Selectivity, Indexing, Database Programming
Advanced Database Management

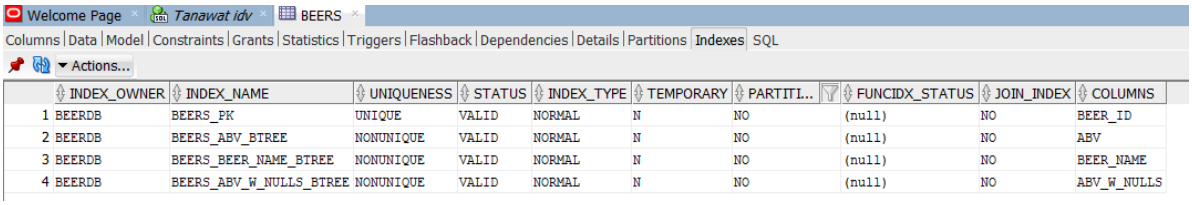
Suryateja Chalapati
MS in Business Analytics and Information Systems
University of South Florida
October 17, 2020

Contents

1 Idea 1: Investigating Selectivity

In this section, the BEERDB is used to identify when the table using index by selecting the factor. First of all the index in this table contain which are the following in Fig 1:

BEER_ID
ABV
BEER_NAME
ABV_W_NULLS



	INDEX_OWNER	INDEX_NAME	UNIQUENESS	STATUS	INDEX_TYPE	TEMPORARY	PARTITI...	FUNCIDX_STATUS	JOIN_INDEX	COLUMNS
1	BEERDB	BEERS_PK	UNIQUE	VALID	NORMAL	N	NO	(null)	NO	BEER_ID
2	BEERDB	BEERS_ABV_BTREE	NONUNIQUE	VALID	NORMAL	N	NO	(null)	NO	ABV
3	BEERDB	BEERS_BEER_NAME_BTREE	NONUNIQUE	VALID	NORMAL	N	NO	(null)	NO	BEER_NAME
4	BEERDB	BEERS_ABV_W_NULLS_BTREE	NONUNIQUE	VALID	NORMAL	N	NO	(null)	NO	ABV_W_NULLS

Figure 1: Index on BEERDB

1.1 Using style_id, cat_id, and brewery_id

Those three above are not indexes. The results in Fig 2 shows that the application decide to use standard searching, also known as full scan.

```
1 SELECT *
2 FROM beerdb.beers
3 WHERE style_id = 38 AND cat_id = 3 AND brewery_id = 133;
```

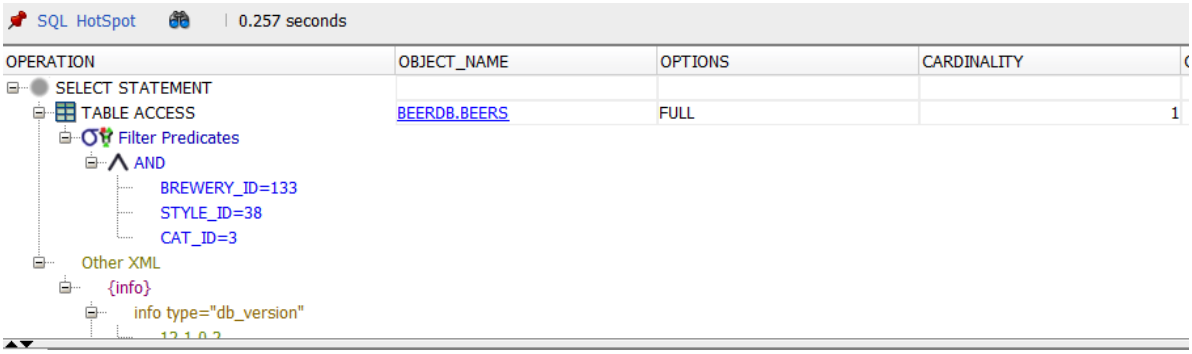


Figure 2: style_id, cat_id, and brewery_id

1.2 Using beer_name, style_id, and brewery_id

Those three above, only beer_name is an index. The results in Fig 3 shows that the application decide to use indexing by which using range scan on beer_name.

```sql
SELECT *
FROM beerdb.beers
WHERE beer_name = 'Nutty Brewnette' AND
      style_id = 38 AND
      brewery_id = 133;
```

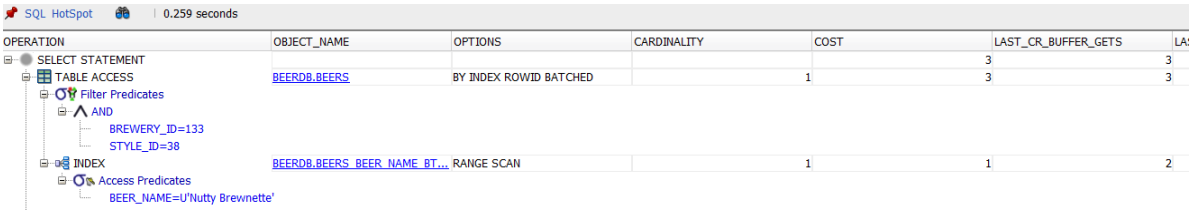


Figure 3: beer_name, style_id, and brewery_id

1.3 Using beer_name, beer_id, and brewery_id

Those three above, beer_name and beer_id are indexes. However, because beer_d is the primary key, the oracle focus to the column of beer_id and uses unique scan instead of range scan. The Fig 4 shows the results on this experiment.

```sql
SELECT *
FROM beerdb.beers
WHERE beer_name = 'Nutty Brewnette' AND
      beer_id = 319 AND
      brewery_id = 133;
```

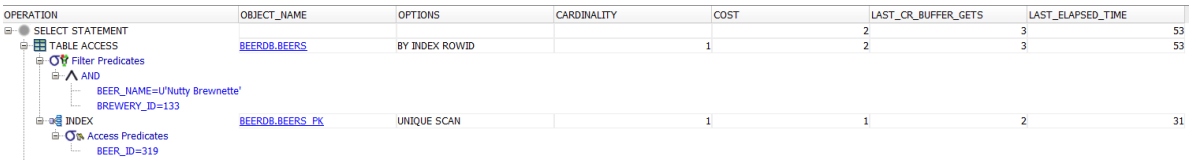


Figure 4: unique scan

2 Idea 2: Start Simple and Show that Indexing Works

The idea for showing the performance of indexing is conducted in RELMDB database. This database has already been added the index, which are film_title and film_ID.

2.1 Comparison between index factor and non-index factor

First of all, the experiment try to use non-index of the movies named ”Fight Club” to perform the searching. Using runtime as a searching factor represents the session log read equal to 7 shown by Fig 2.1.

```
SELECT *
FROM
    relmdb.movies
WHERE runtime = 139;
```

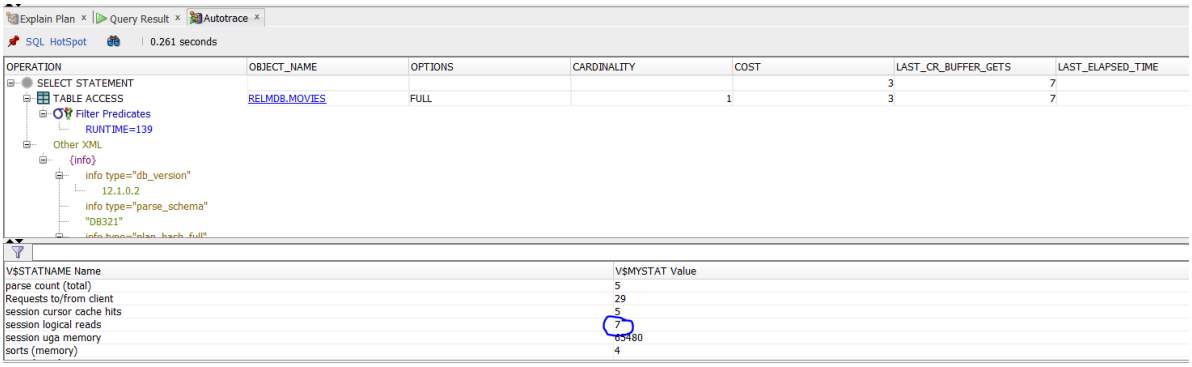


Figure 5: Fig 2.1

Then, the index factors, which are film_title and film_ID, are selected to be compared. The film_title index shows that the session logical read reduce from 7 to 3 shown in Fig 2.2.

```
SELECT *
FROM
    relmdb.movies
WHERE film_title = 'Fight Club';
```

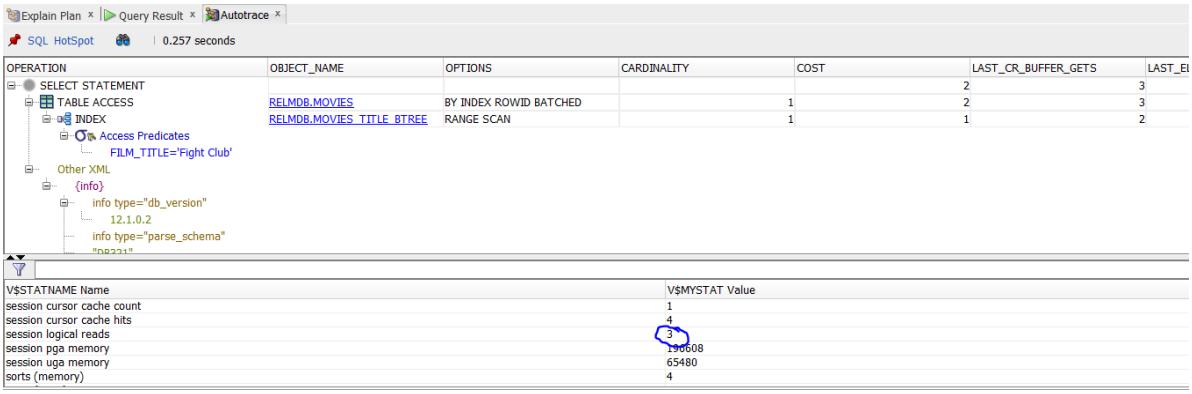


Figure 6: Fig 2.2

Finally, the film_id, the last index, is used. The result is presented by Fig 2.3 which logical read is 2.

```
1 SELECT *
2 FROM
3     relmdb.movies
4 WHERE film_id = '10';
```

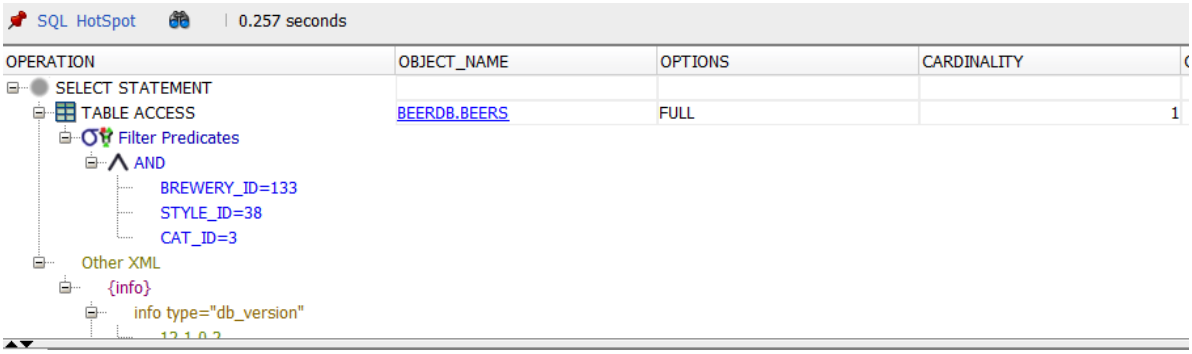


Figure 7: Fig 2.3

In conclusion, among those 3 factors, film_id has the least of session logical reads.

3 Idea 3: Primary Keys and Indexes

To perform this experiment, FANS table from RELMDB database has been selected. Table consists of SSN_12 which is identified as the candidate key and FAN_ID acts as primary key. Experiments are done to compare and estimate the performance cost by inserting a new row.

3.1 Experiment 1

In this experiment, unique constraint is added to the candidate key, which is SSN_12 while there is no Primary key constraint. A new row is inserted into the FAN table which takes 0.039 seconds to complete the task as shown in Fig. 8. This automatically creates the index on the SSN_12.

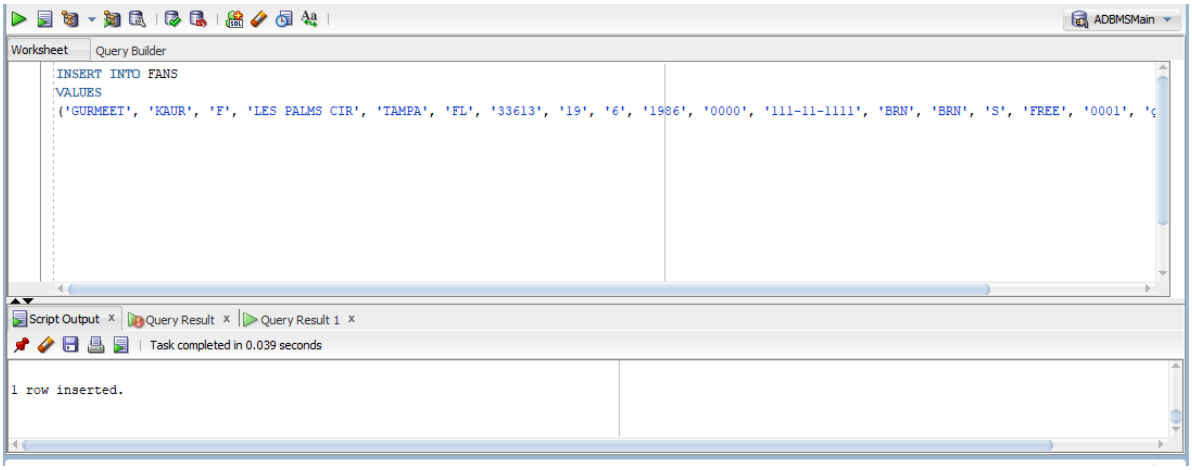


Figure 8: Experiment 1

3.2 Experiment 2

In this experiment, unique constraint from the SSN_12 is removed and primary key constraint is created on FAN_ID. This automatically creates the index on the FAN_ID. Now, the new row is inserted into the FAN table and it takes 0.03 seconds to perform the task as shown in Fig. 9.

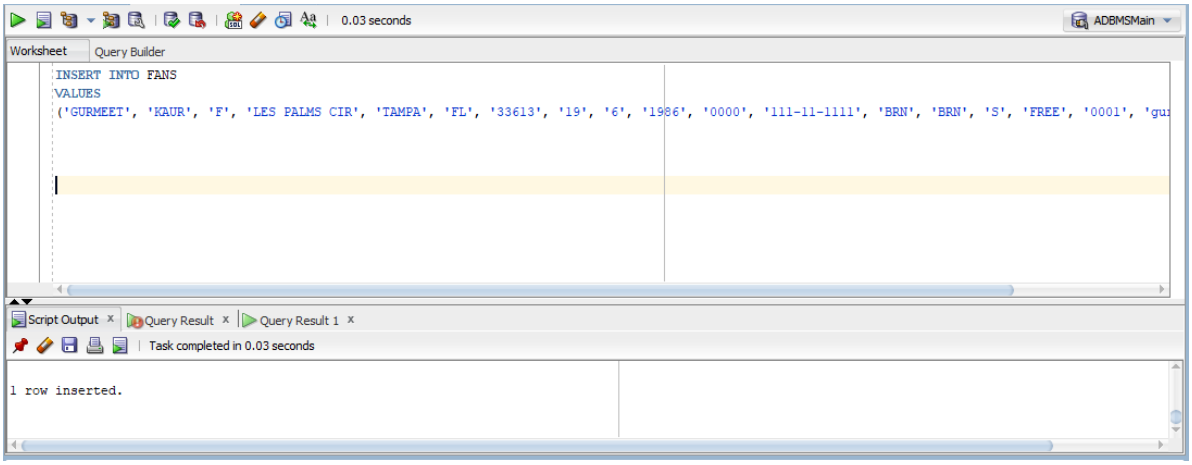


Figure 9: Experiment 2

Performance Comparison between the two experiments shows very little to no difference in

completing the task. There is no performance penalty in a query that uses an unique index, if the table does not have a primary index.

4 Idea 4: Indexing for Different Query Types

MOVIES table is selected from RELMDB database to perform range of queries to explore the importance of index. Primary key constraint is created on FAN_ID column.

4.1 Scenario 1

Point query is created by displaying results for a particular FILM_ID. This performs unique search by using primary key index as shown in Fig. 10. This fetches 1 unique row in 0.058 seconds.

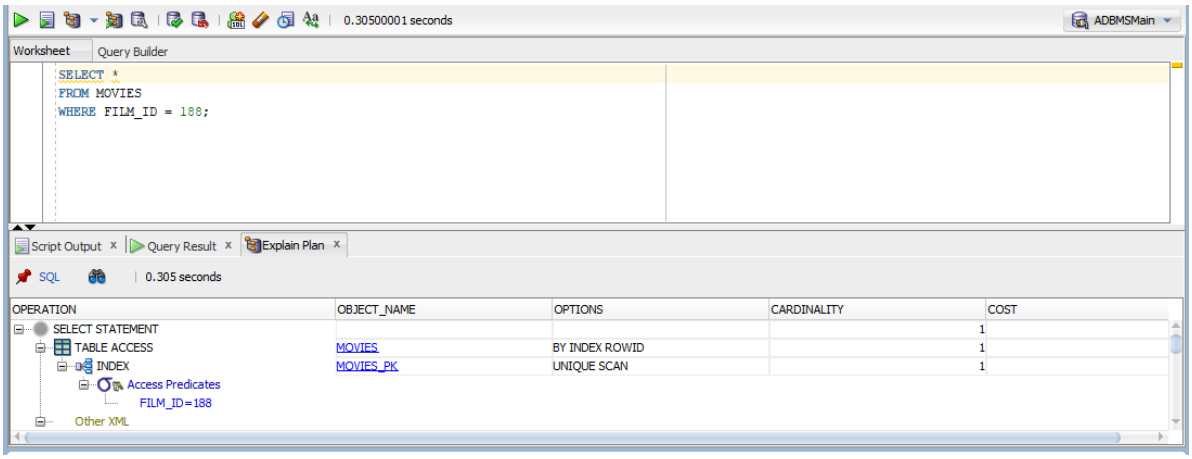


Figure 10: Scenario 1

4.2 Scenario 2

Range query is created by displaying large set of FILM_ID between range 180 and 190. This performs range scan on primary key index and fetches 10 unique rows in 0.055 seconds as shown in Fig. 11.

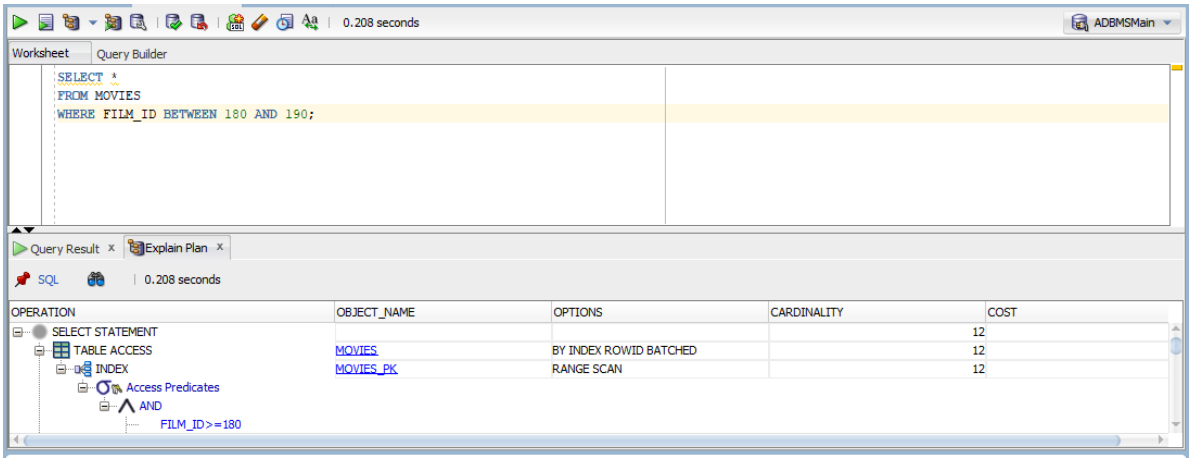


Figure 11: Scenario 2

4.3 Scenario 3

Full scan is created by increasing range of FILM_ID by 1, that is, between 180 and 190 to 180 and 191. This query performs full scan to fetch 11 unique rows in 0.056 seconds as shown in Fig. 12.

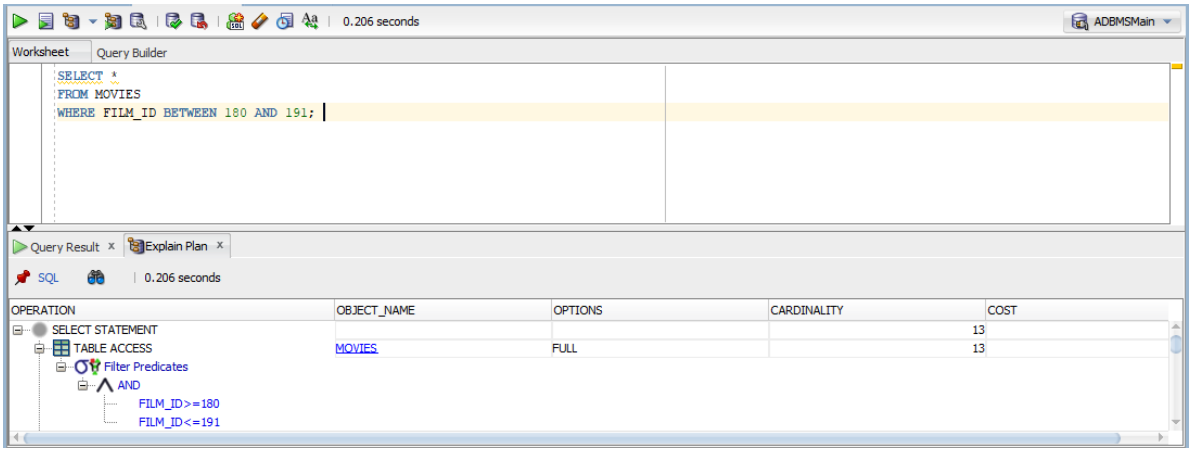


Figure 12: Scenario 3

4.4 Scenario 4

Full scan query displays all FAN_ID that starts with 2. This query performs full scan search to fetch 95 rows in 0.122 seconds as shown in Fig. 13.

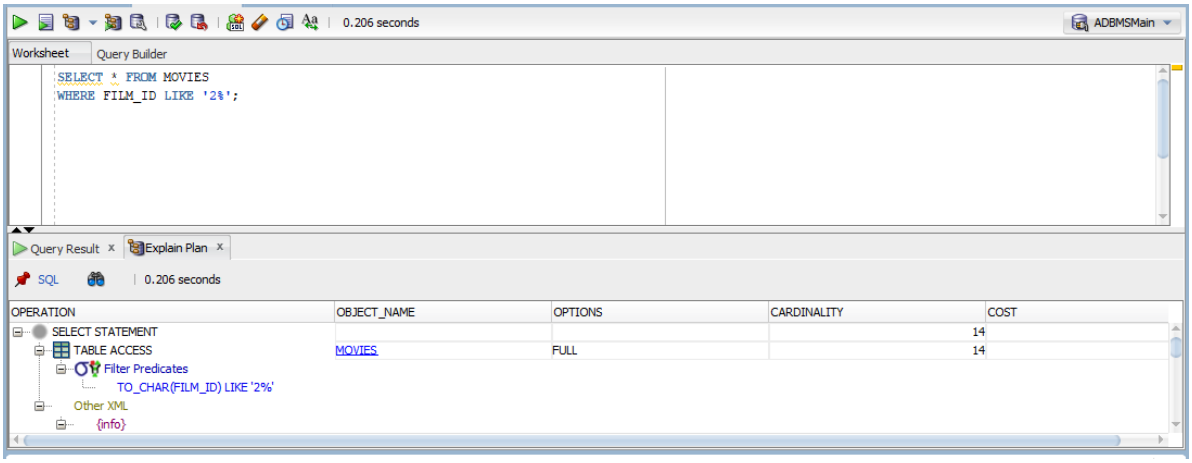


Figure 13: Scenario 4

5 Idea 5: Function-Based Indexes

For this experiment, FANS_DUP table from TEST3 database has been selected. We are specifically interested in finding the impact of Function-Based indexing on a range of data in a particular column. One such scenario is implemented below to reduce the performance cost by using Function-based indexing.

5.1 Step - 1: Without Function-Based Indexing

First we fetch a range of birthday dates from BIRTH_DAY column on FANS_DUP. We are using a CEIL() function to fetch the range on BIRTH_DAY. Upon running the auto-trace on the execution plan, we find out that indexing is not being used and a ”Full” scan takes place. This is shown in Fig. 14. The cost of fetching is 42.

```
SELECT
    COUNT(*)
FROM FANS_DUP
WHERE CEIL(BIRTH_DAY) > 10 AND CEIL(BIRTH_DAY) < 20;
```

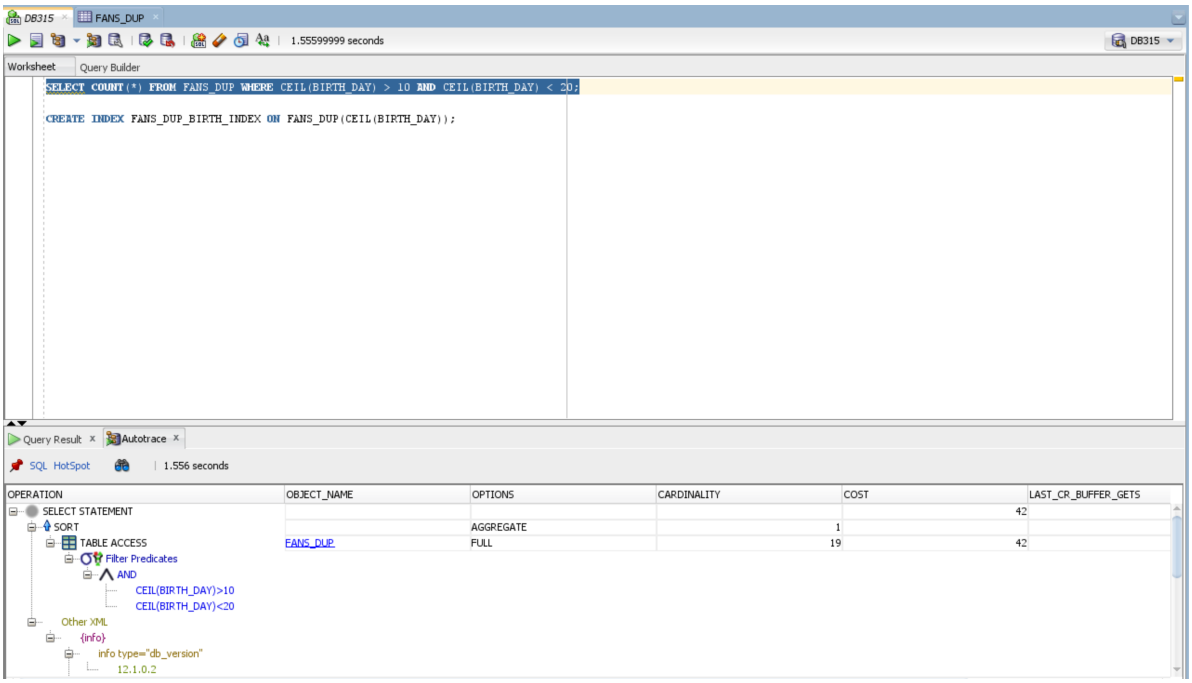


Figure 14: Step - 1

5.2 Step - 2: With Function-Based Indexing

Here, we now use Function-Based indexing by creating an index on FANS_DUP. Then we fetch a range of birthday dates from BIRTH_DAY column on FANS_DUP. We are using a CEIL() function to fetch the range on BIRTH_DAY. Upon running the auto-trace on the execution plan, we find out that Function-based indexing is being used and a ”Range” scan takes place. This is shown in Fig. 15. The cost of fetching is 6.

```
1 CREATE INDEX FANS_DUP_BIRTH_INDEX ON FANS_DUP(CEIL(BIRTH_DAY));
```

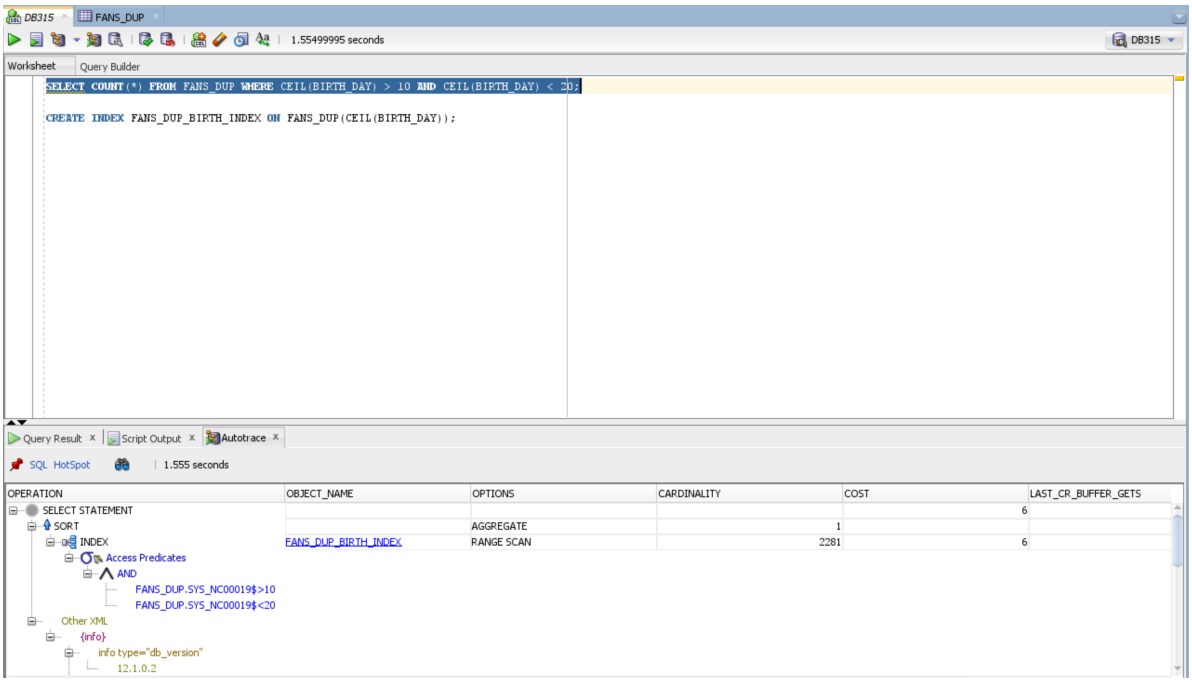


Figure 15: Step - 2

5.3 Conclusion

Function-Based Indexing has following advantages:

1. A function-based index speeds up the query by giving the optimizer more chance to perform an index range scan instead of full index scan.

2. A function-based index reduces computation for the database.

6 Idea 6: Database Programming

For this experiment, We are creating a PROC_FAN_COUNT which counts how many fans have the BIRTH_DAY between a range. The range here is [10, 20]. The steps to create a procedure are as follows.

6.1 Step - 1: Creating a Procedure

First we create a procedure to count the fans birthday range by defining a BEGIN_DATE and END_DATE. We are using the FANS_DUP table for this experiment from TEST3.

```
1 CREATE OR REPLACE PROCEDURE
2     PROC_FAN_COUNT (BEGIN_DATE in number, END_DATE in number)
3 IS COUNT_FAN number := 0;
```

```
4
5  BEGIN
6  SELECT COUNT(*) INTO COUNT_FAN
7  FROM FANS_DUP
8  WHERE BIRTH_DAY BETWEEN BEGIN_DATE AND END_DATE;
9  SYS.DBMS_OUTPUT.PUT_LINE (COUNT_FAN);
10 END PROC_FAN_COUNT;
```

6.2 Step - 2: Executing the Procedure

Then we execute the procedure to fetch the birthday range between dates [10, 20]. This gives us an output of 2779 rows as shown in the Fig. 16

```
1  SET SERVEROUTPUT ON;
2  BEGIN
3  PROC_FAN_COUNT(10,20);
4  END;
```

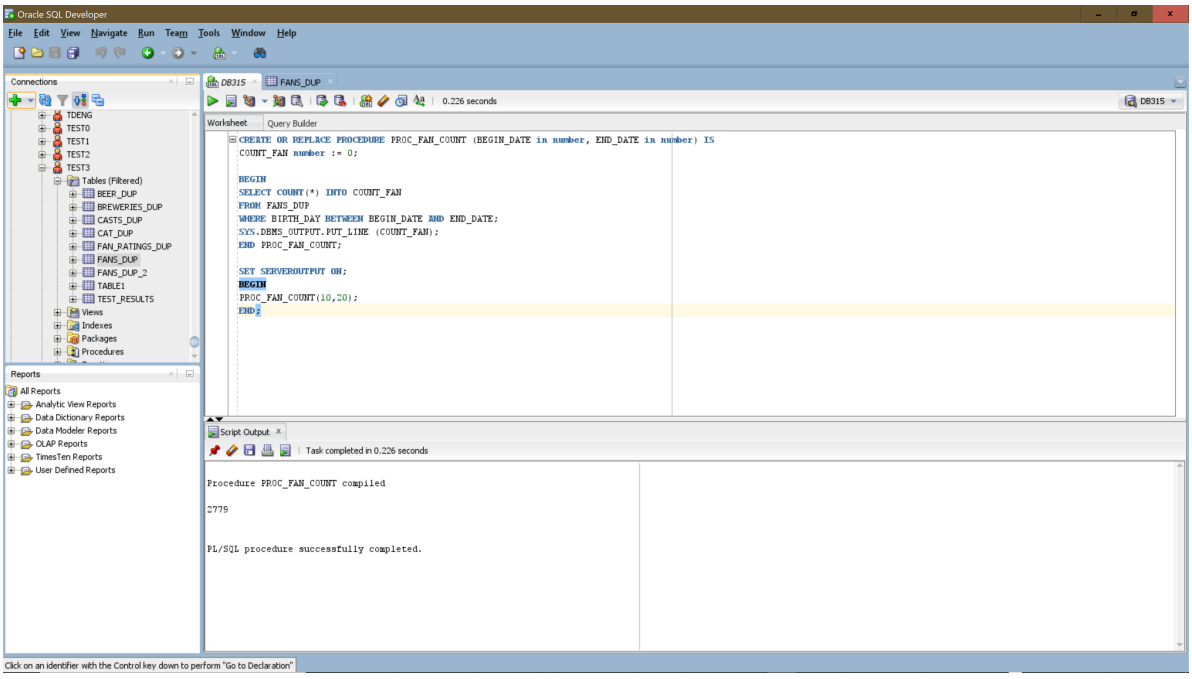


Figure 16: Query Execution