

Optimizers, TCL, Parallelism, Partitioning

Advanced Database Management

Suryateja Chalapati

MS in Business Analytics and Information Systems
University of South Florida

October 26, 2020

Contents

1	Query Processing	1
1.1	Idea 1: Optimizer Modes: Response Time versus Throughput	1
1.2	Idea 2: Investigating Selectivity	2
2	Transaction Processing	3
2.1	Idea 5: Transaction Control Language	3
2.1.1	Scenario 1: INSERT Command	4
2.1.2	Scenario 2: DELETE Command	5
2.1.3	Scenario 3: UPDATE Command	7
2.2	Idea 6: Read Consistency Models	8
2.2.1	Transaction 1	8
2.2.2	Transaction 2	9
3	Parallel Databases	11
3.1	Basic Parallel Execution	11
3.2	Step - 1: Creating a Table and Executing a Simple Query	11
3.3	Step - 2: Running the Same Query with Parallelism	12
3.4	Step - 3: Comparing the Execution Plans	13
3.5	Transitions in More Complex Queries	15
3.6	Step - 1: Creating a New Table	15
3.7	Step - 2: Enabling Parallelism and Executing a Query	16
3.8	Step - 3: Checking the Execution Plan	17
3.9	Partitioned Tables	18
3.10	Step - 1: Creating a New Partitioned Table	18
3.11	Step - 2: Creating Partitioning and Inserting Data	19
3.12	Step - 3: Inserting Data into the New Table	20

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

```
1 SELECT
2     b.brewery_id ,
3     br.name,
4     COUNT(b.brewery_id)  no_of_beer,
5     NTILE(4) OVER(ORDER BY COUNT(br.brewery_id) DESC) rank_amount
6 FROM
7     beerdb.beers b
8     INNER JOIN beerdb.breweries br
9     ON b.brewery_id = br.brewery_id
10 WHERE
11     b.brewery_id IS NOT NULL
12 GROUP BY
13     b.brewery_id,br.name, br.brewery_id
14 ORDER BY
15     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.

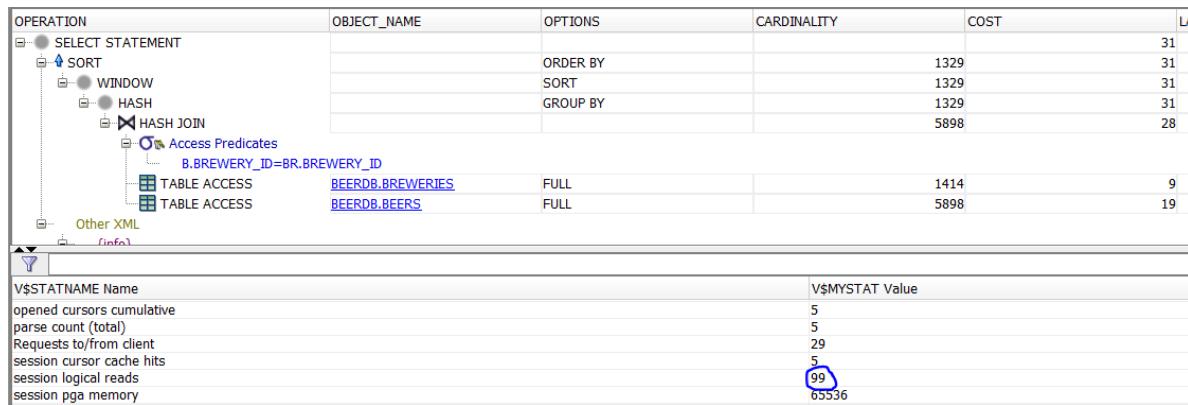


Figure 1: ALL_ROWS

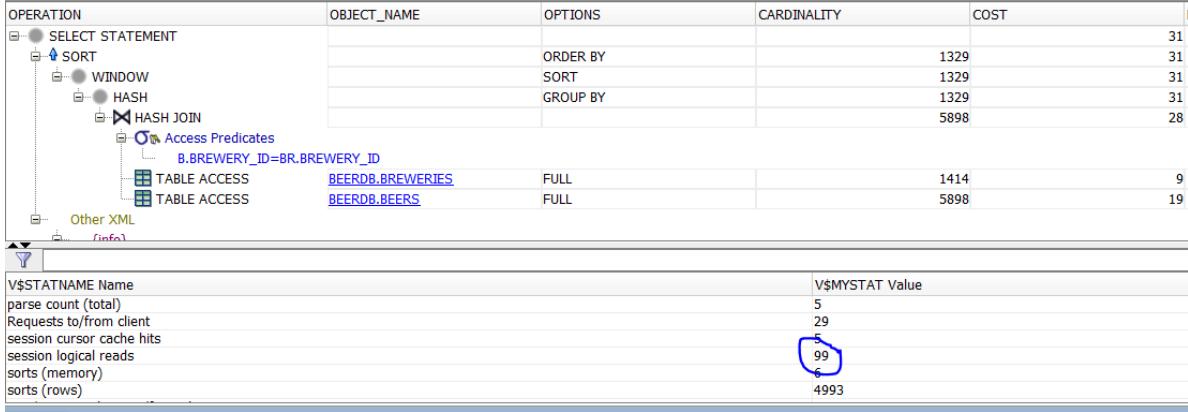


Figure 2: FIRST_ROWS_1

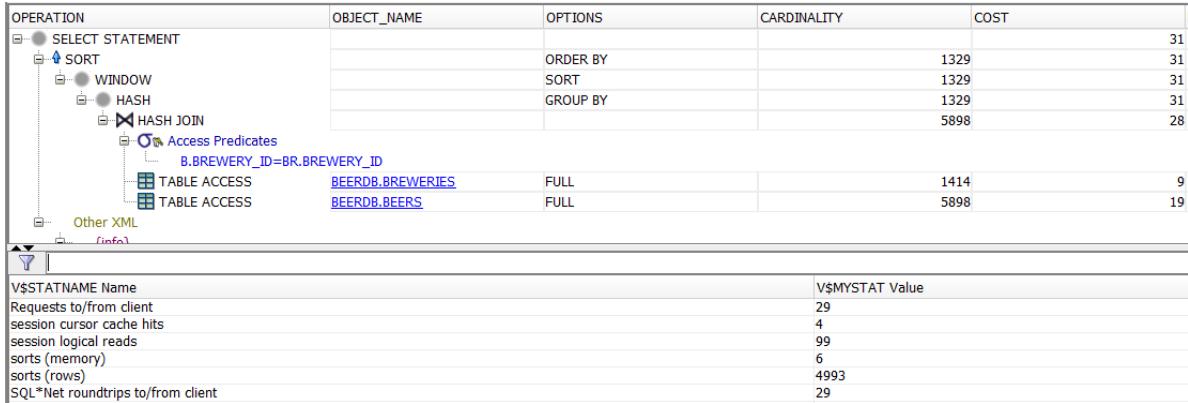


Figure 3: FIRST_ROWS_100

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 Throughput 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 known 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.

The screenshot shows the Oracle SQL Developer interface with the 'Worksheet' tab selected. In the query editor, the command `SELECT * FROM JOBS;` is entered. Below the editor, the 'Script Output' tab is active, displaying the results of the query. The results show 19 rows of data from the JOBS table, with columns: JOB_ID, JOB_TITLE, MIN_SALARY, and MAX_SALARY. The data includes various job titles like President, Vice President, Assistant, Manager, Accountant, Sales Manager, Purchasing Manager, Stock Manager, Clerk, Shipping Clerk, Programmer, Marketing Manager, Representative, and Human Resources Representative.

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
1	AD_PRES	20080	40000
2	AD_VP	15000	30000
3	AD_ASST	3000	6000
4	FI_MGR	8200	16000
5	FI_ACCOUNT	4200	9000
6	AC_MGR	8200	16000
7	AC_ACCOUNT	4200	9000
8	SA_MAN	10000	20080
9	SA_REP	6000	12008
10	PU_MAN	8000	15000
11	PU_CLERK	2500	5500
12	ST_MAN	5500	8500
13	ST_CLERK	2008	5000
14	SH_CLERK	2500	5500
15	IT_PROG	4000	10000
16	MK_MAN	9000	15000
17	MK_REP	4000	9000
18	HR_REP	4000	9000
19	PR_REP	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.

The screenshot shows the Oracle SQL Developer interface with the 'Worksheet' tab selected. In the query editor, the following commands are entered:

```

SELECT * FROM JOBS;

INSERT INTO JOBS
VALUES ('PR_MAN', 'Public Relationship Manager', 9000, 18000);

DELETE JOBS WHERE JOB_ID = 'PR_MAN';

UPDATE JOBS SET JOB_TITLE = 'PR Manager'
WHERE JOB_ID = 'PR_MAN';

COMMIT;

ROLLBACK;
    
```

The 'Script Output' tab is active, showing the results of the query. The results show 20 rows of data from the JOBS table, with columns: JOB_ID, JOB_TITLE, MIN_SALARY, and MAX_SALARY. The data includes various job titles and the newly inserted row for 'Public Relationship Manager' with ID 20.

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
10	PU_MAN	8000	15000
11	PU_CLERK	2500	5500
12	ST_MAN	5500	8500
13	ST_CLERK	2008	5000
14	SH_CLERK	2500	5500
15	IT_PROG	4000	10000
16	MK_MAN	9000	15000
17	MK_REP	4000	9000
18	HR_REP	4000	9000
19	PR_REP	4500	10500
20	PR_MAN	9000	18000

Figure 7: INSERT Without COMMIT

The screenshot shows a SQL developer interface with a query builder window. The code entered is:

```

SELECT * FROM JOBS;

INSERT INTO JOBS
VALUES ('PR_MAN', 'Public Relationship Manager', 9000, 18000);

DELETE JOBS WHERE JOB_ID = 'PR_MAN';

UPDATE JOBS SET JOB_TITLE = 'PR Manager'
WHERE JOB_ID = 'PR_MAN';

COMMIT;

ROLLBACK;

```

Below the code, the status bar indicates "All Rows Fetched: 19 in 0.055 seconds". A query result window is open, displaying the following table:

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
9	SA_REP	Sales Representative	6000 12000
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 8: ROLLBACK Before COMMIT

The screenshot shows a SQL developer interface with a query builder window. The code entered is identical to Figure 8:

```

SELECT * FROM JOBS;

INSERT INTO JOBS
VALUES ('PR_MAN', 'Public Relationship Manager', 9000, 18000);

DELETE JOBS WHERE JOB_ID = 'PR_MAN';

UPDATE JOBS SET JOB_TITLE = 'PR Manager'
WHERE JOB_ID = 'PR_MAN';

COMMIT;

ROLLBACK;

```

Below the code, the status bar indicates "All Rows Fetched: 20 in 0.053 seconds". A query result window is open, displaying the following table:

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 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.

The screenshot shows the Oracle SQL Developer interface. The 'Worksheet' tab is active, displaying the following SQL script:

```

SELECT * FROM JOBS;

INSERT INTO JOBS
VALUES ('PR_MAN', 'Public Relationship Manager', 9000, 18000);

DELETE JOBS WHERE JOB_ID = 'PR_MAN';

UPDATE JOBS SET JOB_TITLE = 'PR Manager'
WHERE JOB_ID = 'PR_MAN';

COMMIT;

ROLLBACK;

```

Below the worksheet, the 'Query Result 1' tab is selected, showing the following table output:

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
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 10: DELETE Before COMMIT

The screenshot shows the Oracle SQL Developer interface. The 'Worksheet' tab is active, displaying the same SQL script as Figure 10:

```

SELECT * FROM JOBS;

INSERT INTO JOBS
VALUES ('PR_MAN', 'Public Relationship Manager', 9000, 18000);

DELETE JOBS WHERE JOB_ID = 'PR_MAN';

UPDATE JOBS SET JOB_TITLE = 'PR Manager'
WHERE JOB_ID = 'PR_MAN';

COMMIT;

ROLLBACK;

```

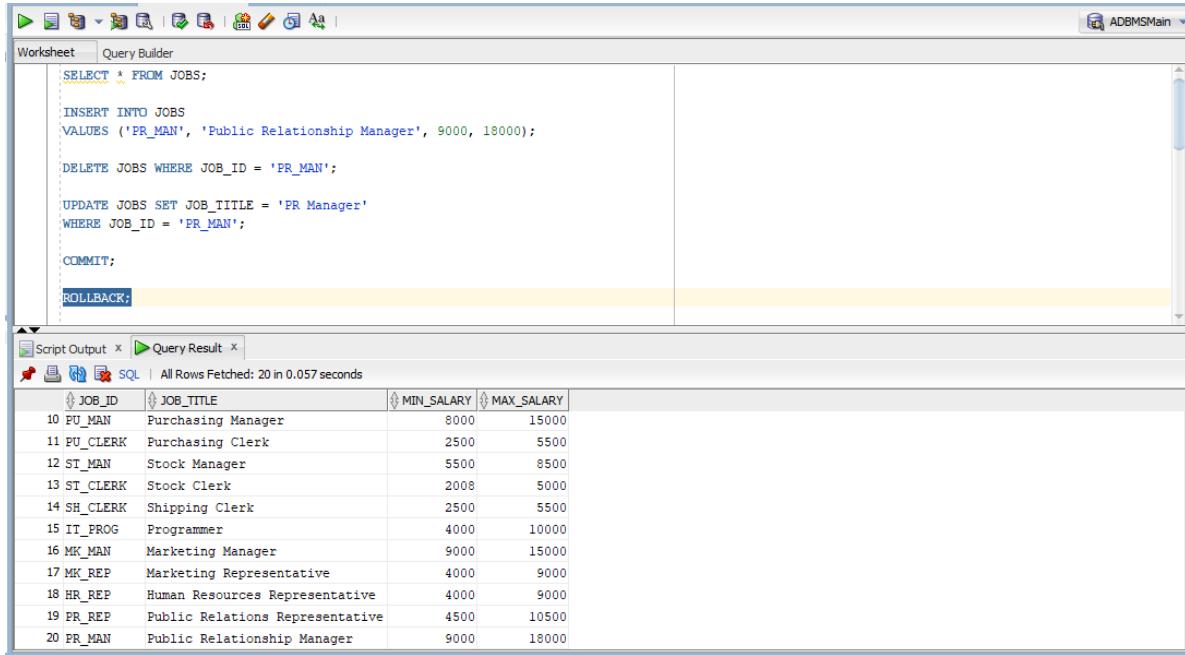
Below the worksheet, the 'Query Result 1' tab is selected, showing the following table output:

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 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.



The screenshot shows the Oracle SQL Developer interface. The top window is a 'Worksheet' containing a SQL script:

```
SELECT * FROM JOBS;  
  
INSERT INTO JOBS  
VALUES ('PR_MAN', 'Public Relationship Manager', 9000, 18000);  
  
DELETE JOBS WHERE JOB_ID = 'PR_MAN';  
  
UPDATE JOBS SET JOB_TITLE = 'PR Manager'  
WHERE JOB_ID = 'PR_MAN';  
  
COMMIT;  
  
ROLLBACK;
```

The bottom window is a 'Query Result' tab showing the contents of the JOBS table:

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 12: UPDATE Before COMMIT

The screenshot shows a SQL developer interface. In the top pane, a script is being run:

```

Worksheet | Query Builder
INSERT INTO JOBS
VALUES ('PR_MAN', 'Public Relationship Manager', 9000, 18000);

INSERT INTO JOBS
VALUES ('REL_MAN', 'Relationship Manager', 8000, 16000);

SELECT * FROM JOBS;

DELETE JOBS WHERE JOB_ID = 'REL_MAN';

UPDATE JOBS SET JOB_TITLE = 'PR Manager'
WHERE JOB_ID = 'PR_MAN';

COMMIT;

ROLLBACK;

```

The bottom pane shows the results of the SELECT query:

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
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	PR Manager	9000 18000

Script Output: All Rows Fetched: 20 in 0.054 seconds

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.

```

Worksheet | Query Builder
SELECT * FROM TRADES WHERE TRADE_ID = 1000000012
UPDATE TRADES SET QUANTITY = 4
WHERE TRADE_ID = 1000000012
|
--- Bill the customer
Waitfor Delay '00:00:15'

Rollback;

Query Result x
SQL | All Rows Fetched: 1 in 0.053 seconds
TRADE_ID BUYER_ID SELLER_ID MM_ID QUANTITY PRICE TRADE_TSTAMP SETTLED_TSTAMP SETTLED_PRICE BUY_ORD_ID S
1 1000000012 1000000009 100000001 100010 5 1 14-OCT-19 09.15.42.000000000 PM 15-JUL-20 11.00.01.000000000 PM 100 1000000045 1

```

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 5 as 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.

The screenshot shows the Oracle SQL Developer interface. The top window is a Worksheet titled "Query Builder". It contains the following SQL code:

```
Set transaction level read uncommitted;
SELECT * FROM TRADES WHERE TRADE_ID = 1000000012;
```

The bottom window is a "Query Result" tab showing the results of the query. The results are as follows:

TRADE_ID	BUYER_ID	SELLER_ID	MM_ID	QUANTITY	PRICE	TRADE_TSTAMP	SETTLED_TSTAMP	SETTLED_PRICE	BUY_ORD_ID	S
1 1000000012	100000009	100000001	100010	4	1 14-OCT-19 09.15.42.000000000 PM	15-JUL-20 11.00.01.000000000 PM	100	100000045	1	

The status bar at the bottom indicates "All Rows Fetched: 1 in 0.054 seconds".

Figure 15: Transaction 2 Uncommitted

The screenshot shows the Oracle SQL Developer interface. The top window is a Worksheet titled "Query Builder". It contains the following SQL code:

```
SELECT * FROM TRADES WHERE TRADE_ID = 1000000012
```

The bottom window is a "Query Result" tab showing the results of the query. The results are as follows:

TRADE_ID	BUYER_ID	SELLER_ID	MM_ID	QUANTITY	PRICE	TRADE_TSTAMP	SETTLED_TSTAMP	SETTLED_PRICE	BUY_ORD_ID	S
1 1000000012	100000009	100000001	100010	5	1 14-OCT-19 09.15.42.000000000 PM	15-JUL-20 11.00.01.000000000 PM	100	100000045	1	

The status bar at the bottom indicates "All Rows Fetched: 1 in 0.054 seconds".

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';
```

The screenshot shows the Oracle SQL Developer interface. The top window is titled 'FANS_PARALLEL' and displays the SQL code for creating a parallel table and performing a count operation. The bottom window is titled 'Script Output' and shows the resulting message: 'Table FANS_PARALLEL created.' The status bar at the bottom indicates the task completed in 0.184 seconds.

Figure 17: Table Created

The screenshot shows the Oracle SQL Developer interface. In the top-left pane (Worksheet), there is a code editor with the following SQL script:

```

CREATE TABLE FANS_PARALLEL AS
SELECT * FROM RELMDB.FANS;

SELECT COUNT(*) FROM FANS_PARALLEL WHERE MEMBERSHIP = 'GOLD';

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

```

In the bottom-right pane (Query Result), the output is displayed in a table:

	COUNT(*)
1	1914

Below the table, the status bar indicates "All Rows Fetched: 1 in 0.047 seconds".

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.

```

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

```

The screenshot shows the Oracle SQL Developer interface. In the top-left pane (Worksheet), there is a code editor with the following SQL script:

```
CREATE TABLE FANS_PARALLEL AS
SELECT * FROM RELMDB.FANS;

SELECT COUNT(*) FROM FANS_PARALLEL WHERE MEMBERSHIP = 'GOLD';

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

In the bottom-right pane (Query Result), the output is displayed in a table:

	COUNT(*)
1	1914

Below the table, the status bar shows: Script Output | Query Result | All Rows Fetched: 1 in 0.047 seconds.

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.

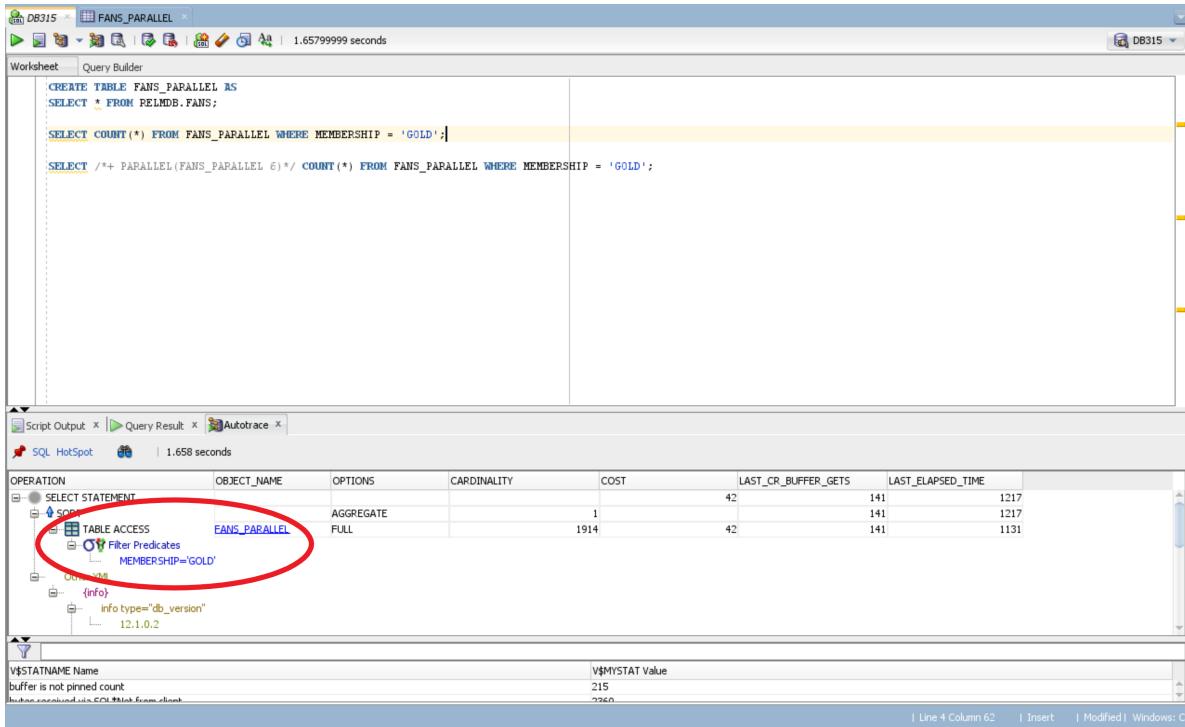


Figure 20: With PX

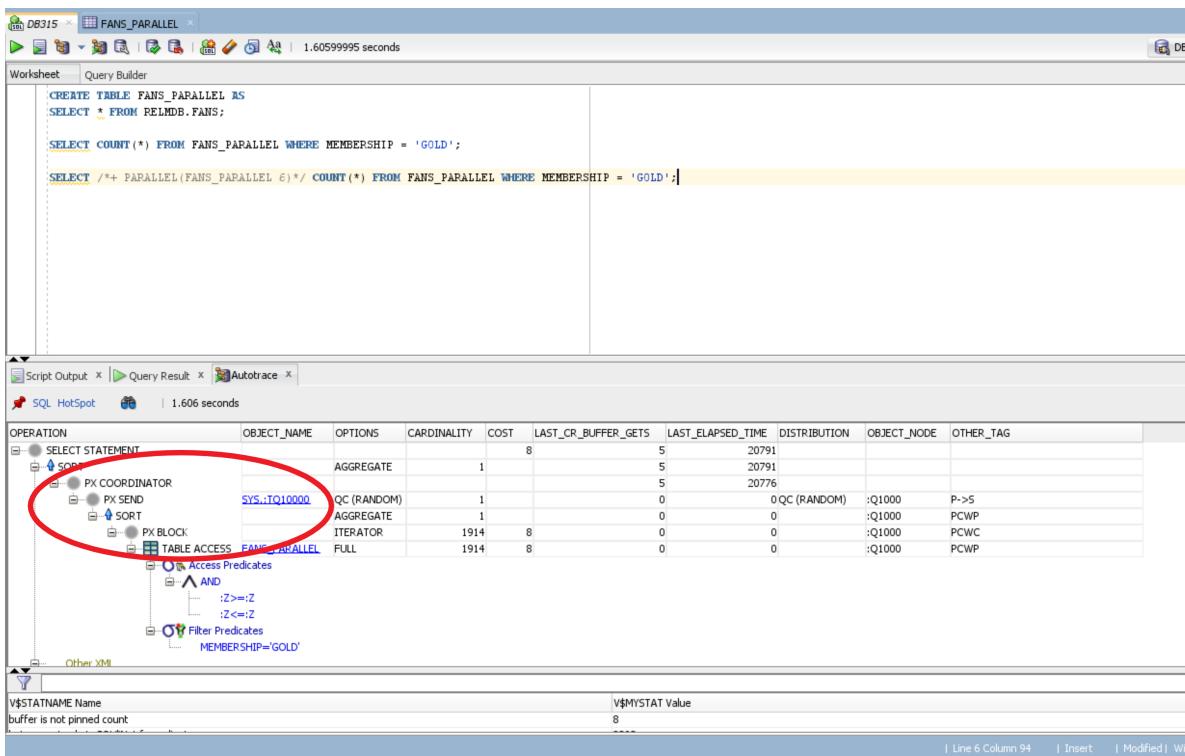


Figure 21: Without PX

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.

```
1 CREATE TABLE MOVIE_DIRECTORS AS (
2     SELECT
3         MOVIES.FILM_ID,
4         MOVIES.IMDB_RATING,
5         MOVIES.FILM_TITLE,
6         MOVIES.FILM_YEAR,
7         MOVIES.MPAA_RATING,
8         DIRECTORS.DIRECTOR
9     FROM
10        RELMDB.MOVIES,
11        RELMDB.DIRECTORS
12    WHERE
13        MOVIES.FILM_ID = DIRECTORS.FILM_ID
14    )
```

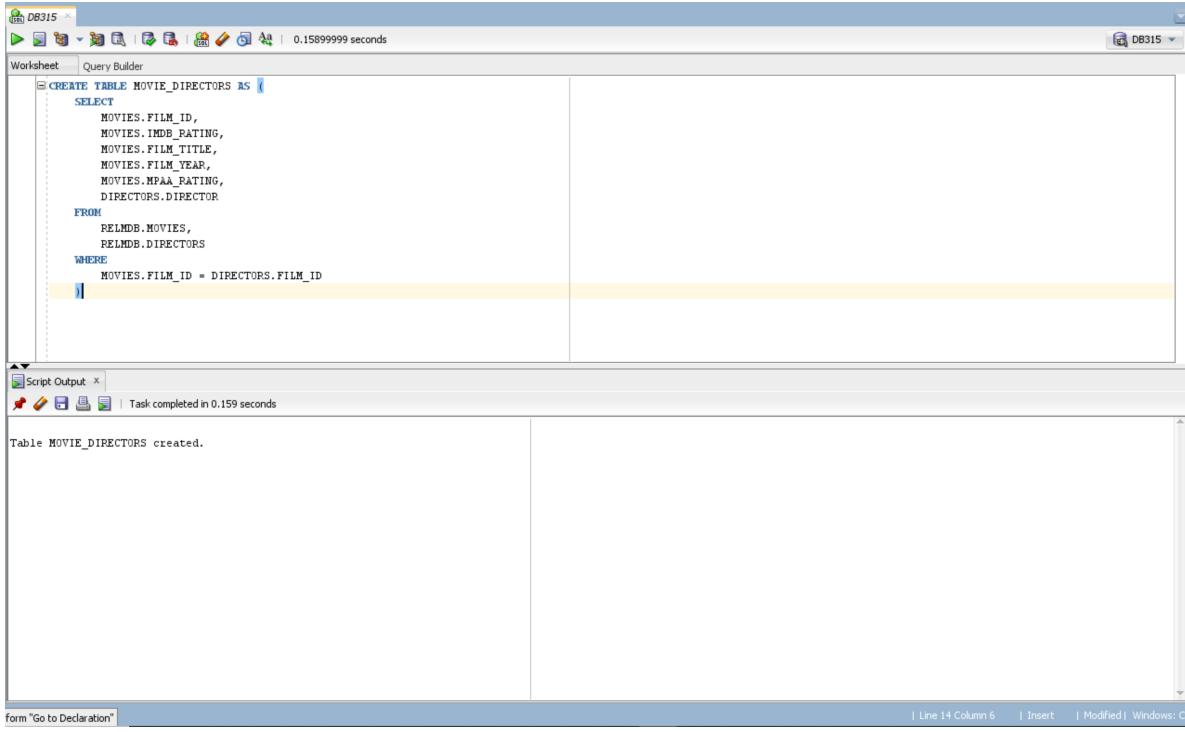


Figure 22: Table Created

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
10     FROM
11         MOVIE_DIRECTORS
12
13     GROUP BY
14         IMDB_RATING,
15         FILM_TITLE,
16         FILM_YEAR,
17         MPAA_RATING,
```

```

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

```

The screenshot shows the Oracle SQL Developer interface. In the top-left corner, there's a code editor window titled 'MOVIE_DIRECTORS' with the following SQL query:

```

SELECT /*+ PARALLEL(MOVIE_DIRECTORS 6) */
       IMDB_RATING,
       FILM_TITLE,
       FILM_YEAR,
       MPAA_RATING,
       DIRECTOR
  FROM MOVIE_DIRECTORS
 GROUP BY
       IMDB_RATING,
       FILM_TITLE,
       FILM_YEAR,
       MPAA_RATING,
       DIRECTOR
 HAVING COUNT(*) > 1;

```

The 'Query Result' tab below it displays the results of the query. The results table has columns: IMDB_RATING, FILM_TITLE, FILM_YEAR, MPAA_RATING, and DIRECTOR. There are no rows present in the result set.

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.

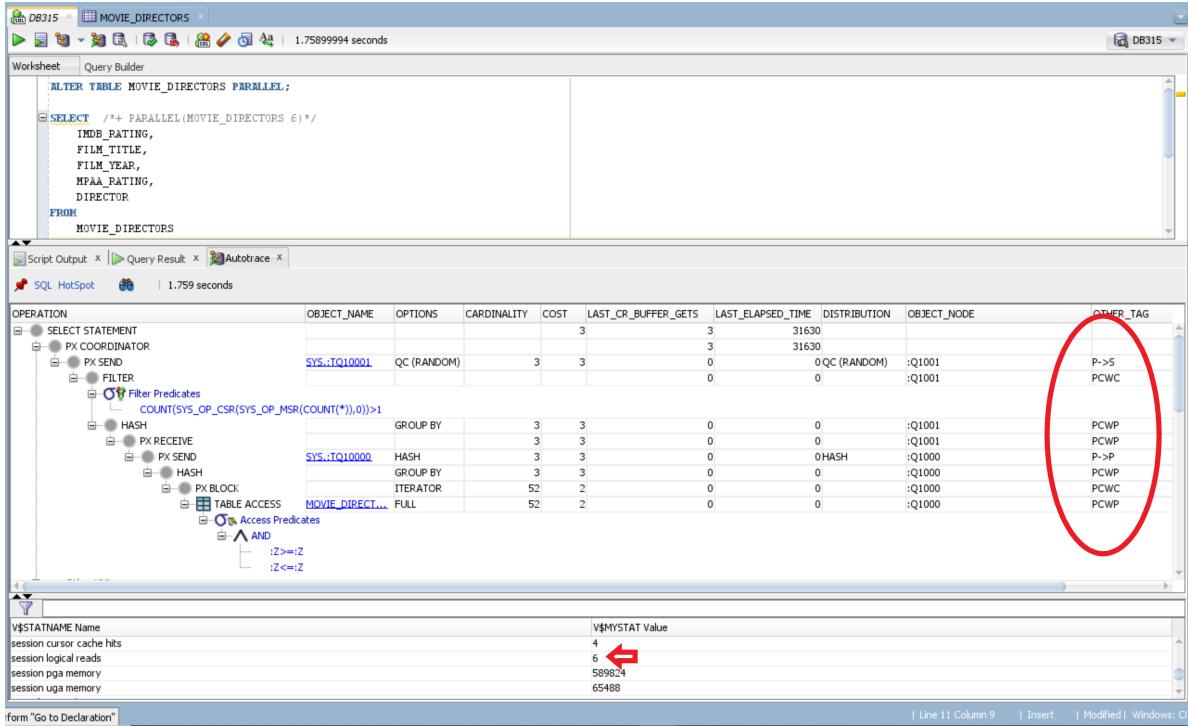


Figure 24: Execution Plan

3.9 Partitioned Tables

In this section, we 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),

```

```

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

```

The screenshot shows the Oracle SQL Developer interface. The top window is a Worksheet containing SQL code to create a partitioned table. The bottom window is a Script Output window showing the results of the execution.

```

CREATE TABLE GENRE_TEST AS (SELECT * FROM RELMDB.GENRES);

CREATE TABLE GENRE_TEST_RPART (
  FILM_ID NUMBER(10,0),
  GENRE VARCHAR2(20),
  GENRE_NAME VARCHAR2(20))
PARTITION BY RANGE (FILM_ID)
(PARTITION P30 VALUES LESS THAN (30),
 PARTITION P60 VALUES LESS THAN (60),
 PARTITION P90 VALUES LESS THAN (90),
 PARTITION P120 VALUES LESS THAN (120));

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

COMMIT;

```

Script Output:

```

Table GENRE_TEST_RPART created.

127 rows inserted.

Commit complete.

```

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.

```

1  INSERT INTO GENRE_TEST_RPART (
2    FILM_ID,
3    GENRE,
4    GENRE_NAME)
5  SELECT
6    FILM_ID,
7    GENRE,
8    GENRE_NAME
9  FROM GENRE_TEST;
10

```

```

11 COMMIT;
12
13 ANALYZE TABLE GENRE_TEST_RPART COMPUTE STATISTICS;

```

The screenshot shows the Oracle SQL Developer interface. The top window is titled 'Worksheet' and contains the following SQL script:

```

CREATE TABLE GENRE_TEST AS (SELECT * FROM RELNDB.GENRES);

CREATE TABLE GENRE_TEST_RPART (
    FILM_ID NUMBER(10,0),
    GENRE VARCHAR2(20),
    GENRE_NAME VARCHAR2(20))
PARTITION BY RANGE (FILM_ID)
(PARTITION P30 VALUES LESS THAN (30),
 PARTITION P60 VALUES LESS THAN (60),
 PARTITION P90 VALUES LESS THAN (90),
 PARTITION P120 VALUES LESS THAN (120));

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

COMMIT;

ANALYZE TABLE GENRE_TEST_RPART COMPUTE STATISTICS;

```

The bottom window is titled 'Script Output' and displays the execution results:

```

127 rows inserted.

Commit complete.

Table GENRE_TEST_RPART analyzed.

```

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.

DB315 | GENRE_TEST | GENRE_TEST_RPART

Actions... ▾

PARTITION_NAME	LAST_ANALYZED	NUM_ROWS	BLOCKS	SAMPLE_SIZE	HIGH_VALUE
1 P120	(null)	(null)	(null)	(null)	120
2 P30	(null)	(null)	(null)	(null)	30
3 P60	(null)	(null)	(null)	(null)	60
4 P90	(null)	(null)	(null)	(null)	90

SUBPARTITION_NAME	LAST_ANALYZED	NUM_ROWS	BLOCKS	SAMPLE_SIZE	HIGH_VALUE

Figure 27: Before Data Insertion

DB315 | GENRE_TEST | GENRE_TEST_RPART

Actions... ▾

PARTITION_NAME	LAST_ANALYZED	NUM_ROWS	BLOCKS	SAMPLE_SIZE	HIGH_VALUE
1 P120	25-OCT-20	0	0	(null)	120
2 P30	25-OCT-20	38	1006	38	30
3 P60	25-OCT-20	53	€28	53	60
4 P90	25-OCT-20	36	1006	36	90

SUBPARTITION_NAME	LAST_ANALYZED	NUM_ROWS	BLOCKS	SAMPLE_SIZE	HIGH_VALUE

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.

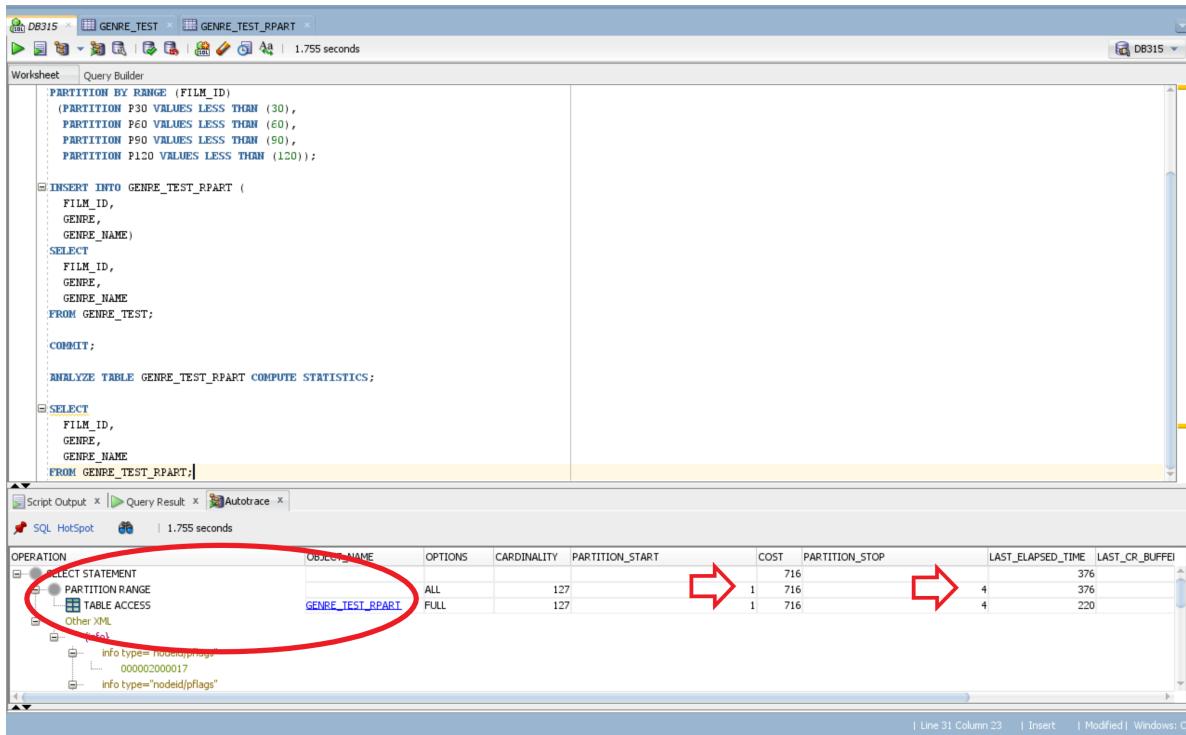


Figure 29: Execution Plan