Optimizers, TCL, Parallelism, Partioning
Advanced Database Management

Suryateja Chalapati
MS in Business Analytics and Information Systems
University of South Florida
October 26, 2020

Contents

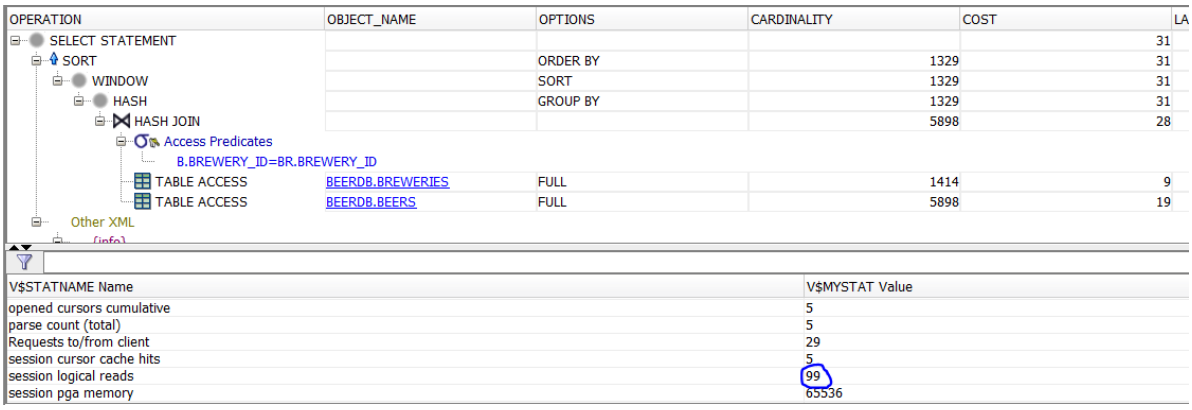
1 Query Processing

1.1 Idea 1: Optimizer Modes: Response Time versus Throughput

In this section, BEERDB will be used to demonstrate the processing of the optimizer. According to the assignment 1, the question 19 will be used to estimate comparing the Throughput and Response Time. The query of that question is the following below

```
SELECT
    b.brewery_id ,
    br.name,
    COUNT(b.brewery_id)  no_of_beer,
    NTILE(4) OVER(ORDER BY COUNT(br.brewery_id) DESC) rank_amount
FROM
    beerdb.beers b
    INNER JOIN beerdb.breweries br
    ON b.brewery_id = br.brewery_id
WHERE
    b.brewery_id IS NOT NULL
GROUP BY
    b.brewery_id,br.name, br.brewery_id
ORDER BY
    rank_amount,no_of_beer DESC, b.brewery_id ;
```

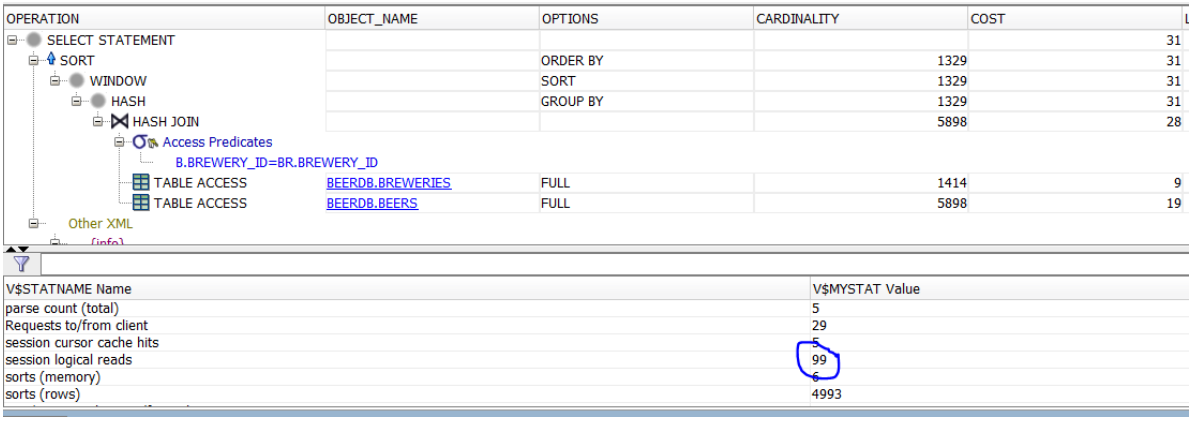
First of all, the optimizer has been altered to ALL_ROWS mode. The result of the explain plan has been shown in the Fig. 1. Then, the FIRST_ROWS_1 and the FIRST_ROWS_100 are shown in Fig. 2 and Fig. 3 respectively.



OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST	LA
SELECT STATEMENT				31	
SORT		ORDER BY	1329	31	
WINDOW		SORT	1329	31	
HASH		GROUP BY	1329	31	
HASH JOIN			5898	28	
Access Predicates					
B.BREWERY_ID=BR.BREWERY_ID					
TABLE ACCESS	BEERDB.BREWERIES	FULL	1414	9	
TABLE ACCESS	BEERDB.BEERS	FULL	5898	19	
Other XML					

V$STATNAME Name	V$MYSTAT Value
opened cursors cumulative	5
parse count (total)	5
Requests to/from client	29
session cursor cache hits	5
session logical reads	99
session pga memory	65536

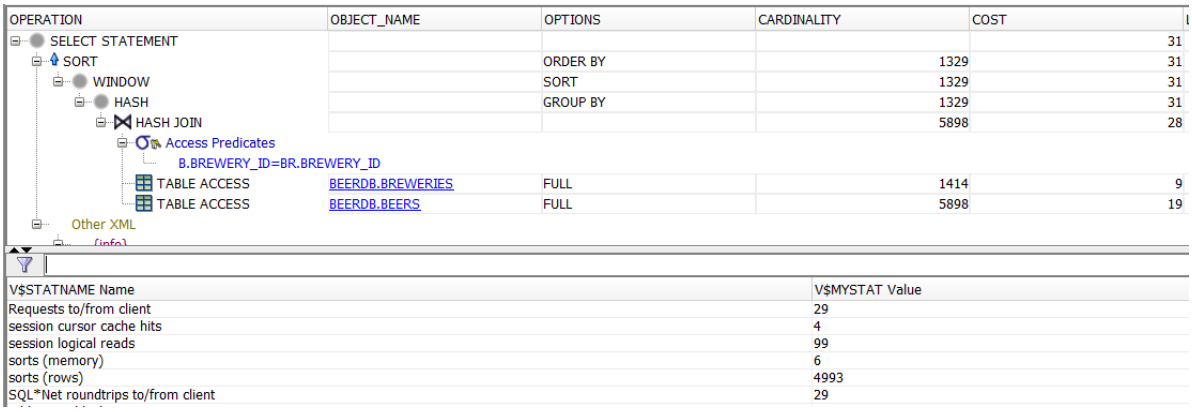
Figure 1: ALL_ROWS



OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT				31
SORT		ORDER BY	1329	31
WINDOW		SORT	1329	31
HASH		GROUP BY	1329	31
HASH JOIN			5898	28
Access Predicates				
B.BREWERY_ID=BR.BREWERY_ID				
TABLE ACCESS	BEERDB.BREWERIES	FULL	1414	9
TABLE ACCESS	BEERDB.BEERS	FULL	5898	19
Other XML				

V$STATNAME Name	V$MYSTAT Value
parse count (total)	5
Requests to/from client	29
session cursor cache hits	5
session logical reads	99
sorts (memory)	6
sorts (rows)	4993

Figure 2: FIRST_ROWS_1



OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT				31
SORT		ORDER BY	1329	31
WINDOW		SORT	1329	31
HASH		GROUP BY	1329	31
HASH JOIN			5898	28
Access Predicates				
B.BREWERY_ID=BR.BREWERY_ID				
TABLE ACCESS	BEERDB.BREWERIES	FULL	1414	9
TABLE ACCESS	BEERDB.BEERS	FULL	5898	19
Other XML				

V$STATNAME Name	V$MYSTAT Value
Requests to/from client	29
session cursor cache hits	4
session logical reads	99
sorts (memory)	6
sorts (rows)	4993
SQL*Net roundtrips to/from client	29

Figure 3: FIRST_ROWS_100

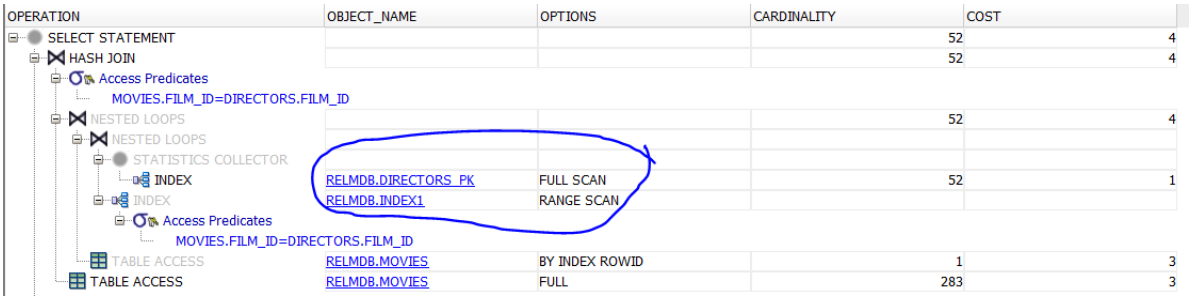
According to the result, there is no different session logic read and cost of the query even the mode is changed. Although the result of changing the optimizer mode could affect the cost of the query shown in class, the optimizer might not have an influence in certain query based on database structure.

1.2 Idea 2: Investigating Selectivity

This section will investigate the result of indexing when the optimizer mode is changed. Using the assignment 3 reference in relmdb database, the query of the example is the following below:

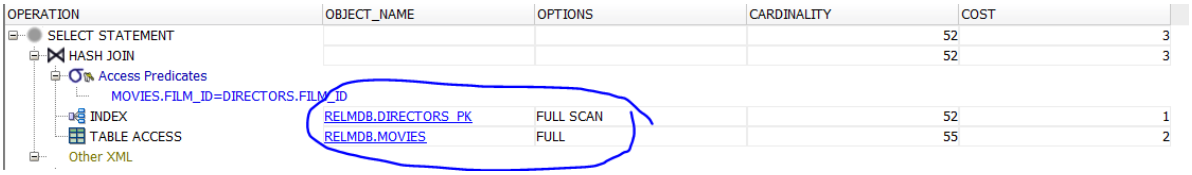
```
1 SELECT
2     film_title,
3     director
4 FROM relmdb.movies
5     INNER JOIN relmdb.directors
6     USING (film_id);
```

The result of ALL_ROWS and FIRST_ROWS_10 has been shown in Fig.4 and Fig.5 respectively. The Thoughput method uses full scan and range scan to deliver the result while the response time conduct only full scan.



OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			52	4
HASH JOIN			52	4
Access Predicates				
MOVIES.FILM_ID=DIRECTORS.FILM_ID				
NESTED LOOPS			52	4
NESTED LOOPS				
STATISTICS COLLECTOR				
INDEX	RELMDB.DIRECTORS_PK	FULL SCAN	52	1
INDEX	RELMDB.INDEX1	RANGE SCAN		
Access Predicates				
MOVIES.FILM_ID=DIRECTORS.FILM_ID				
TABLE ACCESS	RELMDB.MOVIES	BY INDEX ROWID	1	3
TABLE ACCESS	RELMDB.MOVIES	FULL	283	3

Figure 4: ALL_ROWS



OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			52	3
HASH JOIN			52	3
Access Predicates				
MOVIES.FILM_ID=DIRECTORS.FILM_ID				
INDEX	RELMDB.DIRECTORS_PK	FULL SCAN	52	1
TABLE ACCESS	RELMDB.MOVIES	FULL	55	2
Other XML				

Figure 5: FIRST_ROWS_10

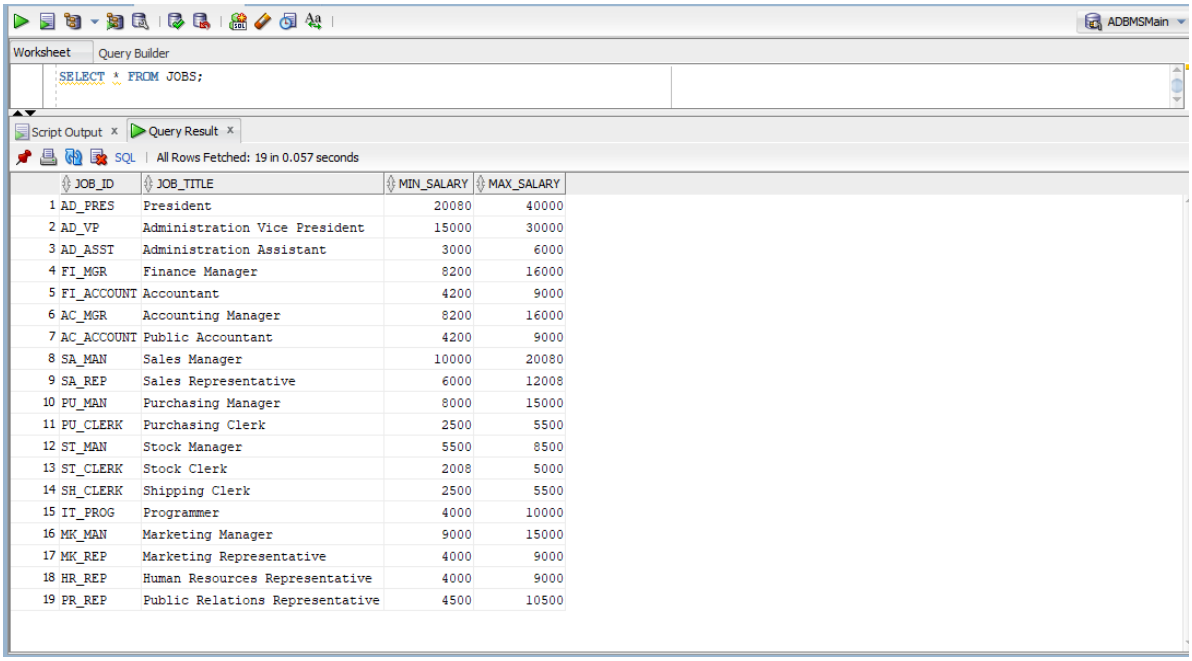
2 Transaction Processing

Transaction is a group of commands that change the stored data in a database. In transaction processing, either all of the commands succeed or none of them are successful. If any of the command at any step fails, the complete process fails and any of the modified data rolls back to the original. Transactions have properties knows as ACID - Atomicity, Consistency, Isolation, Durability. Idea 5 (Transaction Control Language) and Idea 6 (Read Consistency Models) will demonstrate the ACID properties of transaction processing.

2.1 Idea 5: Transaction Control Language

When we use DML statements like INSERT, UPDATE AND DELETE, the changes made by these commands are not stored in the current session immediately until it is closed using COMMIT. If the session is not permanently stored it can be rolled back to the original point using ROLLBACK.

To perform this, JOBS table is selected from HR database. Contents of the table are shown in Fig. 6.

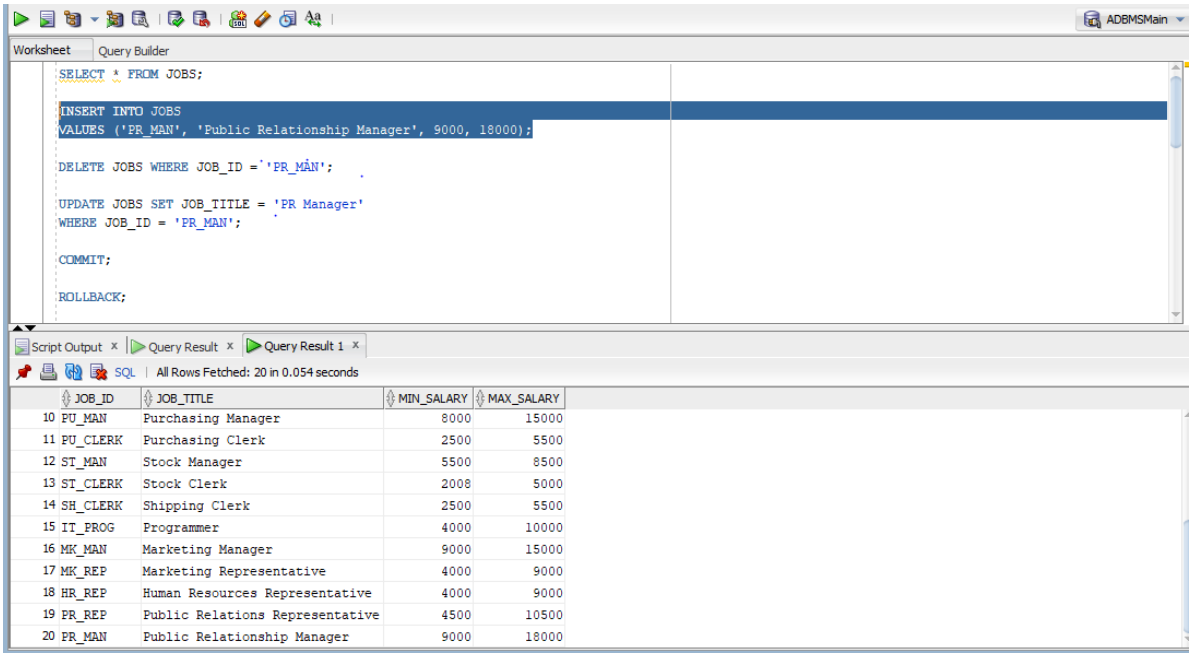


	JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
1	AD_PRES	President	20080	40000
2	AD_VP	Administration Vice President	15000	30000
3	AD_ASST	Administration Assistant	3000	6000
4	FI_MGR	Finance Manager	8200	16000
5	FI_ACCOUNT	Accountant	4200	9000
6	AC_MGR	Accounting Manager	8200	16000
7	AC_ACCOUNT	Public Accountant	4200	9000
8	SA_MAN	Sales Manager	10000	20080
9	SA_REP	Sales Representative	6000	12008
10	PU_MAN	Purchasing Manager	8000	15000
11	PU_CLERK	Purchasing Clerk	2500	5500
12	ST_MAN	Stock Manager	5500	8500
13	ST_CLERK	Stock Clerk	2008	5000
14	SH_CLERK	Shipping Clerk	2500	5500
15	IT_PROG	Programmer	4000	10000
16	MK_MAN	Marketing Manager	9000	15000
17	MK_REP	Marketing Representative	4000	9000
18	HR_REP	Human Resources Representative	4000	9000
19	PR_REP	Public Relations Representative	4500	10500

Figure 6: JOBS Table

2.1.1 Scenario 1: INSERT Command

A row is inserted into the table and can be displayed using SELECT statement as shown in Fig. 7. ROLLBACK undo the change made to the table as shown in Fig. 8. If the change is stored permanently using COMMIT, then it can not be rolled back to the original as shown in Fig. 9.



	JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
10	PU_MAN	Purchasing Manager	8000	15000
11	PU_CLERK	Purchasing Clerk	2500	5500
12	ST_MAN	Stock Manager	5500	8500
13	ST_CLERK	Stock Clerk	2008	5000
14	SH_CLERK	Shipping Clerk	2500	5500
15	IT_PROG	Programmer	4000	10000
16	MK_MAN	Marketing Manager	9000	15000
17	MK_REP	Marketing Representative	4000	9000
18	HR_REP	Human Resources Representative	4000	9000
19	PR_REP	Public Relations Representative	4500	10500
20	PR_MAN	Public Relationship Manager	9000	18000

Figure 7: INSERT Without COMMIT

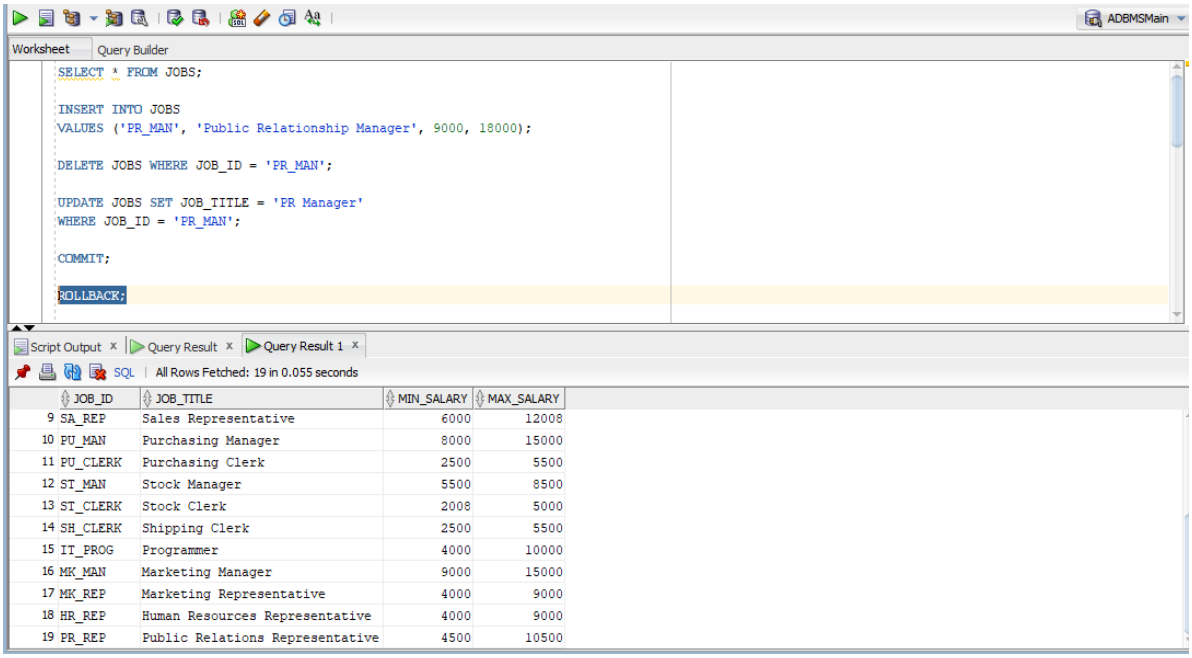


Figure 8: ROLLBACK Before COMMIT

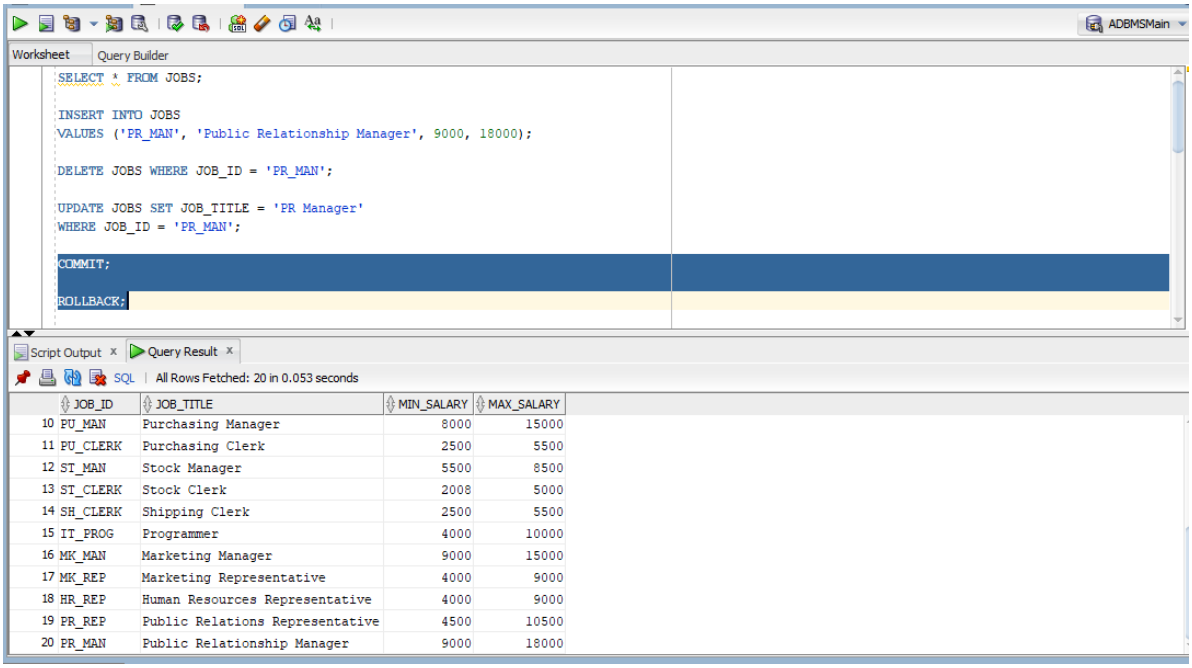


Figure 9: ROLLBACK After COMMIT

2.1.2 Scenario 2: DELETE Command

If inserted row is deleted without COMMIT then it can be rolled back to where the row does not exist as shown in Fig. 10. If the DELETE command is operated on the inserted row which has

been closed using COMMIT, ROLLBACK brings back the permanently stored row as shown in Fig. 11.

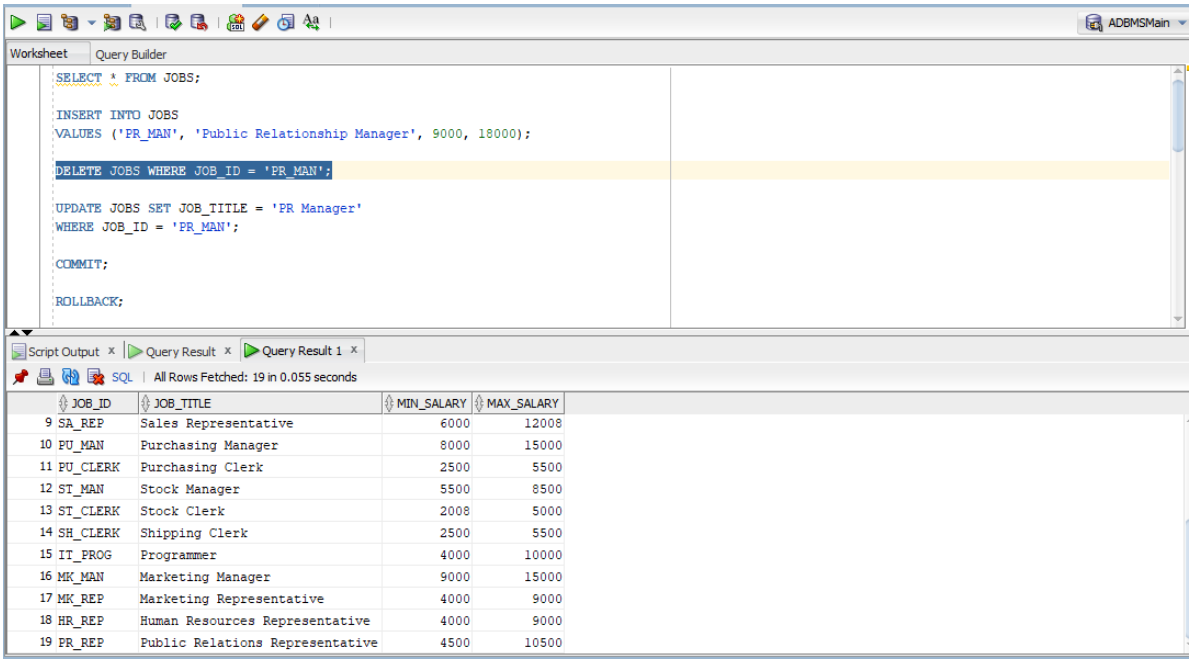


Figure 10: DELETE Before COMMIT

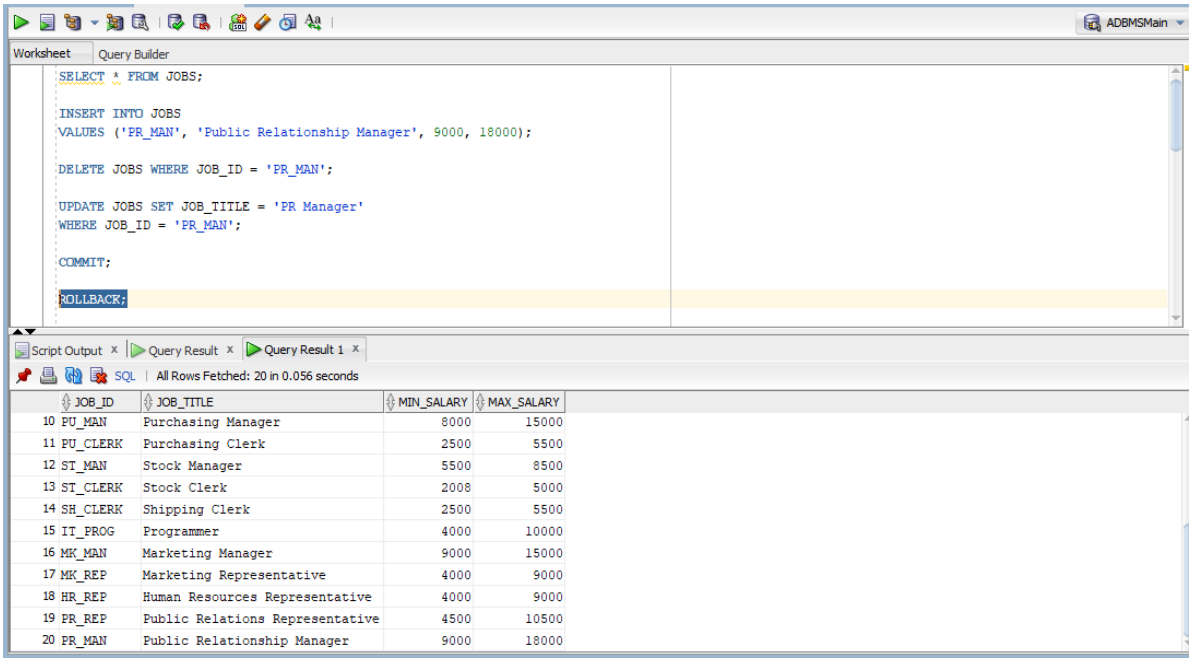


Figure 11: DELETE After COMMIT

2.1.3 Scenario 3: UPDATE Command

Inserted row is updated for the JOB_TITLE. This change can be rolled back to the original inserted row if COMMIT command is not used to permanently store the update as shown in Fig. 12. Once the update is stored using COMMIT, ROLLBACK can not go back to original row as shown in Fig. 13.

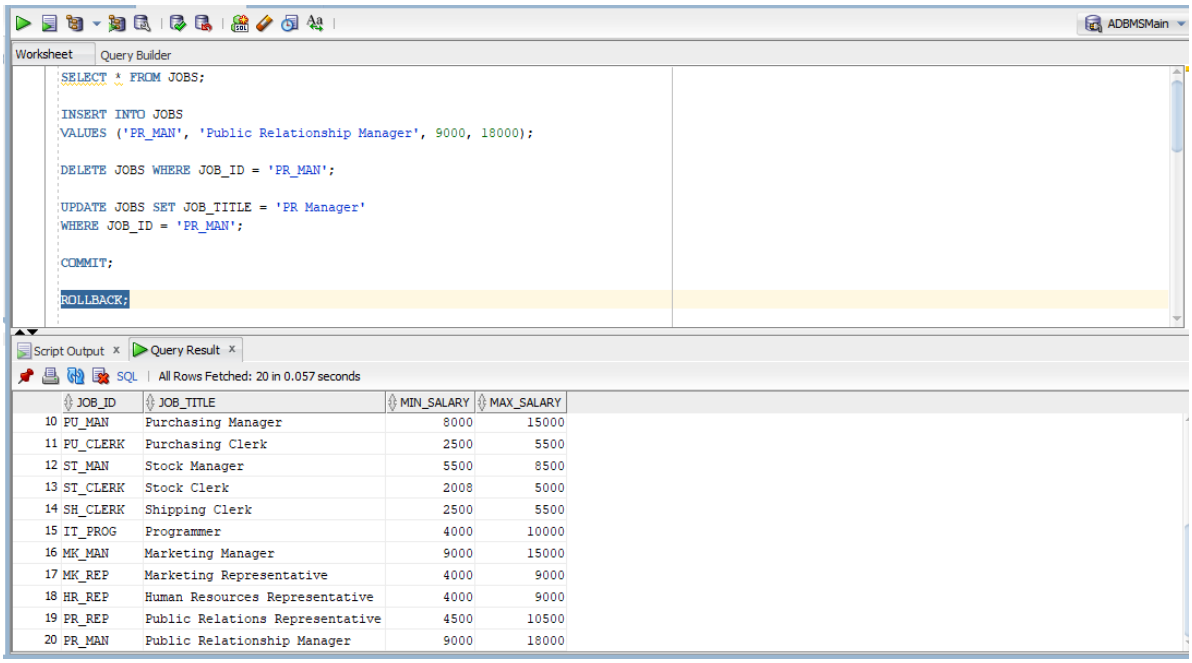


Figure 12: UPDATE Before COMMIT

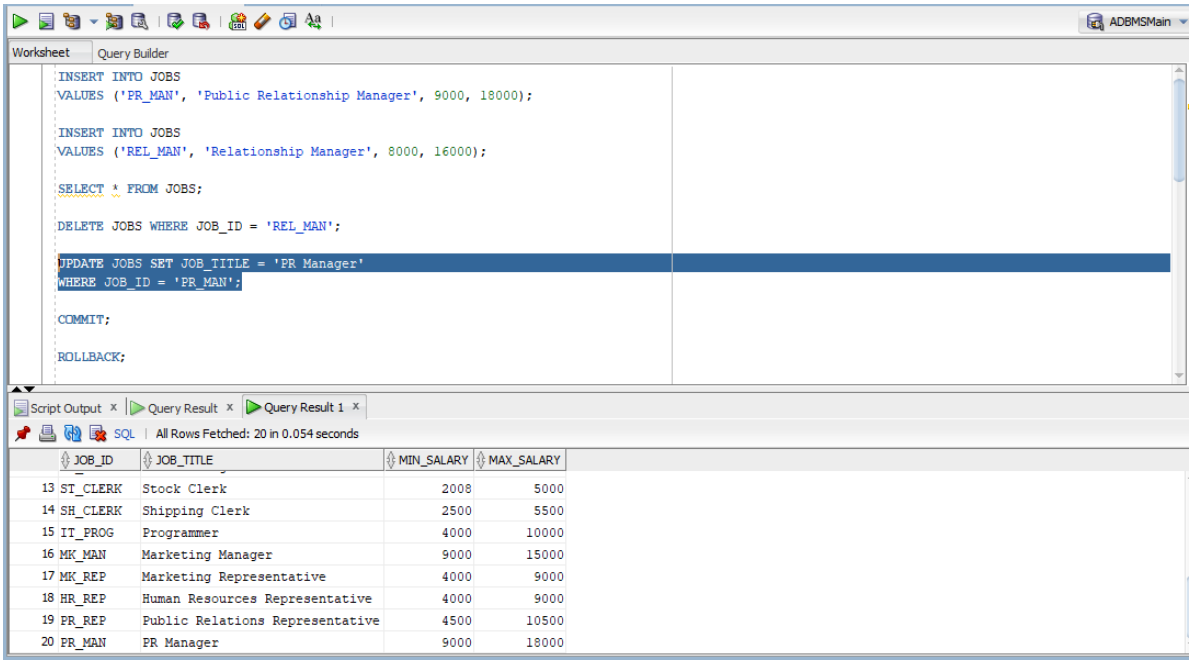


Figure 13: UPDATE After COMMIT

2.2 Idea 6: Read Consistency Models

Dirty read happens when one transaction read the data of another transaction which is not yet committed. In case if the first transaction is rolled back after the second transaction has read the data already, second transaction has dirty read data that does not exist anymore.

For this section, TRADES table is used from MMKRT database.

2.2.1 Transaction 1

If stock quantity for a TRADE_ID is 5. If a transaction is made and the stock quantity is updated to 4. While the billing is made, there is a delay time set before the transaction fails and is rolled back to stock quantity 5 as shown in Fig. 14.

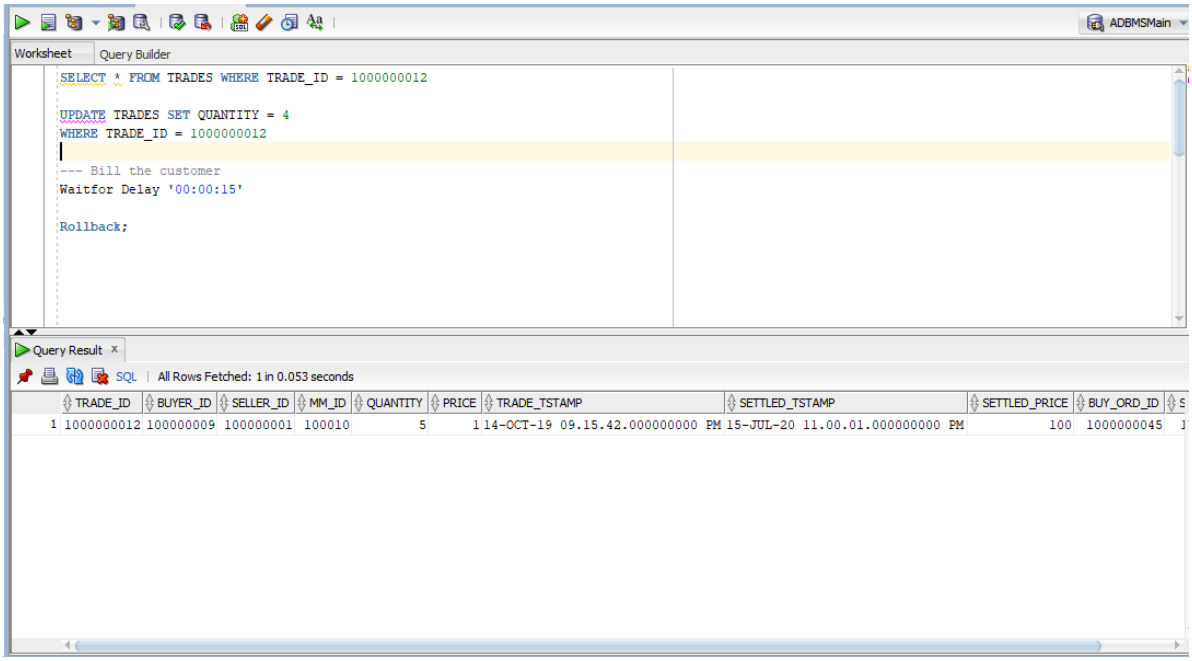


Figure 14: Transaction 1

2.2.2 Transaction 2

If the first transaction is not committed and second transaction is allowed to read uncommitted transaction, it updates stock quantity to 4 even before the first transaction is rolled back to 5as shown in Fig. 15. To maintain read consistency, second transaction is allowed to read committed first transaction so the stock quantity is updated same in both as shown in Fig. 16.

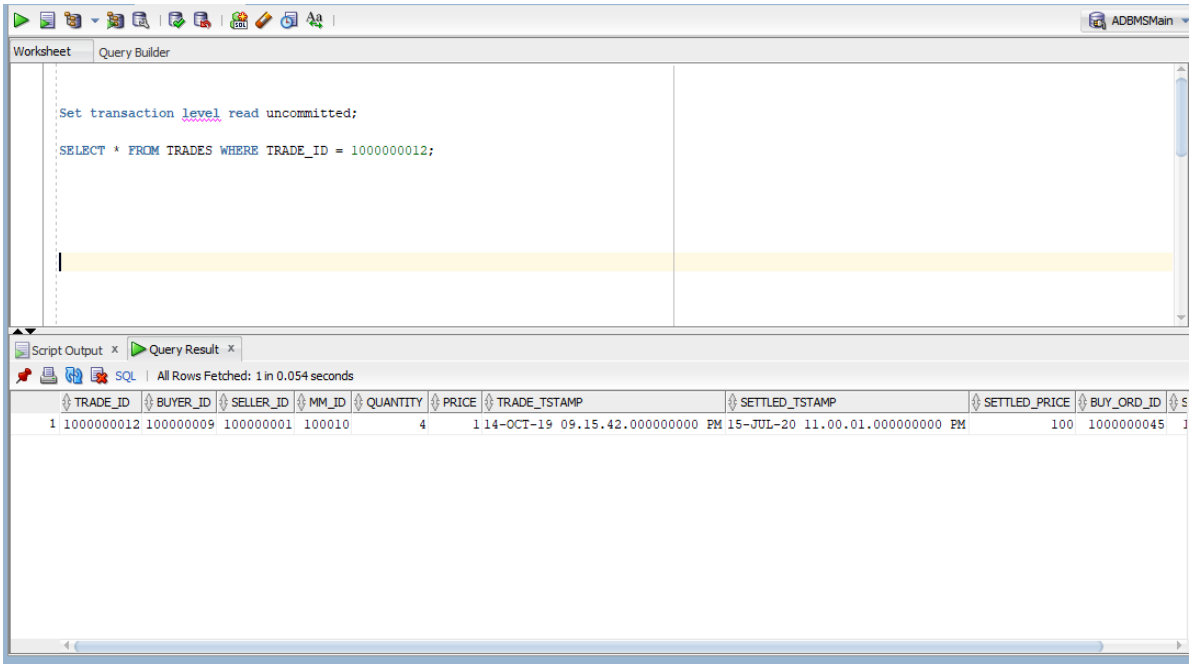


Figure 15: Transaction 2 Uncommitted

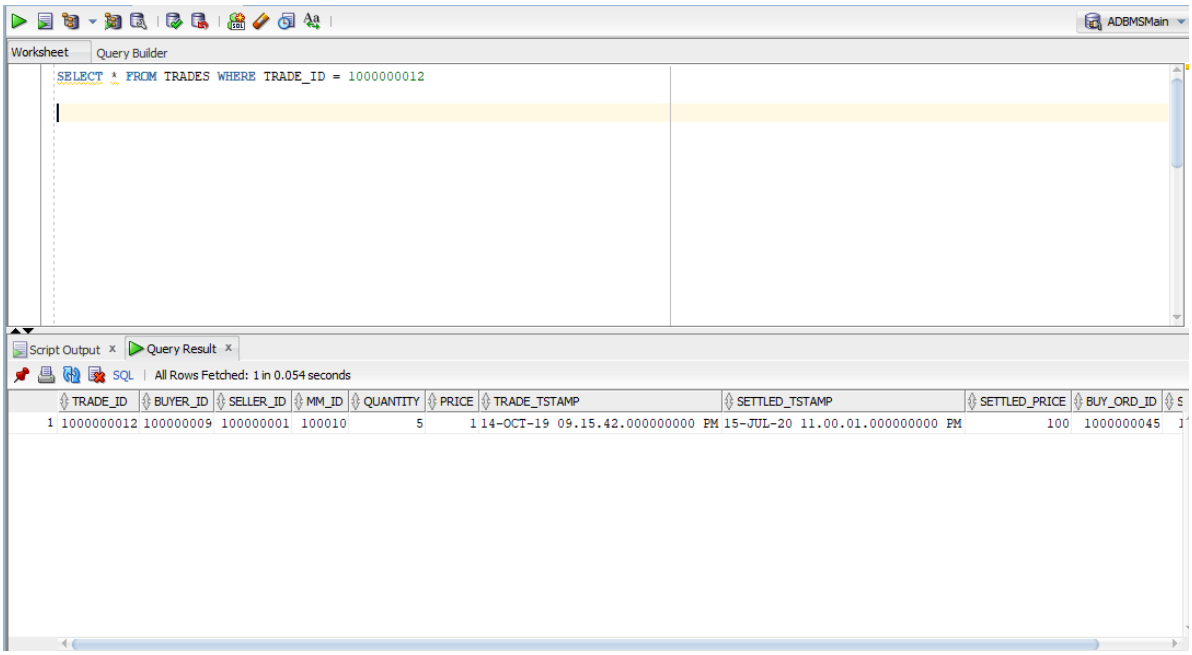


Figure 16: Transaction 2 Committed

3 Parallel Databases

Here we will discuss about parallel databases. We will use multiple databases to show how parallel execution works and write some interesting queries.

3.1 Basic Parallel Execution

In this section, the RELMDB database is used to show how parallelism works. First of all we begin by creating a table and executing a simple query. Then we proceed to enable parallelism on the same query and execute again.

3.2 Step - 1: Creating a Table and Executing a Simple Query

We start by creating a new table from RELMDB.FANS. Then we will run a simple query with a condition. The results in Fig 17 shows the output.

```
1 CREATE TABLE FANS_PARALLEL AS
2 SELECT * FROM RELMDB.FANS;
3
4 SELECT COUNT(*) FROM FANS_PARALLEL WHERE MEMBERSHIP = 'GOLD';
```

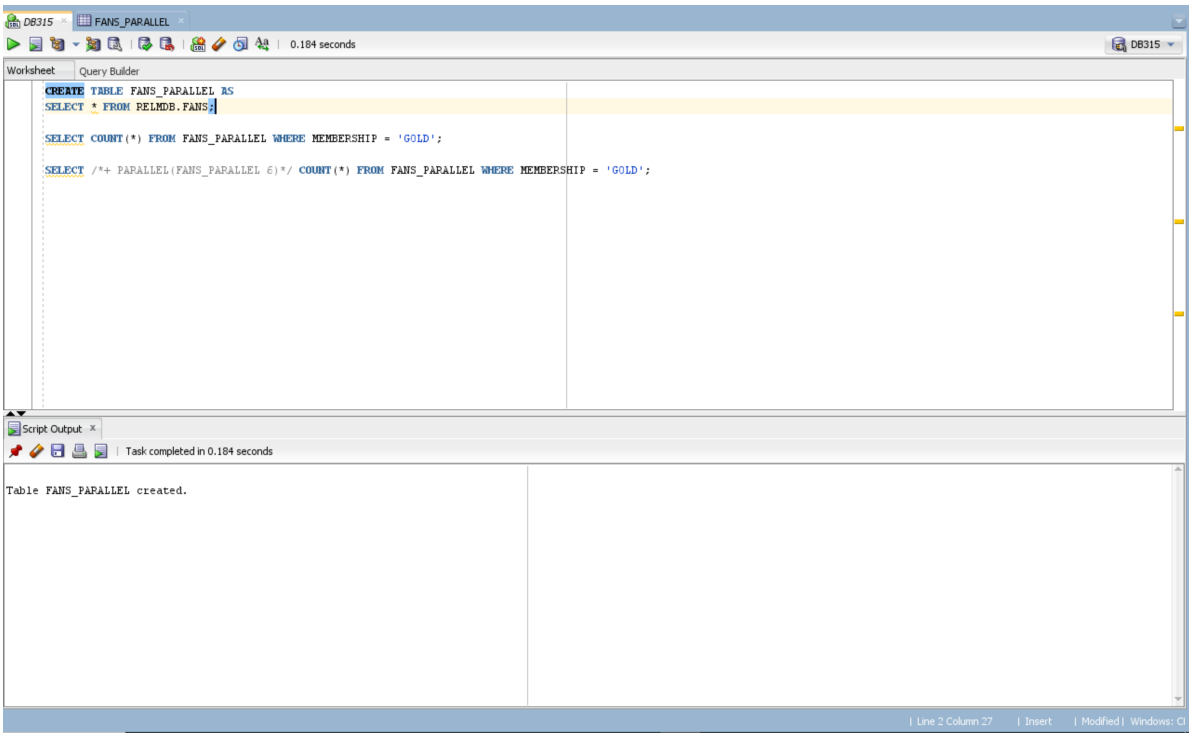


Figure 17: Table Created

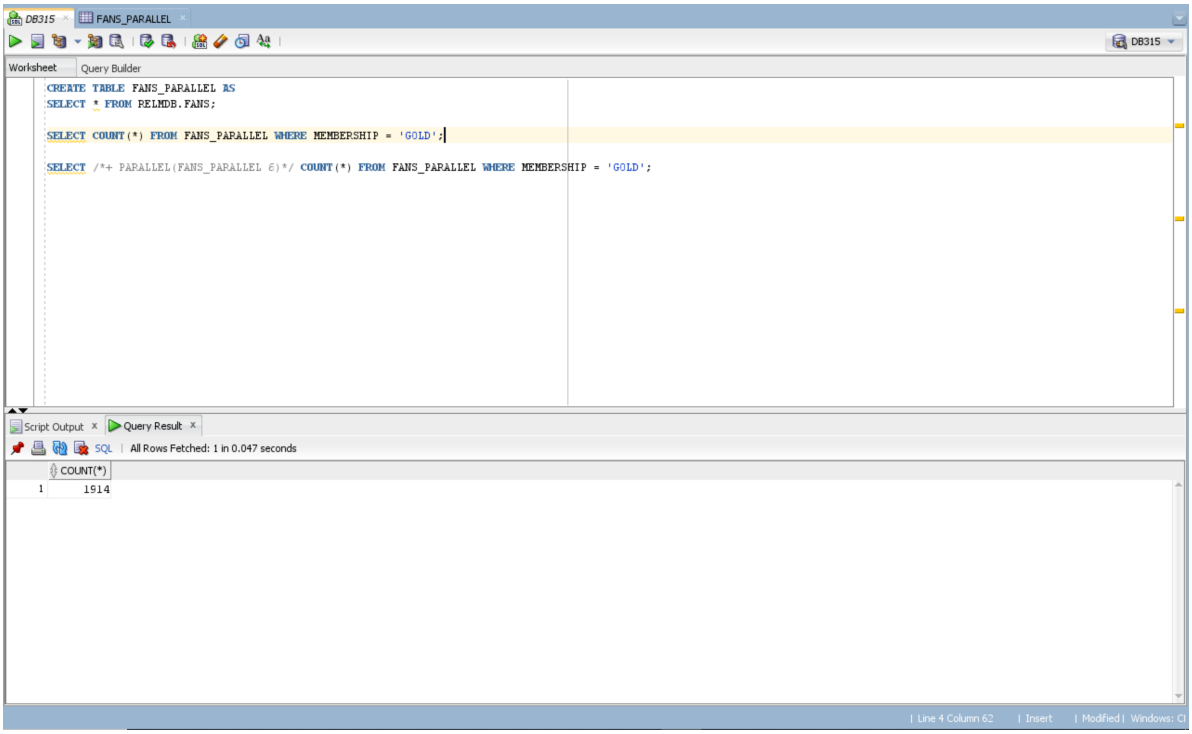


Figure 18: Query Output

3.3 Step - 2: Running the Same Query with Parallelism

Now we enable parallelism and run the same query again. Here we have applied 6 parallel process for this query. When the query is run once again, the Fig 19 shows the output which is 1914 entries.

```
SELECT /*+ PARALLEL(FANS_PARALLEL 6)*/ COUNT(*)
FROM FANS_PARALLEL WHERE MEMBERSHIP = 'GOLD';
```

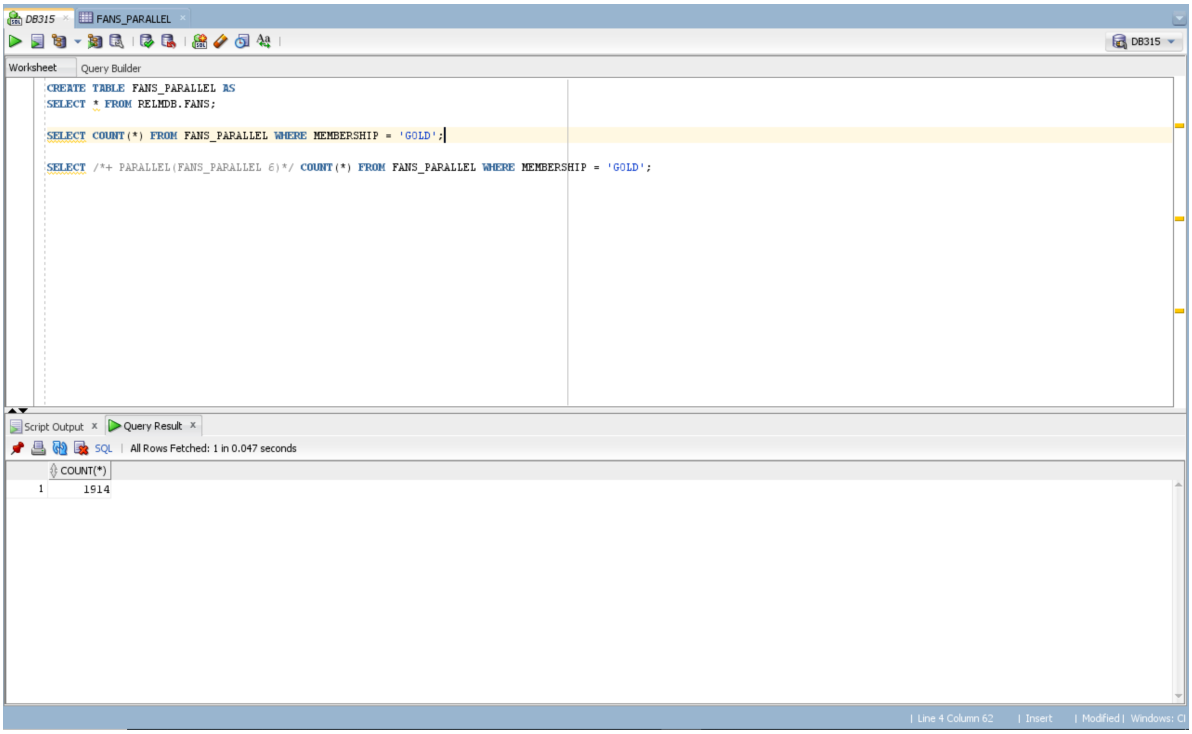


Figure 19: Query Output

3.4 Step - 3: Comparing the Execution Plans

Lastly, lets compare the execution plans for both the scenarios. We can clearly see the (PX) in the execution plan in Fig 21.

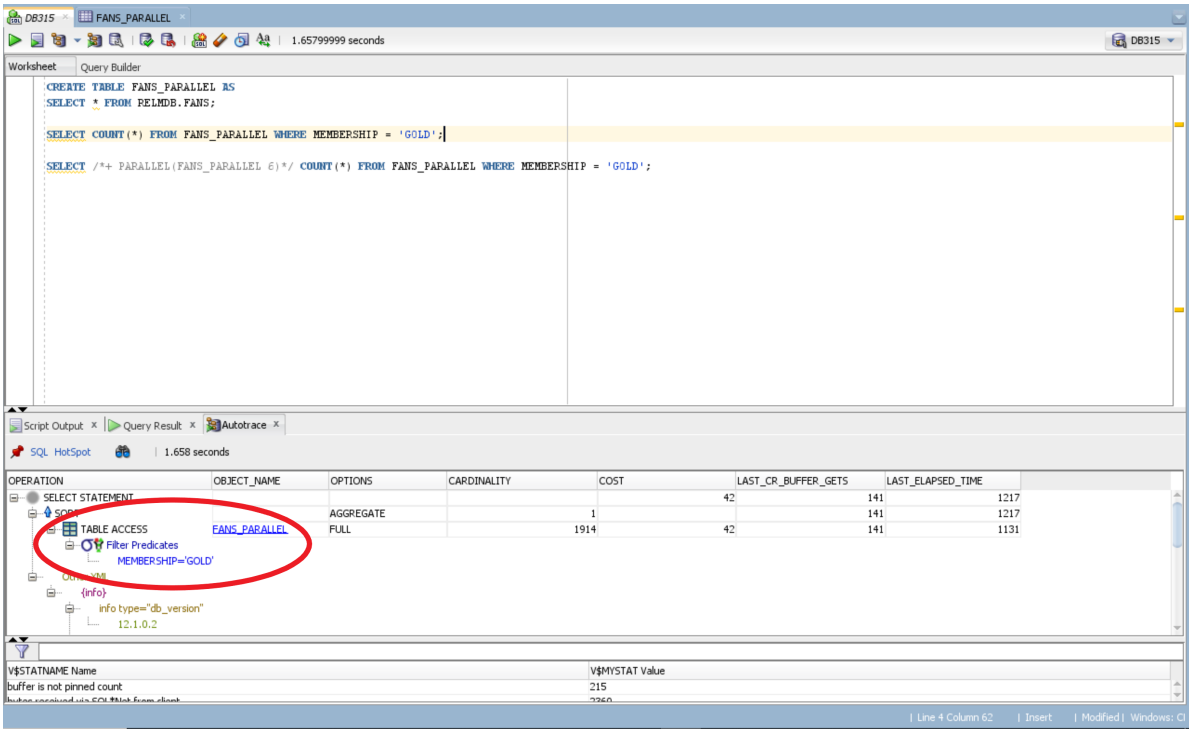


Figure 20: With PX

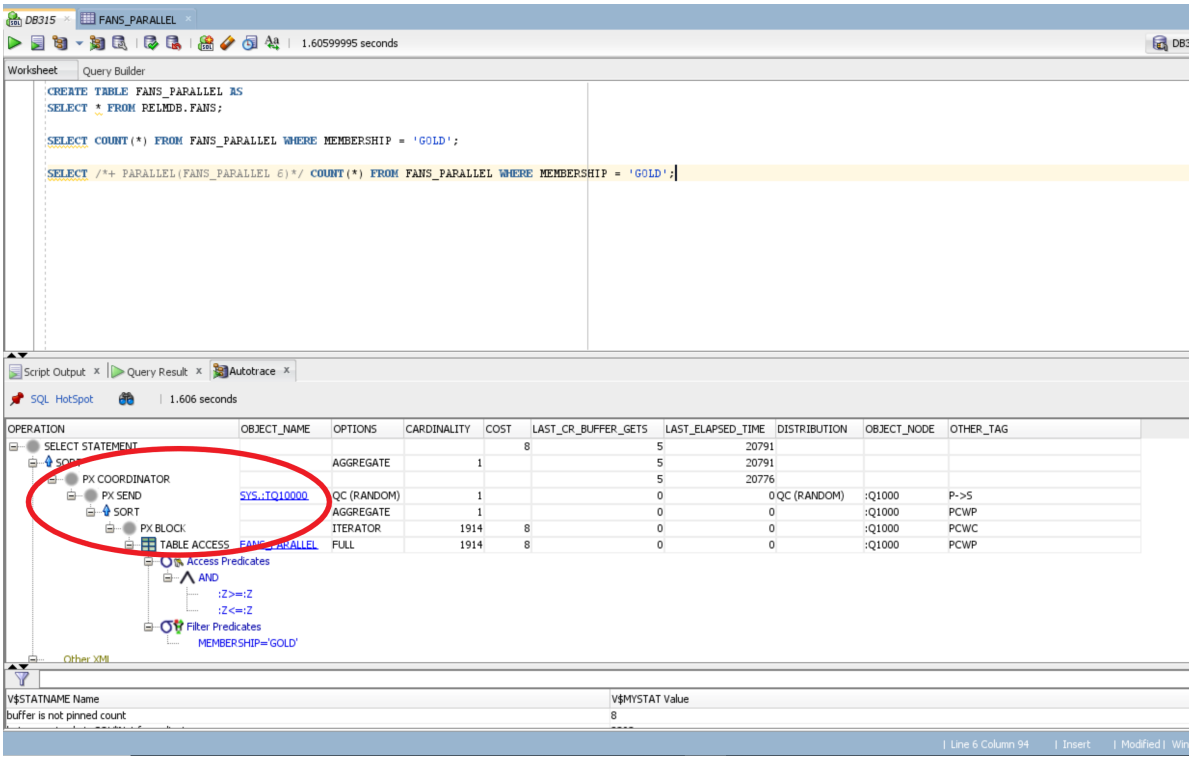


Figure 21: Without PX

3.5 Transitions in More Complex Queries

In this section, the RELMDB database is used again to show various transitions like parallel-to-parallel and parallel-to-serial using complex queries. In this example we see the best transition which is parallel-to-parallel, showing we had no bottlenecks for this processing.

3.6 Step - 1: Creating a New Table

We start by creating a new MOVIE_DIRECTORS table from MOVIES and DIRECTORS table in the RELMDB database. Then we run a complex query by enabling parallelism. The results in Fig 22 shows the output.

```
CREATE TABLE MOVIE_DIRECTORS AS (
    SELECT
        MOVIES.FILM_ID,
        MOVIES.IMDB_RATING,
        MOVIES.FILM_TITLE,
        MOVIES.FILM_YEAR,
        MOVIES.MPAA_RATING,
        DIRECTORS.DIRECTOR
    FROM
        RELMDB.MOVIES,
        RELMDB.DIRECTORS
    WHERE
        MOVIES.FILM_ID = DIRECTORS.FILM_ID
    )
```

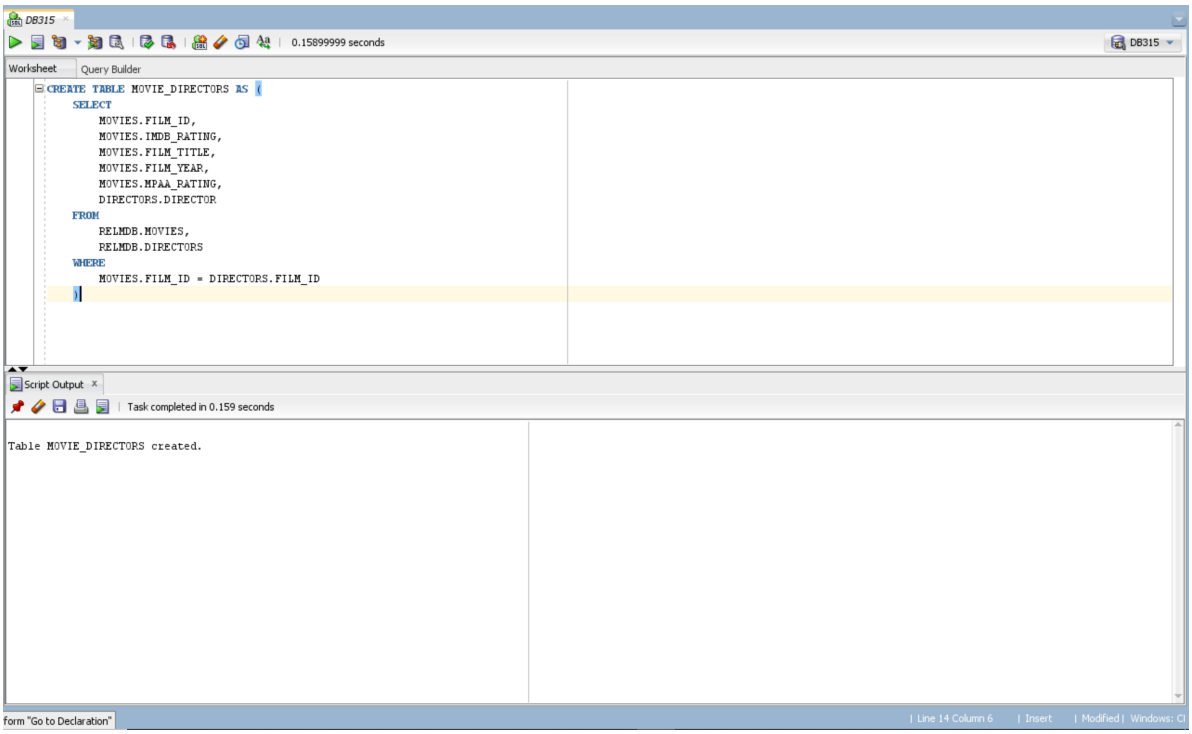


Figure 22: Table Created

3.7 Step - 2: Enabling Parallelism and Executing a Query

Now we run a complex query to find duplicates in the table based on the condition. We get an output with no entries, which means there are no duplicates. The Fig 23 shows the result.

```
1  ALTER TABLE MOVIE_DIRECTORS PARALLEL;
2
3  SELECT  /*+ PARALLEL(MOVIE_DIRECTORS 6)*/
4      IMDB_RATING,
5      FILM_TITLE,
6      FILM_YEAR,
7      MPAA_RATING,
8      DIRECTOR
9  FROM
10     MOVIE_DIRECTORS
11 GROUP BY
12     IMDB_RATING,
13     FILM_TITLE,
14     FILM_YEAR,
15     MPAA_RATING,
```

```
16      DIRECTOR
17  HAVING
18      COUNT(*) > 1;
```

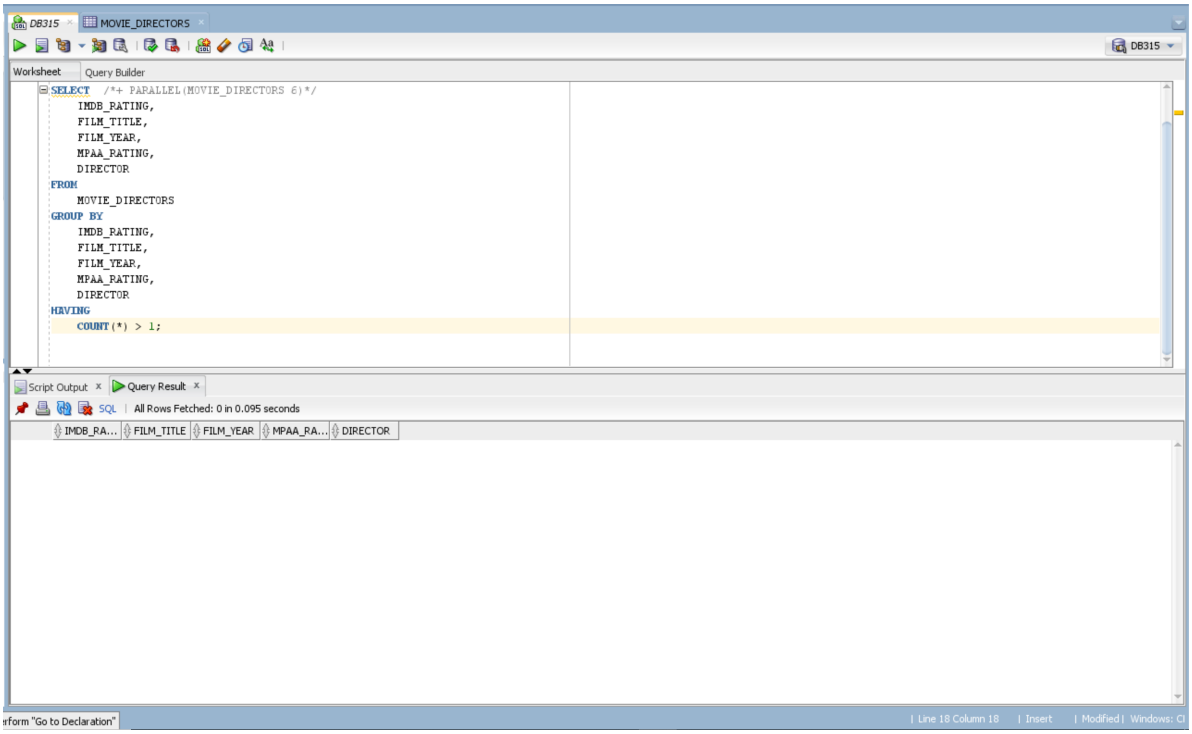


Figure 23: PX Enabled

3.8 Step - 3: Checking the Execution Plan

Lets now check the execution plans and see if there is any P-P, P-S transitions when the query executed. The results in Fig 24 shows that there was in fact multiple transitions at the time of execution.

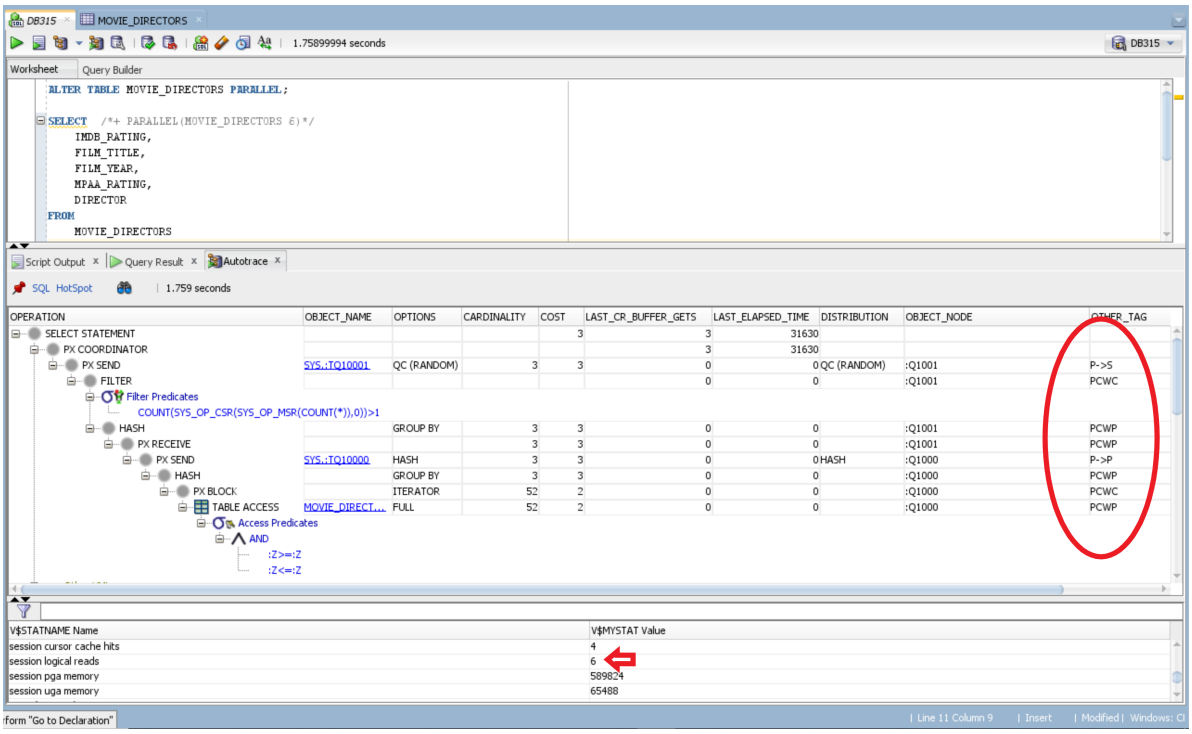


Figure 24: Execution Plan

3.9 Partitioned Tables

In this section, wee again use the RELMDB database to show how partitioning works. We will write some interesting queries to test out how the range partition works and see the partitioned result.

3.10 Step - 1: Creating a New Partitioned Table

We start by creating a new table called GENRE_TEST_RPART. We use the range partition to create this new table. The results in Fig 25 shows the output.

```
1 CREATE TABLE GENRE_TEST AS (SELECT * FROM RELMDB.GENRES);
2
3 CREATE TABLE GENRE_TEST_RPART (
4   FILM_ID NUMBER(10,0),
5   GENRE VARCHAR2(20),
6   GENRE_NAME VARCHAR2(20))
7 PARTITION BY RANGE (FILM_ID)
8  (PARTITION P30 VALUES LESS THAN (30),
9   PARTITION P60 VALUES LESS THAN (60),
```

```
  PARTITION P90 VALUES LESS THAN (90),
  PARTITION P120 VALUES LESS THAN (120));
```

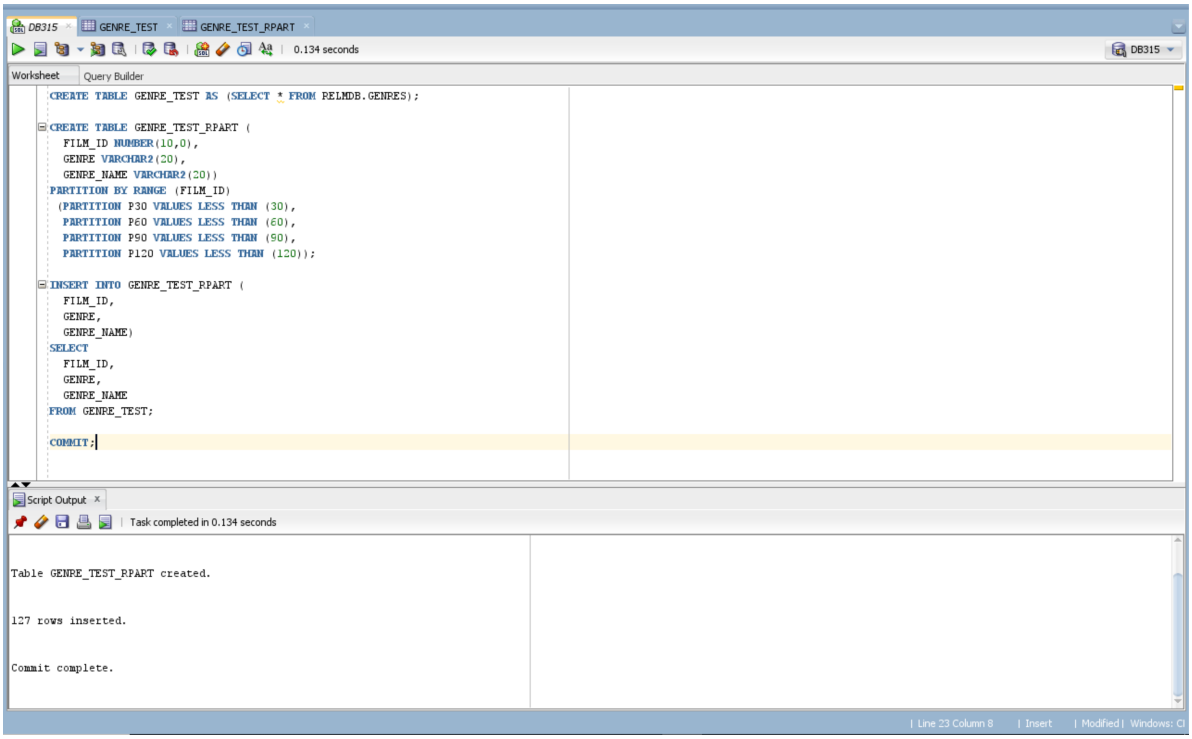


Figure 25: Table Created

3.11 Step - 2: Creating Partitioning and Inserting Data

Now we insert the data into the range partitioned table, commit and analyse for compute statistics. We have partitioned the data based on the FILM_ID in the new table with a range of 30 values in each partition. The results in Fig 26 shows the execution of our code.

```
INSERT INTO GENRE_TEST_RPART (
  FILM_ID,
  GENRE,
  GENRE_NAME)
SELECT
  FILM_ID,
  GENRE,
  GENRE_NAME
FROM GENRE_TEST;

```

```
11 COMMIT;
12
13 ANALYZE TABLE GENRE_TEST_RPART COMPUTE STATISTICS;
```

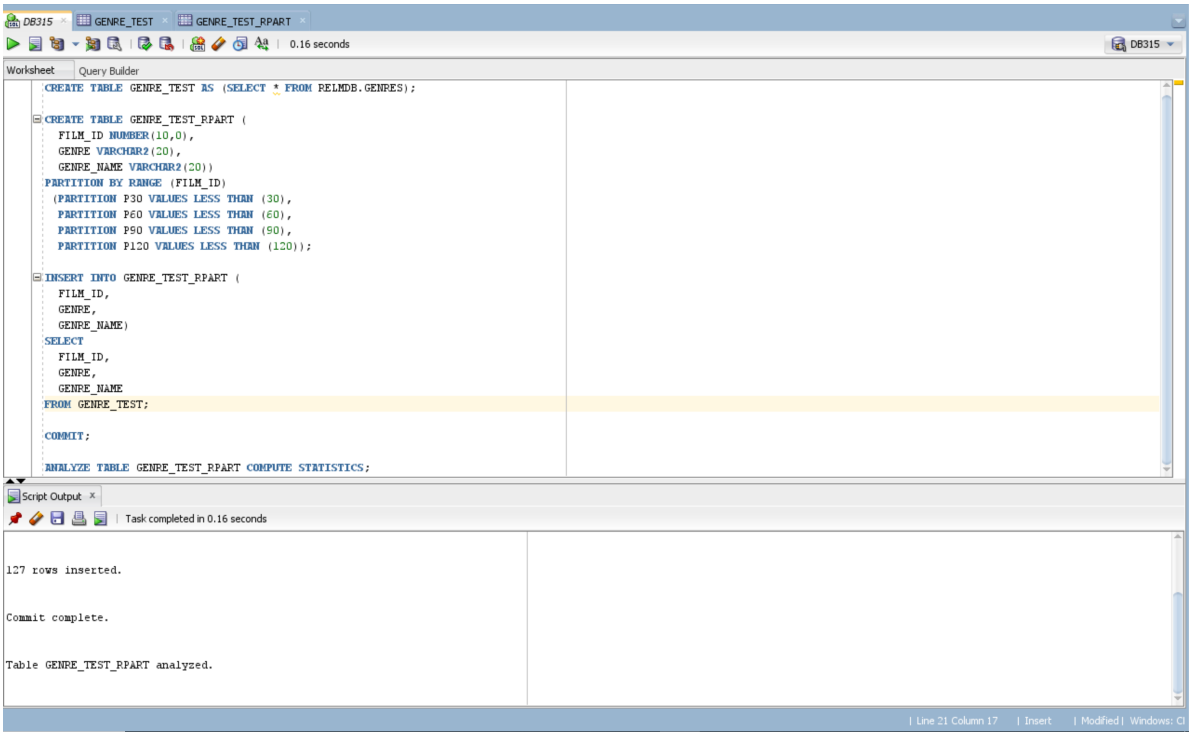


Figure 26: Query Executed

3.12 Step - 3: Inserting Data into the New Table

Here, we compare the partition tab before and after analysing to see the range partitioned results. The Fig 27 and 28 shows the comparison.

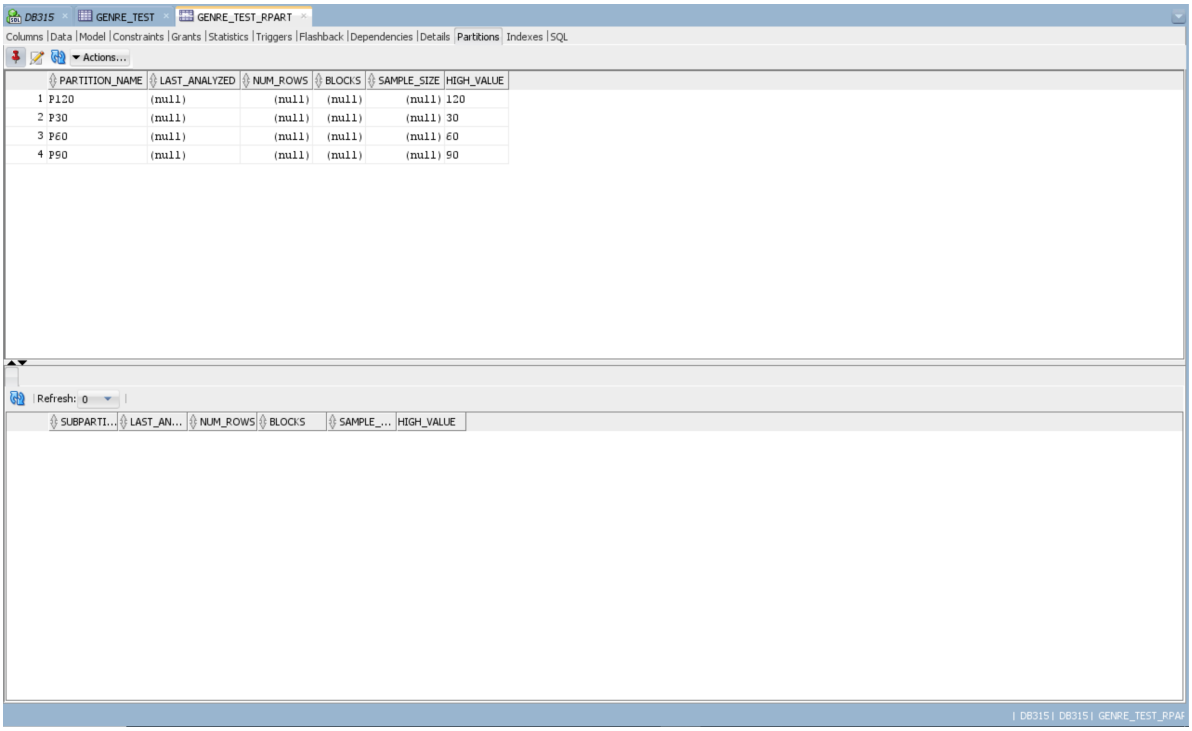


Figure 27: Before Data Insertion

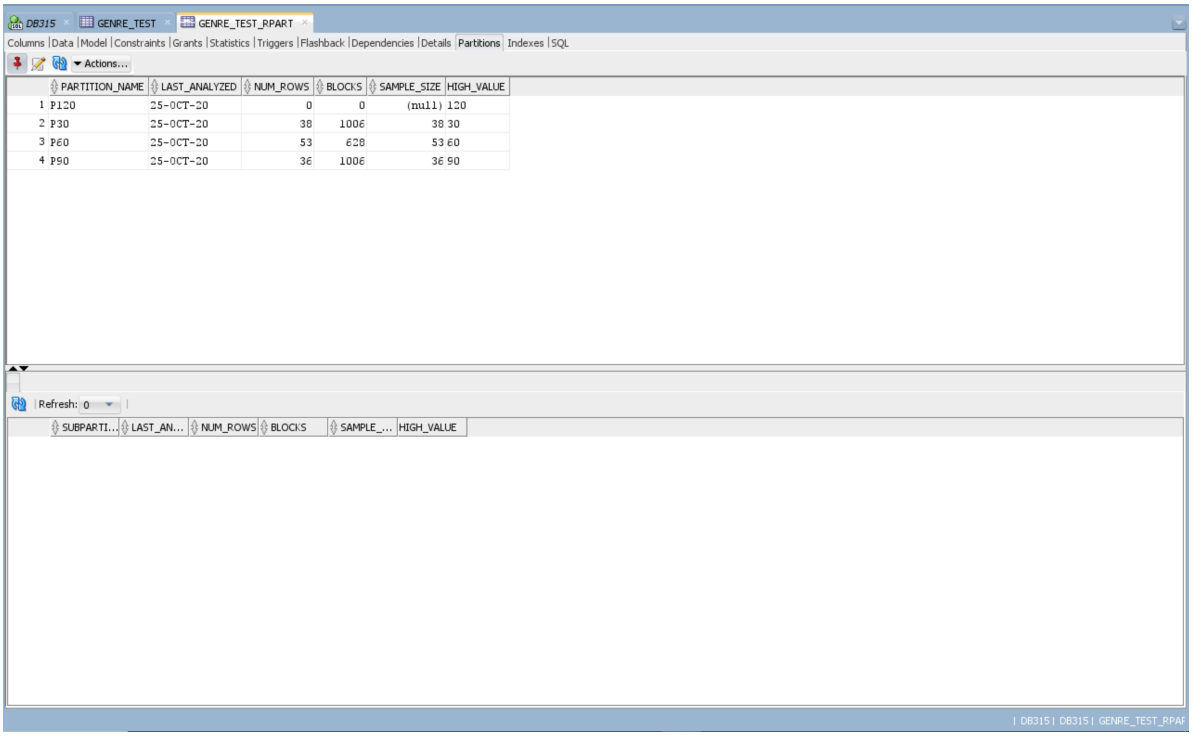


Figure 28: After Data Insertion

Finally, we see the PARTITION_START and PARTITION_STOP, which lists the partition identification numbers that define the range. From the Fig **??** below, we ran a simple query to check the execution plan which shows the partition range as well.

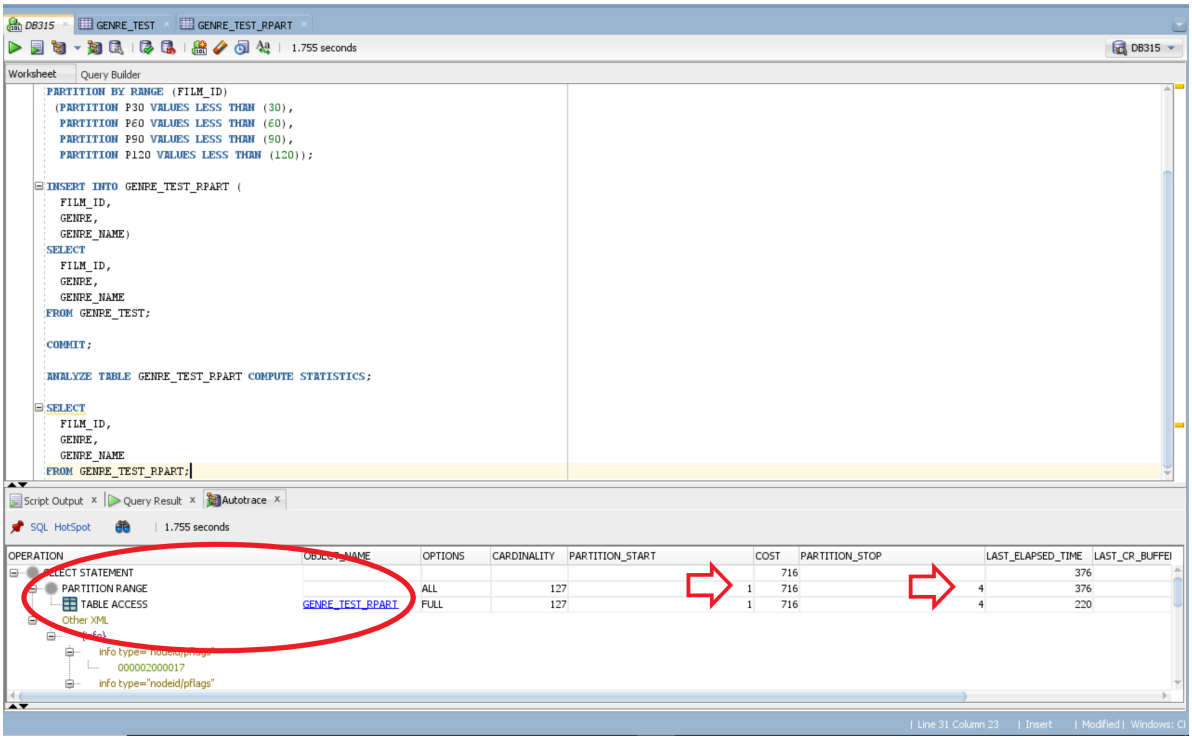


Figure 29: Execution Plan