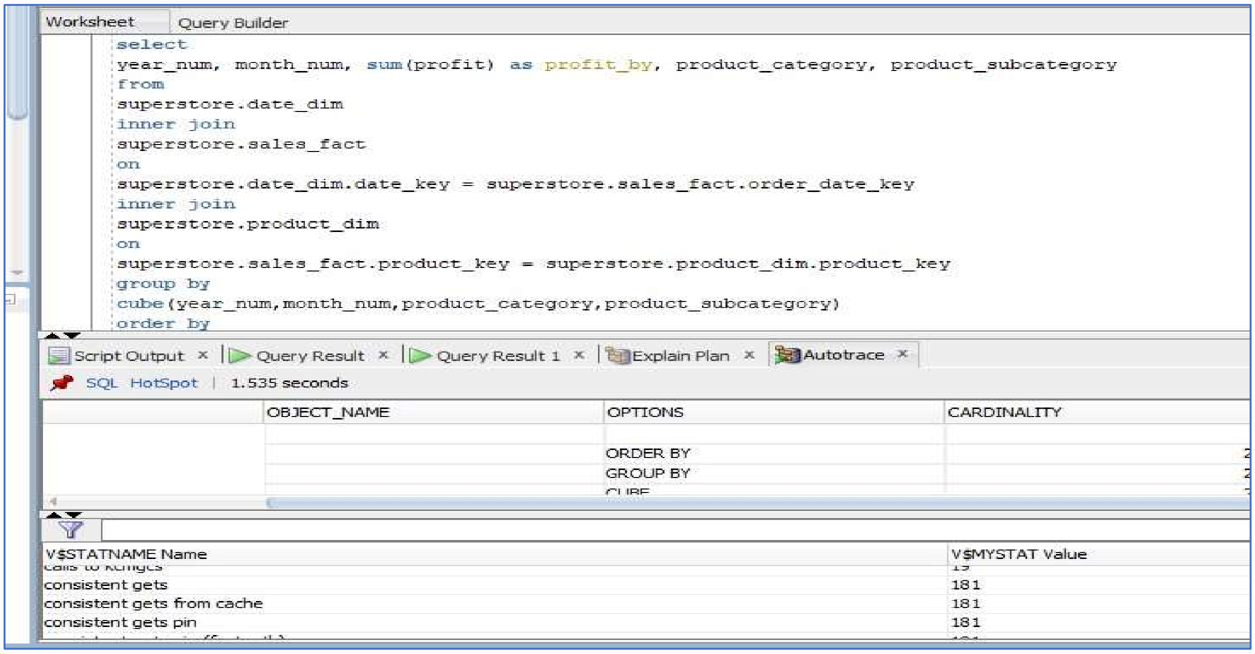
Below is such a demonstration:

The reporting query is to find out year, month a particular product category and product subcategory and their profit.

The Query:

```
SELECT year_num,
        month_num,
       Sum(profit) AS profit_by,
       product_category,
       product_subcategory
FROM   superstore.date_dim
      INNER JOIN superstore.sales_fact
             ON superstore.date_dim.date_key =
                superstore.sales_fact.order_date_key
       INNER JOIN superstore.product_dim
              ON superstore.sales_fact.product_key =
                 superstore.product_dim.product_key
GROUP  BY cube( year_num, month_num, product_category, product_subcate
gory )
ORDER  BY profit_by;
```

Let us see its computation cost



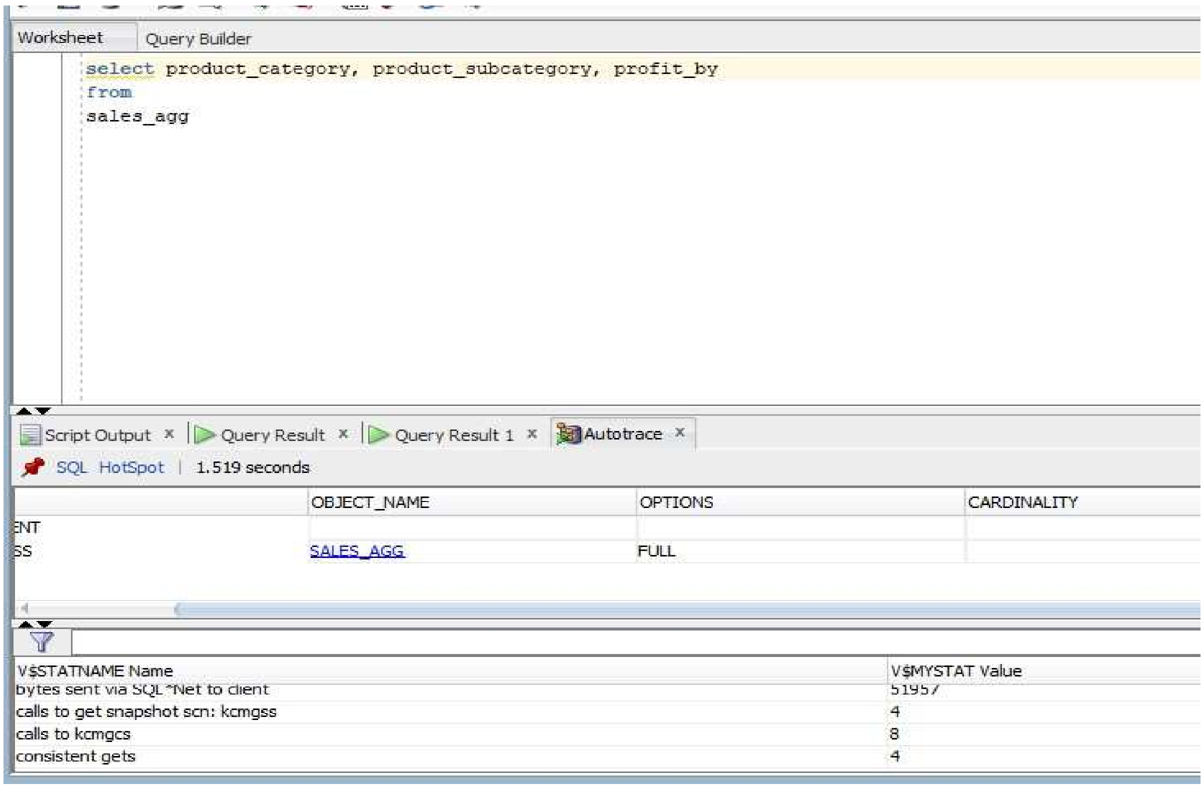
We can see the DB Block reads : db gets +consistent gets = 0 + 181 = 181 With operations like order by group by in execution plan

Now when we create a table for an aggregate query

```
CREATE TABLE sales_agg AS
 SELECT year_num,
        month_num,
        Sum(profit) AS profit_by,
        product_category,
        product_subcategory
 FROM   superstore.date_dim
        INNER JOIN superstore.sales_fact
               ON superstore.date_dim.date_key =
                  superstore.sales_fact.order_date_key
        INNER JOIN superstore.product_dim
               ON superstore.sales_fact.product_key =
                  superstore.product_dim.product_key
 GROUP  BY cube( year_num, month_num, product_category, product_subca
 tegory )
 ORDER  BY profit_by;
```

Calling from that aggregate query

```
SELECT product_category,
        product_subcategory,
       profit_by
FROM   sales_agg
```



We can see by the execution plan

DB block reads = DB block gets + consistent gets

DB block reads = 0 + 4 = 4

We can see how DB block gets cost got reduced from 181 to 4

Now we can fetch our details at a very low cost as compared to whole operation done earlier. This shows the advantage of aggregation of a reporting query to another table

Idea 5: Bitmap Indexing

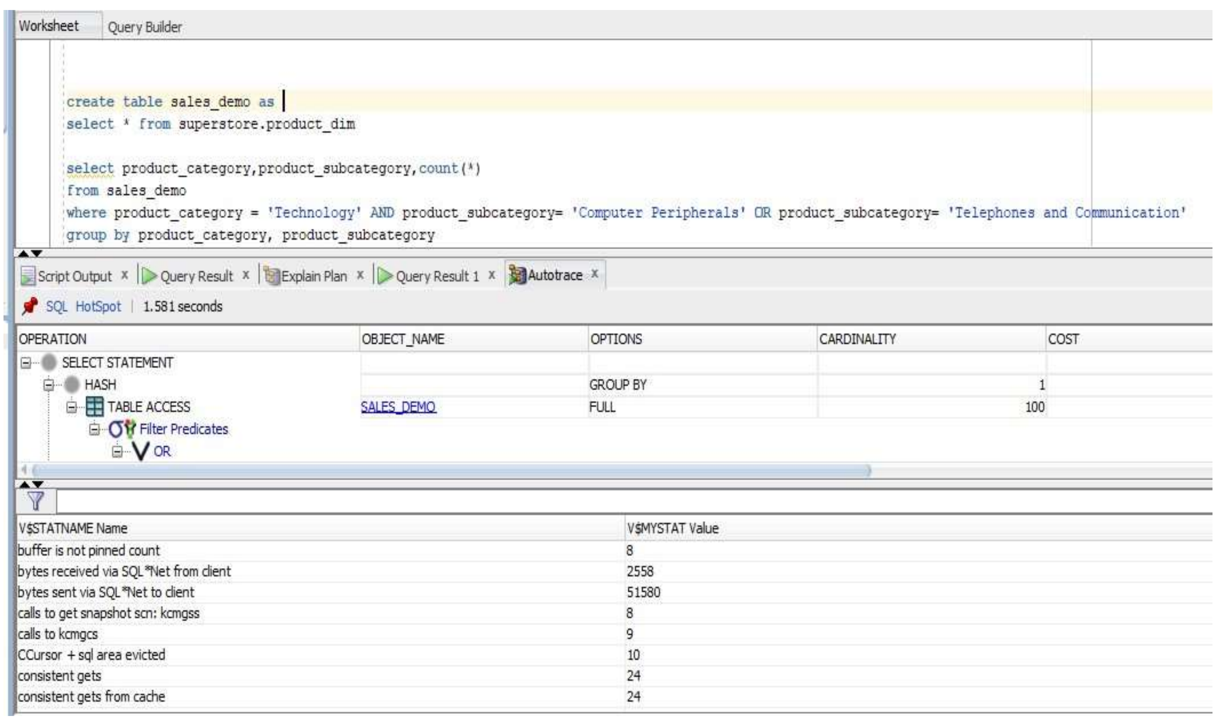
This performance tuning task is to decrease the computation costs where there are ANDs and ORs.

Below is the query which makes use of AND, OR to fetch results.

This query takes out product_category, product_subcategory and their count by grouping product category and product_subcategory.

```
SELECT product_category,
       product_subcategory,
       Count(*)
FROM   sales_demo
WHERE  product_category = 'Technology'
       AND product_subcategory = 'Computer Peripherals'
        OR product_subcategory = 'Telephones and Communication'
GROUP  BY product_category,

          product_subcategory
```

Result without Indexing



We can see higher cost in the execution plan with full join and group by DB block reads(without Indexing) = Db_block_gets + consistent gets i.e DB block reads = 24+0 = 24.

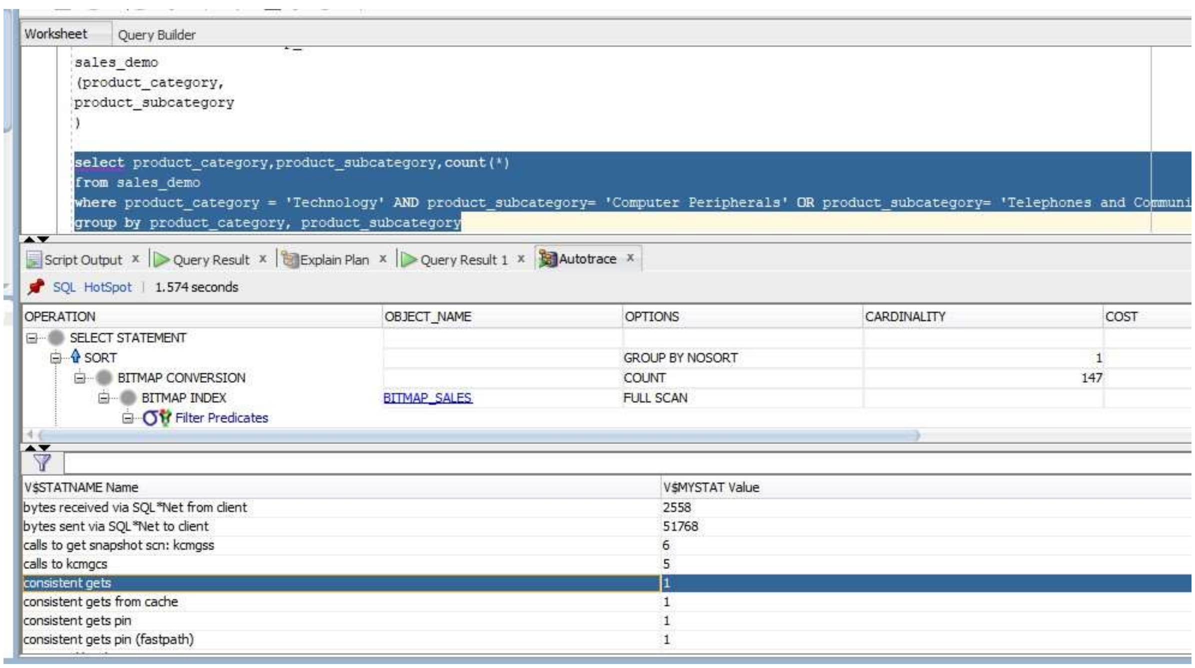
Let us try this with bitmap indexing experiment

Creating bitmap index on product_category and product_subcategory

```
CREATE TABLE sales_demo AS
  SELECT *
  FROM   superstore.product_dim
```

```
CREATE bitmap INDEX bitmap_sales
  ON sales_demo (product_category, product_subcategory)
```

Now Running the Query



We can see the DB block reads get to = 0(DB block gets) + 1(consistent gets) And our Bitmap index in the execution plan by the name of bitmap_sales.

This is the power of bitmap indexing directly reduced the computational cost from 24 to just 1.