Assignment 3: Indexing EotW

Group 10 – Emerald

Kanchan Chowdhari

Payal Kaur

Purva Khandelwal

Samiksha Mhatre

1. Idea 1:Investigating Selectivity

To demonstrate this experiment, we have queried upon one of our Group project's table- PATIENT_RECORDS.

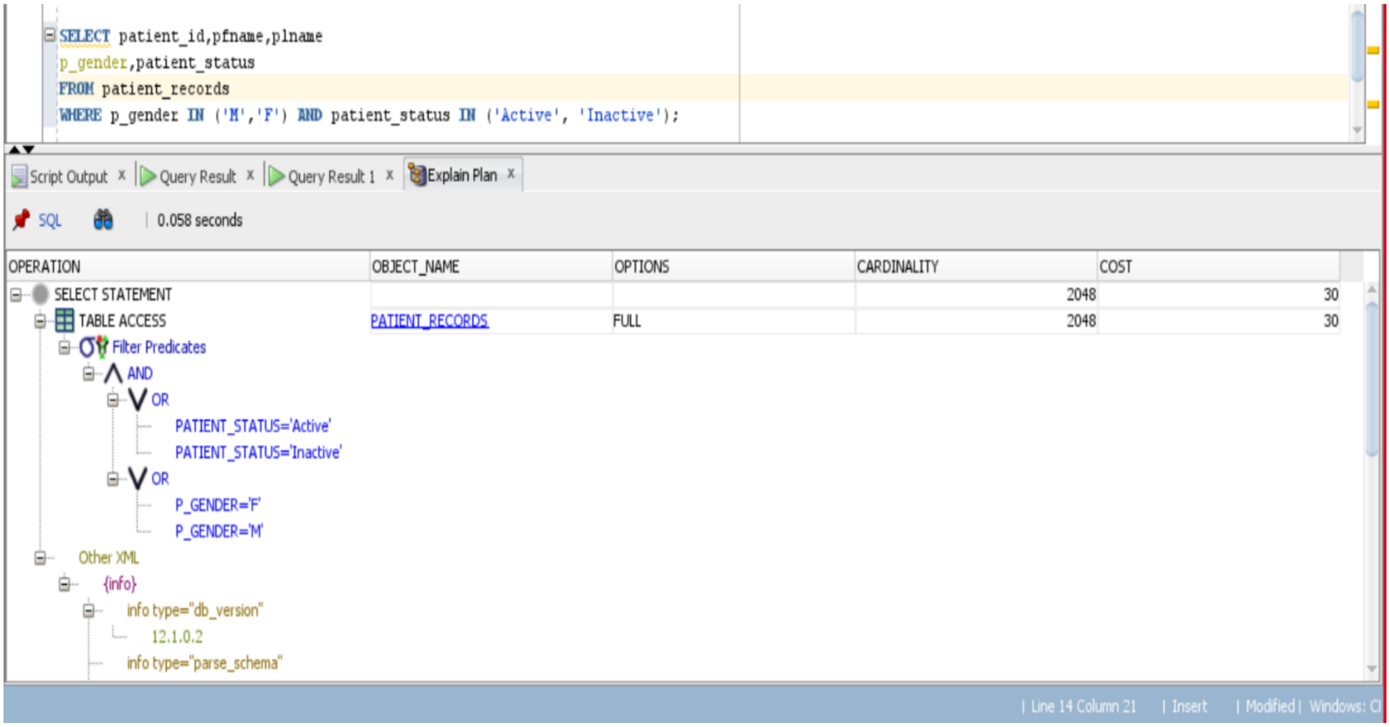
1. Let's **display the Patient name, Patient ID, gender and patient status** from Patient_Records table **without indexing**.

   SELECT patient_id, pfname,plname,

   p_gender,patient_status

   FROM patient_records

   WHERE p_gender IN ('M','F') **AND** patient_status **IN** ('Active');



```
SELECT patient_id,pfname,plname
p_gender,patient_status
FROM patient_records
WHERE p_gender IN ('M','F') AND patient_status IN ('Active', 'Inactive');
```

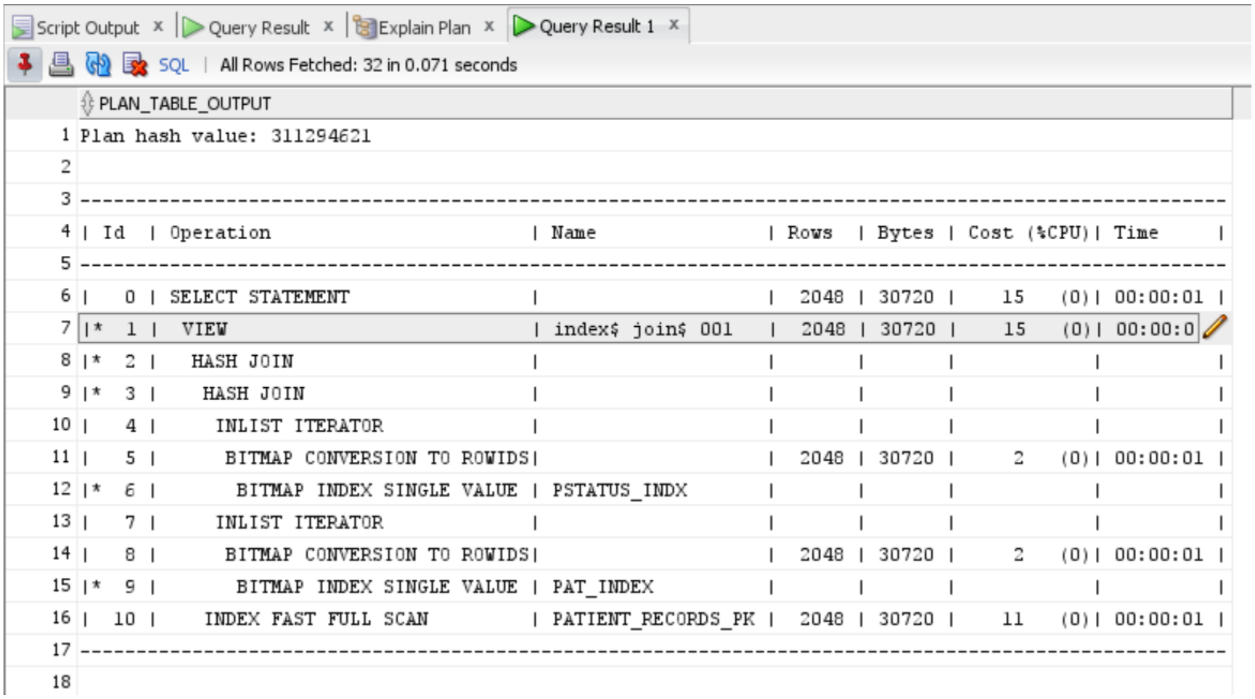
```
OPERATION                          OBJECT_NAME        OPTIONS        CARDINALITY        COST
SELECT STATEMENT                                                            2048          30
  TABLE ACCESS                     PATIENT_RECORDS    FULL                  2048          30
    Filter Predicates
      AND
        OR
          PATIENT_STATUS='Active'
          PATIENT_STATUS='Inactive'
        OR
          P_GENDER='F'
          P_GENDER='M'
  Other XML
    {info}
      info type="db_version"
        12.1.0.2
      info type="parse_schema"
```

2. To check how bitmap indexing affects the performance of the query, we **have created indexes on the gender and patient status columns.**

   CREATE BITMAP INDEX pat_index
   ON patient_records (p_gender);

   CREATE BITMAP INDEX status_index
   ON patient_records (patient_status);

3. Now we get the explain plan for the below query wherein we have **eliminated the columns like Patient first name, Patient last name**, to get the indexing applied on indexed fields.



All Rows Fetched: 32 in 0.071 seconds

```
Plan hash value: 311294621

------------------------------------------------------------------------------------------------
| Id  | Operation                      | Name               | Rows  | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT               |                    |  2048 | 30720 |    15   (0)| 00:00:01 |
|*  1 |  VIEW                          | index$ join$ 001   |  2048 | 30720 |    15   (0)| 00:00:0
|*  2 |   HASH JOIN                    |                    |       |       |            |          |
|*  3 |    HASH JOIN                   |                    |       |       |            |          |
|   4 |     INLIST ITERATOR            |                    |       |       |            |          |
|   5 |      BITMAP CONVERSION TO ROWIDS|                   |  2048 | 30720 |     2   (0)| 00:00:01 |
|*  6 |       BITMAP INDEX SINGLE VALUE | PSTATUS_INDX      |       |       |            |          |
|   7 |     INLIST ITERATOR            |                    |       |       |            |          |
|   8 |      BITMAP CONVERSION TO ROWIDS|                   |  2048 | 30720 |     2   (0)| 00:00:01 |
|*  9 |       BITMAP INDEX SINGLE VALUE | PAT_INDEX         |       |       |            |          |
|  10 |    INDEX FAST FULL SCAN        | PATIENT_RECORDS_PK |  2048 | 30720 |    11   (0)| 00:00:01 |
------------------------------------------------------------------------------------------------
```

Conclusion-When there were no indexes, there was full access to the table for all the selected fields. After adding the bit map indexes for low cardinality columns like gender, status, we could only select these indexed fields to be able to implement bitmap indexing. Hence by shrinking the selective lookup to indexed fields thereby increased the performance by reducing the query cost from 30 to 15.

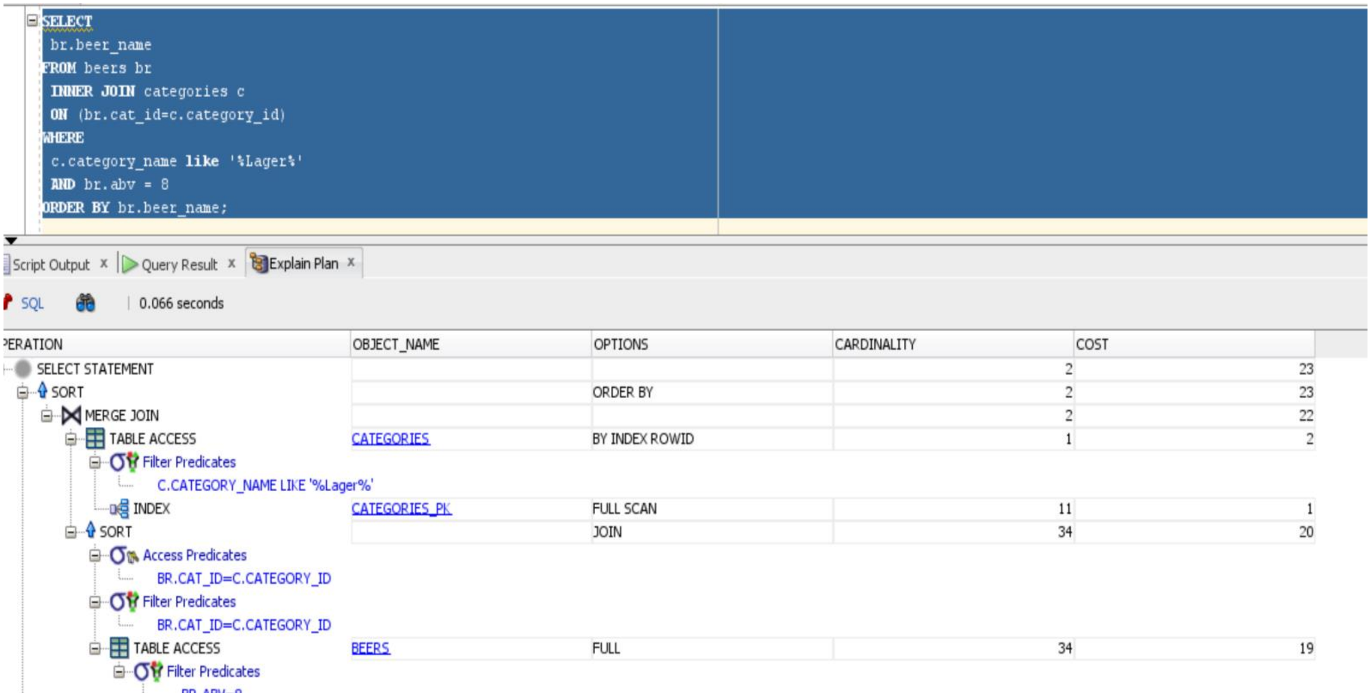
2. Idea 2: Start Simple and show that Indexing works

Let’s take two examples/queries to demonstrate this.

1. **Display all beer names that belong to a category with a name containing “Lager” somewhere in the name and have an alcohol by volume (ABV) of eight. Show the beer names in alphabetical order.**

```
SELECT
        br.beer_name
FROM
        beers br
        INNER JOIN categories c
                ON (br.cat_id=c.category_id)
WHERE c.category_name LIKE '%Lager%'
 AND br.abv = 8
ORDER BY
        br.beer_name;
```

The explain plan for this query is as shown in below:



```
SELECT
 br.beer_name
FROM beers br
 INNER JOIN categories c
 ON (br.cat_id=c.category_id)
WHERE
 c.category_name like '%Lager%'
 AND br.abv = 8
ORDER BY br.beer_name;
```

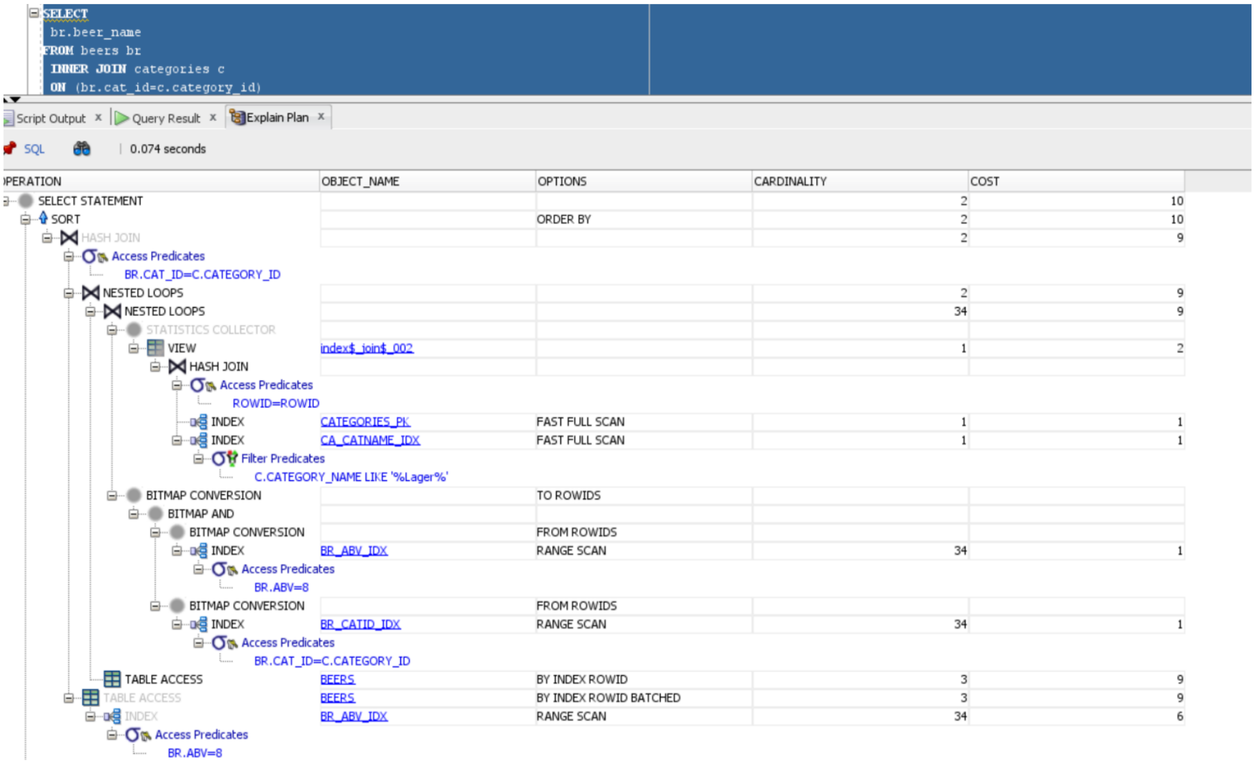
0.066 seconds

```
PERATION                                 OBJECT_NAME     OPTIONS          CARDINALITY   COST
SELECT STATEMENT                                                                    2     23
  SORT                                                   ORDER BY                   2     23
    MERGE JOIN                                                                      2     22
      TABLE ACCESS                       CATEGORIES      BY INDEX ROWID             1      2
        Filter Predicates
            C.CATEGORY_NAME LIKE '%Lager%'
        INDEX                            CATEGORIES_PK   FULL SCAN                 11      1
      SORT                                               JOIN                      34     20
        Access Predicates
            BR.CAT_ID=C.CATEGORY_ID
        Filter Predicates
            BR.CAT_ID=C.CATEGORY_ID
        TABLE ACCESS                     BEERS           FULL                      34     19
          Filter Predicates
              BR.ABV=8
```

As seen from the plan we have full access to tables. In order to remove them we **can create index and check the explain plan again**.

The following indexes were created on the table columns

```
CREATE INDEX ca_catname_idx
ON categories(category_name);

CREATE INDEX br_catid_idx
ON beers(cat_id);

CREATE INDEX br_abv_idx
ON beers(abv);
```



```
SELECT
 br.beer_name
FROM beers br
 INNER JOIN categories c
 ON (br.cat_id=c.category_id)
```

0.074 seconds

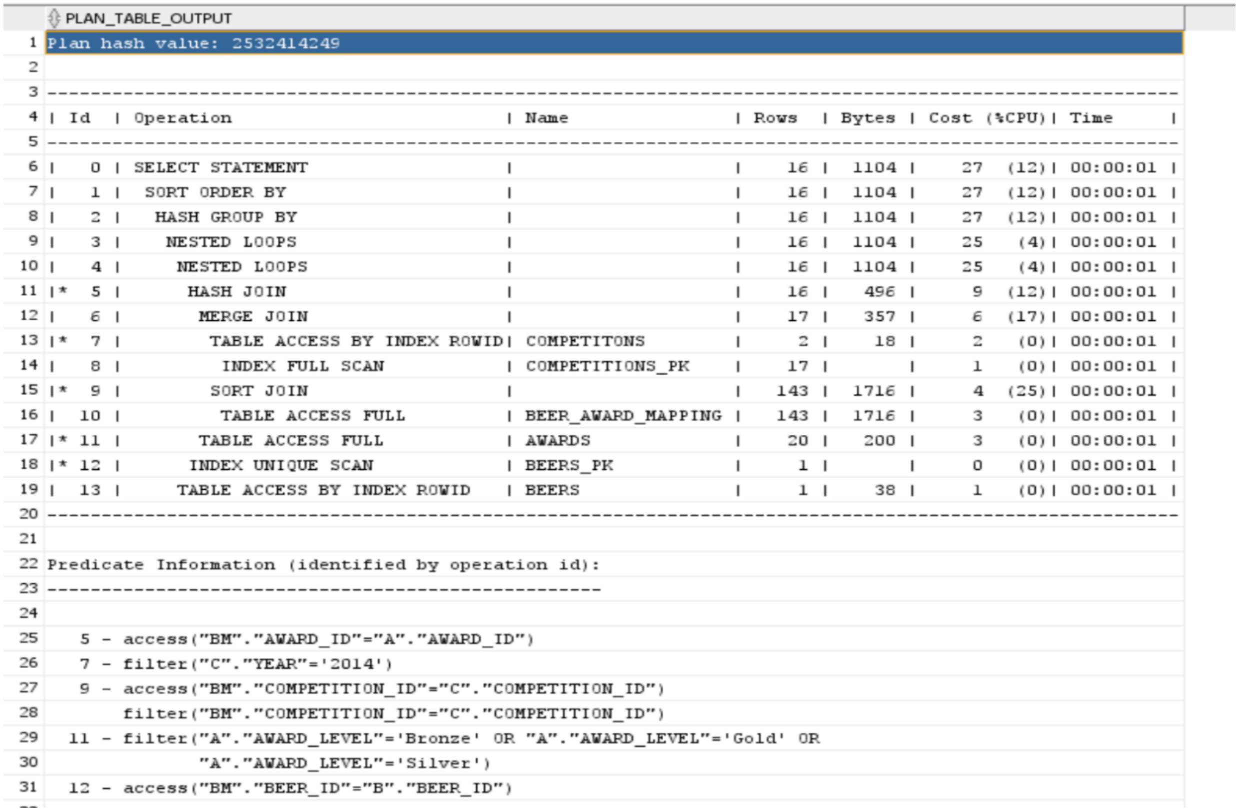
PERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			2	10
SORT		ORDER BY	2	10
HASH JOIN			2	9
Access Predicates				
BR.CAT_ID=C.CATEGORY_ID				
NESTED LOOPS			2	9
NESTED LOOPS			34	9
STATISTICS COLLECTOR				
VIEW	index$_join$_002		1	2
HASH JOIN				
Access Predicates				
ROWID=ROWID				
INDEX	CATEGORIES_PK	FAST FULL SCAN	1	1
INDEX	CA_CATNAME_IDX	FAST FULL SCAN	1	1
Filter Predicates				
C.CATEGORY_NAME LIKE '%Lager%'				
BITMAP CONVERSION		TO ROWIDS		
BITMAP AND				
BITMAP CONVERSION		FROM ROWIDS		
INDEX	BR_ABV_IDX	RANGE SCAN	34	1
Access Predicates				
BR.ABV=8				
BITMAP CONVERSION		FROM ROWIDS		
INDEX	BR_CATID_IDX	RANGE SCAN	34	1
Access Predicates				
BR.CAT_ID=C.CATEGORY_ID				
TABLE ACCESS	BEERS	BY INDEX ROWID	3	9
TABLE ACCESS	BEERS	BY INDEX ROWID BATCHED	3	9
INDEX	BR_ABV_IDX	RANGE SCAN	34	6
Access Predicates				
BR.ABV=8				

After creating the indexes and checking the explain plan, it shows that the **full access on table is removed, the cost is reduced and the query is optimized for faster retrieval of records**.

2. **Display the beers that won maximum awards in the year 2014 for different award categories. (Gold, Silver, Bronze)**

```
SELECT
        b.beer_name,
        COUNT(bm.award_id) as TOTAL_AWARDS
FROM
        beer_award_mapping bm
        INNER JOIN beers b
                ON (bm.beer_id=b.beer_id)
        INNER JOIN awards a
                ON (bm.award_id=a.award_id)
        INNER JOIN competitons c
                ON (bm.competition_id=c.competition_id)
WHERE c.year = '2014'
AND award_level IN ('Gold','Silver','Bronze')
GROUP BY
        b.beer_name
ORDER BY
        TOTAL_AWARDS desc;
```

The explain plan for the above query is

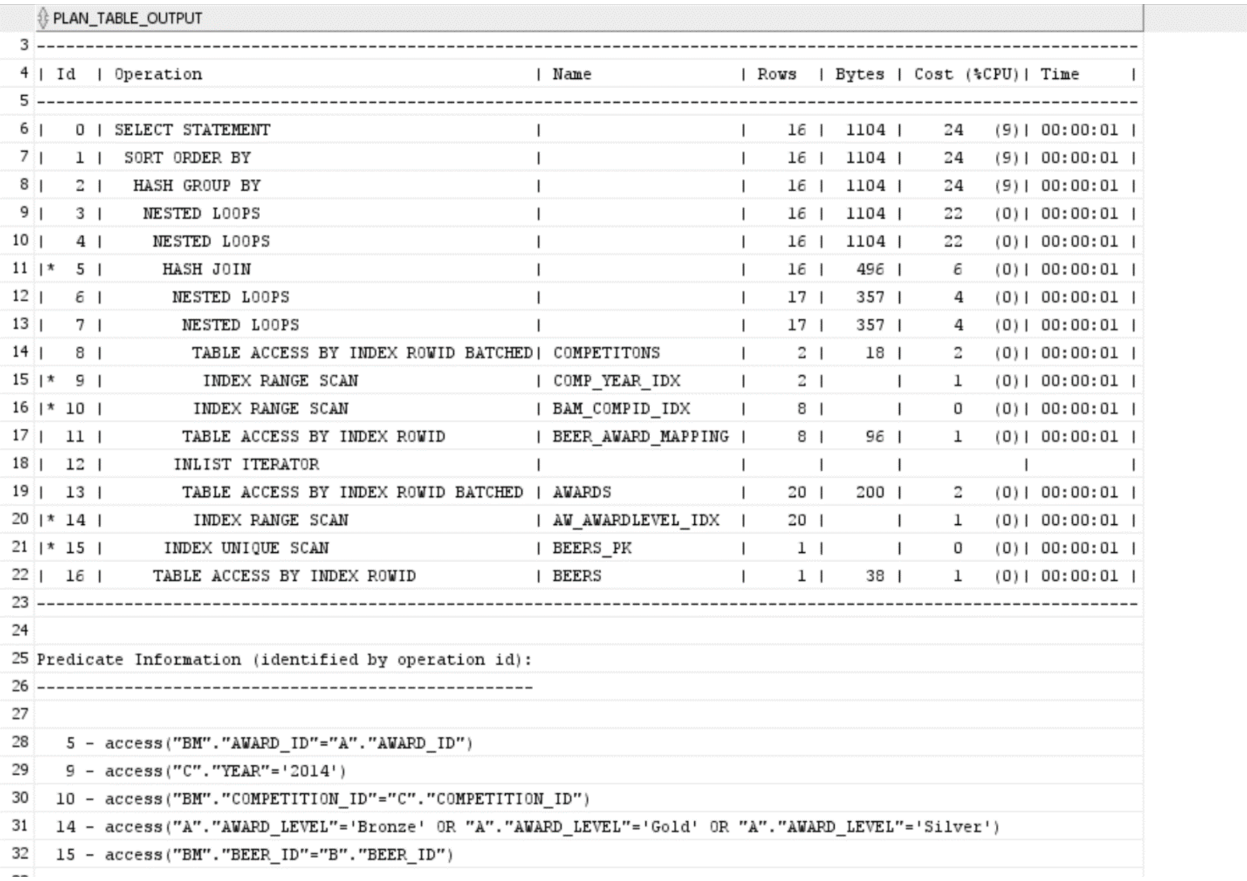


```
Plan hash value: 2532414249

--------------------------------------------------------------------------------------------------------
| Id  | Operation                         | Name               | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                  |                    |    16 |  1104 |    27  (12)| 00:00:01 |
|   1 |  SORT ORDER BY                    |                    |    16 |  1104 |    27  (12)| 00:00:01 |
|   2 |   HASH GROUP BY                   |                    |    16 |  1104 |    27  (12)| 00:00:01 |
|   3 |    NESTED LOOPS                   |                    |    16 |  1104 |    25   (4)| 00:00:01 |
|   4 |     NESTED LOOPS                  |                    |    16 |  1104 |    25   (4)| 00:00:01 |
|*  5 |      HASH JOIN                    |                    |    16 |   496 |     9  (12)| 00:00:01 |
|   6 |       MERGE JOIN                  |                    |    17 |   357 |     6  (17)| 00:00:01 |
|*  7 |        TABLE ACCESS BY INDEX ROWID| COMPETITONS        |     2 |    18 |     2   (0)| 00:00:01 |
|   8 |         INDEX FULL SCAN           | COMPETITIONS_PK    |    17 |       |     1   (0)| 00:00:01 |
|*  9 |        SORT JOIN                  |                    |   143 |  1716 |     4  (25)| 00:00:01 |
|  10 |         TABLE ACCESS FULL         | BEER_AWARD_MAPPING |   143 |  1716 |     3   (0)| 00:00:01 |
|* 11 |       TABLE ACCESS FULL           | AWARDS             |    20 |   200 |     3   (0)| 00:00:01 |
|* 12 |      INDEX UNIQUE SCAN            | BEERS_PK           |     1 |       |     0   (0)| 00:00:01 |
|  13 |     TABLE ACCESS BY INDEX ROWID   | BEERS              |     1 |    38 |     1   (0)| 00:00:01 |
--------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   5 - access("BM"."AWARD_ID"="A"."AWARD_ID")
   7 - filter("C"."YEAR"='2014')
   9 - access("BM"."COMPETITION_ID"="C"."COMPETITION_ID")
       filter("BM"."COMPETITION_ID"="C"."COMPETITION_ID")
  11 - filter("A"."AWARD_LEVEL"='Bronze' OR "A"."AWARD_LEVEL"='Gold' OR
              "A"."AWARD_LEVEL"='Silver')
  12 - access("BM"."BEER_ID"="B"."BEER_ID")
```

As seen from the plan we have full access to tables. In order to remove them we **can create index and check the explain plan again**.

The following indexes were created on the table columns

> **CREATE INDEX** bam_compid_idx
> **ON** beer_award_mapping(competition_id);
>
> **CREATE INDEX** aw_awardlevel_idx
> **ON** awards(award_level);
>
> **CREATE INDEX** comp_year_idx
> **ON** competitons(year);



```
-----------------------------------------------------------------------------------------------------------------
| Id  | Operation                                  | Name               | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                           |                    |    16 |  1104 |    24   (9)| 00:00:01 |
|   1 |  SORT ORDER BY                             |                    |    16 |  1104 |    24   (9)| 00:00:01 |
|   2 |   HASH GROUP BY                            |                    |    16 |  1104 |    24   (9)| 00:00:01 |
|   3 |    NESTED LOOPS                            |                    |    16 |  1104 |    22   (0)| 00:00:01 |
|   4 |     NESTED LOOPS                           |                    |    16 |  1104 |    22   (0)| 00:00:01 |
|*  5 |      HASH JOIN                             |                    |    16 |   496 |     6   (0)| 00:00:01 |
|   6 |       NESTED LOOPS                         |                    |    17 |   357 |     4   (0)| 00:00:01 |
|   7 |        NESTED LOOPS                        |                    |    17 |   357 |     4   (0)| 00:00:01 |
|   8 |         TABLE ACCESS BY INDEX ROWID BATCHED| COMPETITONS        |     2 |    18 |     2   (0)| 00:00:01 |
|*  9 |          INDEX RANGE SCAN                  | COMP_YEAR_IDX      |     2 |       |     1   (0)| 00:00:01 |
|* 10 |         INDEX RANGE SCAN                   | BAM_COMPID_IDX     |     8 |       |     0   (0)| 00:00:01 |
|  11 |        TABLE ACCESS BY INDEX ROWID         | BEER_AWARD_MAPPING |     8 |    96 |     1   (0)| 00:00:01 |
|  12 |       INLIST ITERATOR                      |                    |       |       |            |          |
|  13 |        TABLE ACCESS BY INDEX ROWID BATCHED | AWARDS             |    20 |   200 |     2   (0)| 00:00:01 |
|* 14 |         INDEX RANGE SCAN                   | AW_AWARDLEVEL_IDX  |    20 |       |     1   (0)| 00:00:01 |
|* 15 |      INDEX UNIQUE SCAN                     | BEERS_PK           |     1 |       |     0   (0)| 00:00:01 |
|  16 |     TABLE ACCESS BY INDEX ROWID            | BEERS              |     1 |    38 |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   5 - access("BM"."AWARD_ID"="A"."AWARD_ID")
   9 - access("C"."YEAR"='2014')
  10 - access("BM"."COMPETITION_ID"="C"."COMPETITION_ID")
  14 - access("A"."AWARD_LEVEL"='Bronze' OR "A"."AWARD_LEVEL"='Gold' OR "A"."AWARD_LEVEL"='Silver')
  15 - access("BM"."BEER_ID"="B"."BEER_ID")
```

After creating the indexes and checking the explain plan, it **shows full access on table is removed and the cost is reduced and the query is optimized for faster retrieval of records**.

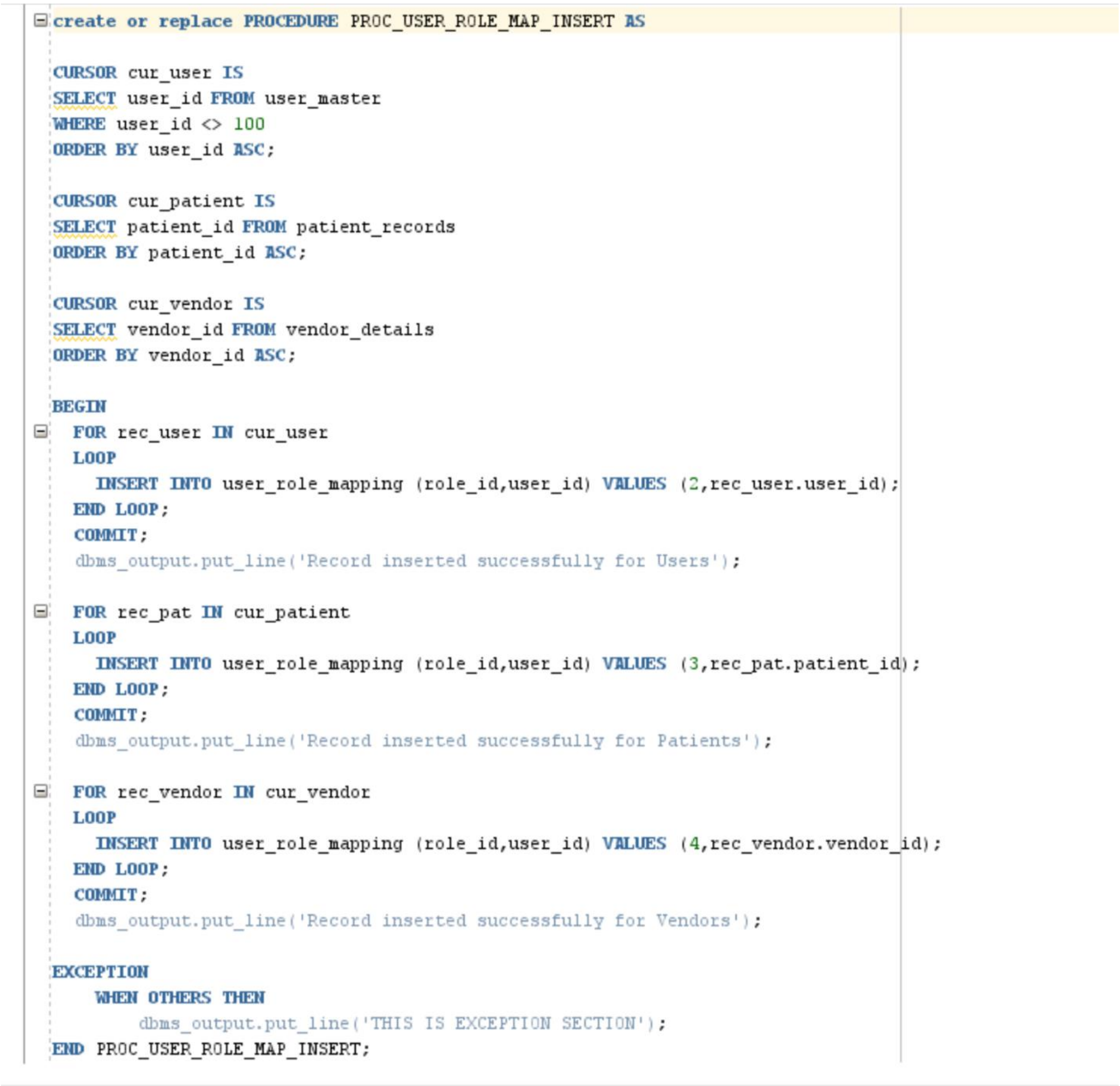
<u>Conclusion:</u> When there were no indexes on the table columns, full access took place until it fetched the required data. Since these tables have large data sets, selection on full tables impact the performance. Adding a B-tree index on these table columns eliminates the full table access and reduces the cost of the query resulting in faster retrieval of records.

3. Idea 6: Database programming

A stored procedure is a group of SQL statements that can be created and stored in database. It can be shared by multiple programs and can be reused. Stored procedures may return result sets and can be processed using cursors, they may contain variables for processing data, they may be also used for data generation or manipulation.

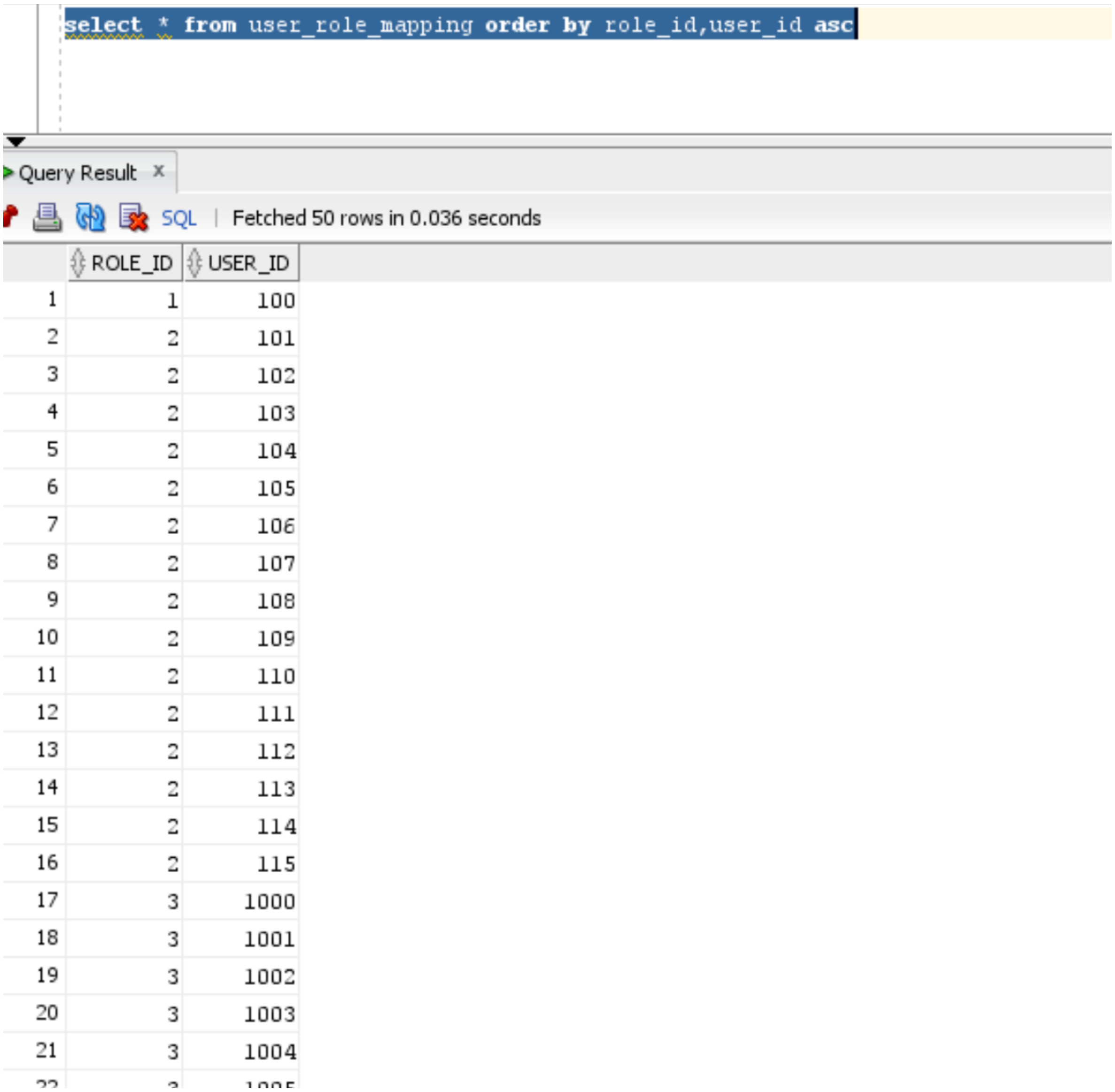
For demonstrating this,

1. Let’s **create a simple procedure that will generate data** for a table **fetching few values from other tables using a cursor.**



```
create or replace PROCEDURE PROC_USER_ROLE_MAP_INSERT AS

CURSOR cur_user IS
SELECT user_id FROM user_master
WHERE user_id <> 100
ORDER BY user_id ASC;

CURSOR cur_patient IS
SELECT patient_id FROM patient_records
ORDER BY patient_id ASC;

CURSOR cur_vendor IS
SELECT vendor_id FROM vendor_details
ORDER BY vendor_id ASC;

BEGIN
  FOR rec_user IN cur_user
  LOOP
    INSERT INTO user_role_mapping (role_id,user_id) VALUES (2,rec_user.user_id);
  END LOOP;
  COMMIT;
  dbms_output.put_line('Record inserted successfully for Users');

  FOR rec_pat IN cur_patient
  LOOP
    INSERT INTO user_role_mapping (role_id,user_id) VALUES (3,rec_pat.patient_id);
  END LOOP;
  COMMIT;
  dbms_output.put_line('Record inserted successfully for Patients');

  FOR rec_vendor IN cur_vendor
  LOOP
    INSERT INTO user_role_mapping (role_id,user_id) VALUES (4,rec_vendor.vendor_id);
  END LOOP;
  COMMIT;
  dbms_output.put_line('Record inserted successfully for Vendors');

EXCEPTION
    WHEN OTHERS THEN
        dbms_output.put_line('THIS IS EXCEPTION SECTION');
END PROC_USER_ROLE_MAP_INSERT;
```

After executing the stored procedure **PROC_USER_ROLE_MAP_INSERT,** the above data is generated for **USER_ROLE_MAPPING** table by fetching few details from **USER_MASTER**, **PATIENT_RECORDS** and **VENDOR_DETAILS** tables using **cursors**.



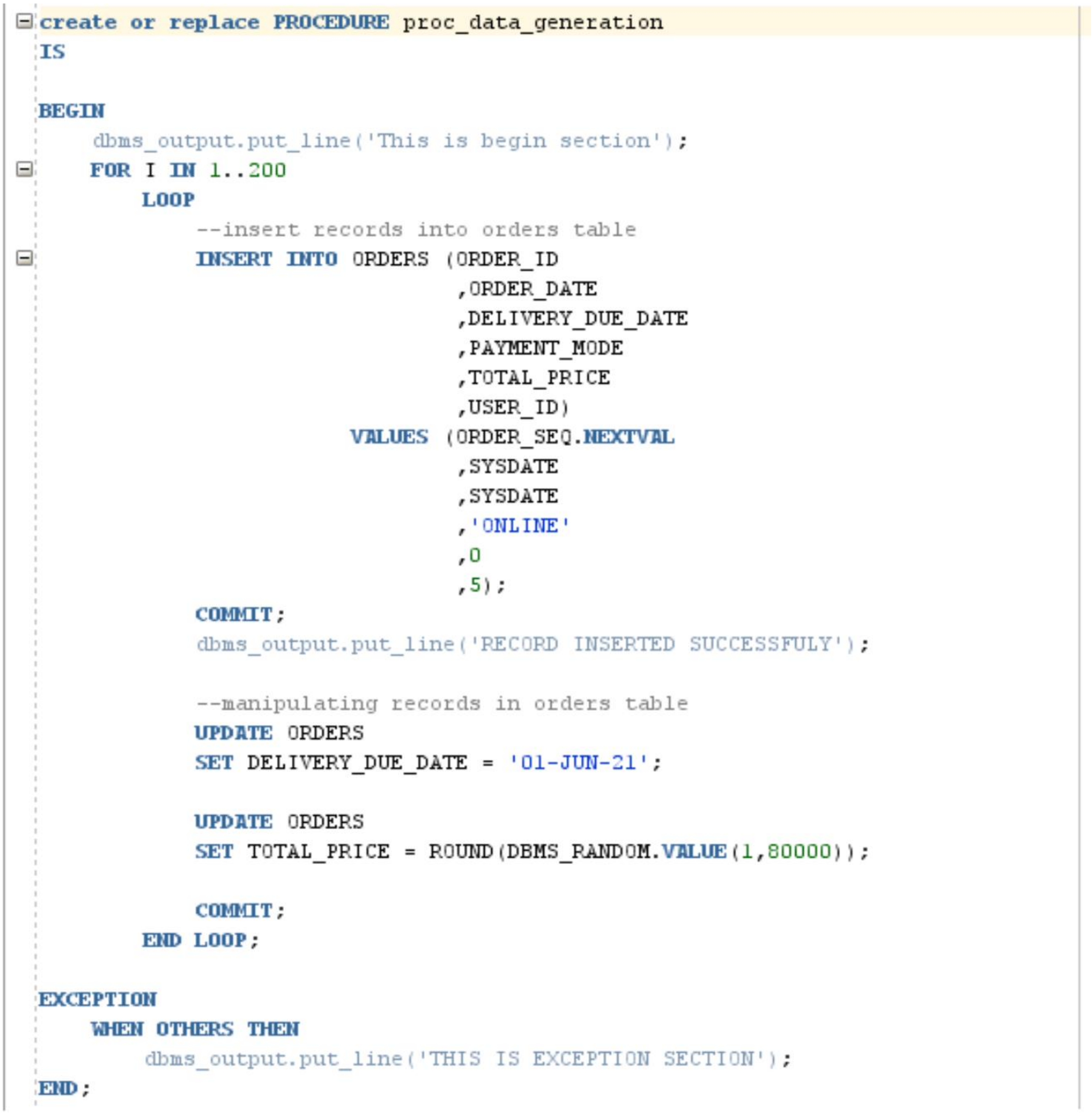
```
select * from user_role_mapping order by role_id,user_id asc
```

Fetched 50 rows in 0.036 seconds

	ROLE_ID	USER_ID
1	1	100
2	2	101
3	2	102
4	2	103
5	2	104
6	2	105
7	2	106
8	2	107
9	2	108
10	2	109
11	2	110
12	2	111
13	2	112
14	2	113
15	2	114
16	2	115
17	3	1000
18	3	1001
19	3	1002
20	3	1003
21	3	1004
22	3	1005

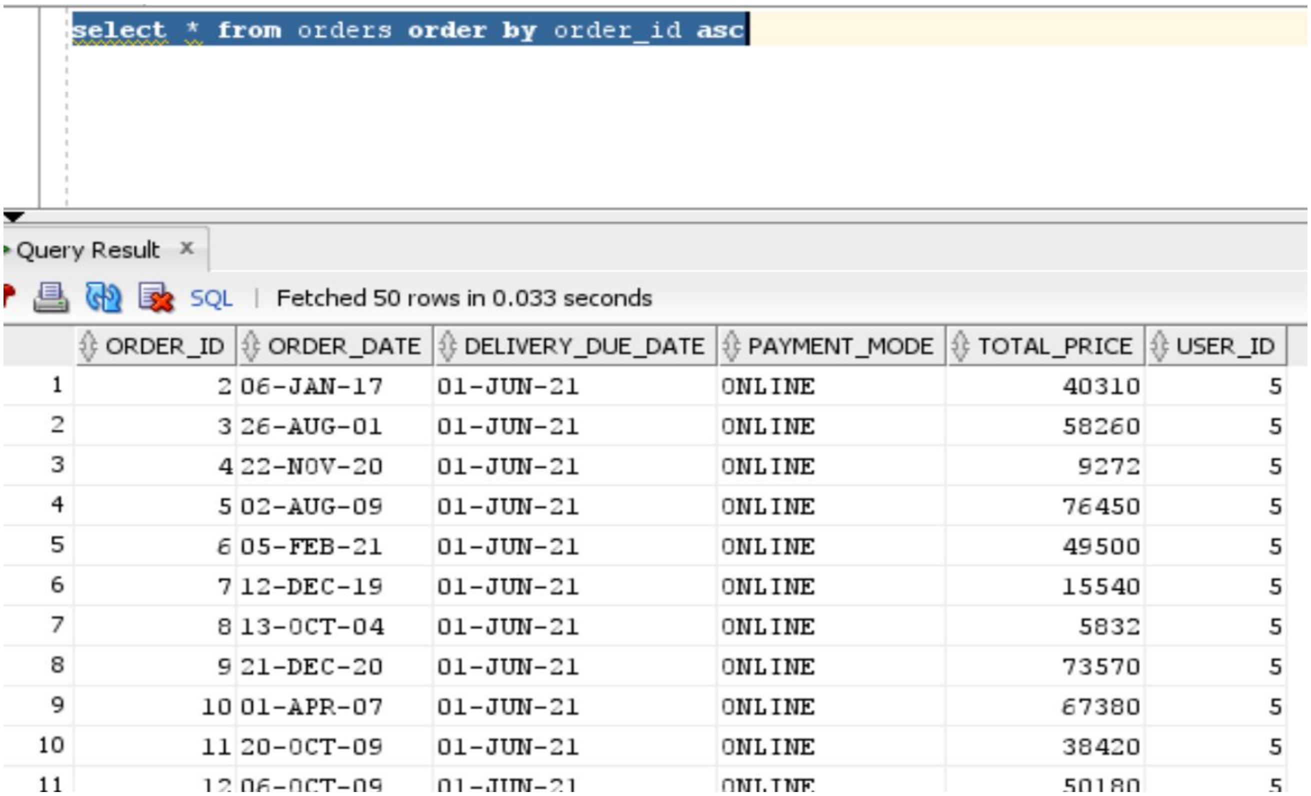
After executing the select query from user_role_mapping table, we can see records in the table which are inserted by the procedure.

2. Let’s **create a simple procedure that will generate data for a table** and after generating records will also **update few columns of the same table.**



```
create or replace PROCEDURE proc_data_generation
IS

BEGIN
    dbms_output.put_line('This is begin section');
    FOR I IN 1..200
        LOOP
            --insert records into orders table
            INSERT INTO ORDERS (ORDER_ID
                               ,ORDER_DATE
                               ,DELIVERY_DUE_DATE
                               ,PAYMENT_MODE
                               ,TOTAL_PRICE
                               ,USER_ID)
                        VALUES (ORDER_SEQ.NEXTVAL
                               ,SYSDATE
                               ,SYSDATE
                               ,'ONLINE'
                               ,0
                               ,5);
            COMMIT;
            dbms_output.put_line('RECORD INSERTED SUCCESSFULY');

            --manipulating records in orders table
            UPDATE ORDERS
            SET DELIVERY_DUE_DATE = '01-JUN-21';

            UPDATE ORDERS
            SET TOTAL_PRICE = ROUND(DBMS_RANDOM.VALUE(1,80000));

            COMMIT;
        END LOOP;

EXCEPTION
    WHEN OTHERS THEN
        dbms_output.put_line('THIS IS EXCEPTION SECTION');
END;
```

After executing the stored procedure **PROC_DATA_GENERATION**, data is generated for **ORDERS** table. Once the default data is inserted into the table, we manipulate the two columns Delivery_due_Date and Total_Price by writing the update statement inside the stored procedure.



```
select * from orders order by order_id asc
```

	ORDER_ID	ORDER_DATE	DELIVERY_DUE_DATE	PAYMENT_MODE	TOTAL_PRICE	USER_ID
1	2	06-JAN-17	01-JUN-21	ONLINE	40310	5
2	3	26-AUG-01	01-JUN-21	ONLINE	58260	5
3	4	22-NOV-20	01-JUN-21	ONLINE	9272	5
4	5	02-AUG-09	01-JUN-21	ONLINE	76450	5
5	6	05-FEB-21	01-JUN-21	ONLINE	49500	5
6	7	12-DEC-19	01-JUN-21	ONLINE	15540	5
7	8	13-OCT-04	01-JUN-21	ONLINE	5832	5
8	9	21-DEC-20	01-JUN-21	ONLINE	73570	5
9	10	01-APR-07	01-JUN-21	ONLINE	67380	5
10	11	20-OCT-09	01-JUN-21	ONLINE	38420	5
11	12	06-OCT-09	01-JUN-21	ONLINE	50180	5

After executing the select query from orders table we can see records in the table which are inserted and updated by the procedure.

Conclusion: Stored procedures can be created to perform various operations like data generation, manipulation, computation, and many more operations in the database that can be reused and shared by multiple programs/applications.