INFS2200 PROJECT ASSIGNMENT

Student ID: 45363809

Name: Minjae Lee

Due Date: 11:59PM, 28 October 2021

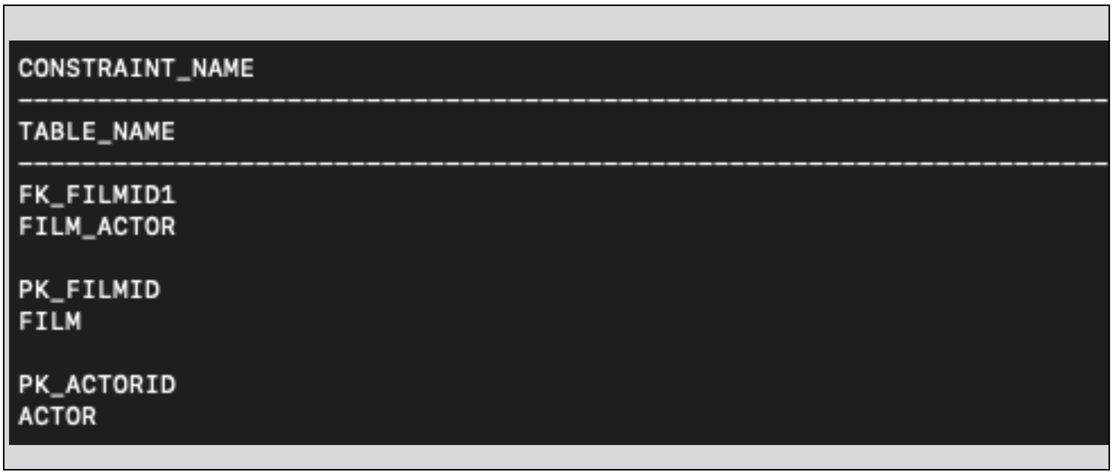
TASK 1 - CONSTRAINTS

1.

SQL statement:

```
SELECT CONSTRAINT_NAME, TABLE_NAME FROM USER_CONSTRAINTS
WHERE TABLE_NAME IN ('FILM_CATEGORY', 'CATEGORY', 'FILM',
'FILM_ACTOR', 'LANGUAGE', 'ACTOR');
```

The list of constraints obtained using the above query is displayed below (excluding Oracle generated indexes):



```
CONSTRAINT_NAME
--------------------------------------------------------------------
TABLE_NAME
--------------------------------------------------------------------
FK_FILMID1
FILM_ACTOR

PK_FILMID
FILM

PK_ACTORID
ACTOR
```

Table FILM_ACTOR has a foreign key constraint `FK_FILMID1`.

Table FILM has a primary key constraint `PK_FILMID`.

Table ACTOR has a primary key constraint `PK_ACTORID`.

2.

Missing Constraints:

Constraint No.	SQL
2	ALTER TABLE CATEGORY ADD CONSTRAINT PK_CATEGORYID PRIMARY KEY (CATEGORY_ID);
4	ALTER TABLE LANGUAGE ADD CONSTRAINT PK_LANGUAGEID PRIMARY KEY (LANGUAGE_ID);
5	ALTER TABLE FILM ADD CONSTRAINT UN_DESCRIPTION UNIQUE (DESCRIPTION);
6	ALTER TABLE ACTOR ADD CONSTRAINT CK_FNAME CHECK (FIRST_NAME IS NOT NULL);
7	ALTER TABLE ACTOR ADD CONSTRAINT CK_LNAME CHECK (LAST_NAME IS NOT NULL);
8	ALTER TABLE CATEGORY ADD CONSTRAINT CK_CATNAME CHECK (NAME IS NOT NULL);
9	ALTER TABLE LANGUAGE ADD CONSTRAINT CK_LANNAME CHECK (NAME IS NOT NULL);
10	ALTER TABLE FILM ADD CONSTRAINT CK_TITLE CHECK (TITLE IS NOT NULL);
11	ALTER TABLE FILM ADD CONSTRAINT CK_RELEASEYR CHECK (RELEASE_YEAR <= 2020);
12	ALTER TABLE FILM ADD CONSTRAINT CK_RATING CHECK (RATING IN ('G', 'PG', 'PG-13', 'R', 'NC-17'));
13	ALTER TABLE FILM ADD CONSTRAINT CK_SPLFEATURES CHECK (SPECIAL_FEATURES = NULL OR SPECIAL_FEATURES IN ('Trailers', 'Commentaries', 'Deleted Scenes', 'Behind the Scenes'));
14	ALTER TABLE FILM ADD CONSTRAINT FK_LANGUAGEID FOREIGN KEY (LANGUAGE_ID) REFERENCES LANGUAGE (LANGUAGE_ID);
15	ALTER TABLE FILM ADD CONSTRAINT FK_ORLANGUAGEID FOREIGN KEY (ORIGINAL_LANGUAGE_ID) REFERENCES LANGUAGE (LANGUAGE_ID);
16	ALTER TABLE FILM_ACTOR ADD CONSTRAINT FK_ACTORID FOREIGN KEY (ACTOR_ID) REFERENCES ACTOR (ACTOR_ID);
17	ALTER TABLE FILM_CATEGORY ADD CONSTRAINT FK_CATEGORYID FOREIGN KEY (CATEGORY_ID) REFERENCES CATEGORY (CATEGORY_ID);
19	ALTER TABLE FILM_CATEGORY ADD CONSTRAINT FK_FILMID2 FOREIGN KEY (FILM_ID) REFERENCES FILM (FILM_ID);

TASK 2 - TRIGGERS

1.

Sequence Object:

```
CREATE SEQUENCE "FILM_ID_SEQ" MINVALUE 20010 MAXVALUE
99999999990 INCREMENT BY 10 START WITH 20010;
```

2.

Trigger:

```
CREATE OR REPLACE TRIGGER "BI_FILM_ID"
BEFORE INSERT ON "FILM"
FOR EACH ROW
BEGIN
    SELECT "FILM_ID_SEQ".NEXTVAL INTO :NEW.FILM_ID FROM
DUAL;
END;
/
```

3.

Trigger:

```
CREATE OR REPLACE TRIGGER "BI_FILM_DESP"
BEFORE INSERT ON "FILM"
FOR EACH ROW
DECLARE
    og_lang VARCHAR(20);
    lang VARCHAR(20);
    temp_seq INTEGER;
BEGIN
IF (:NEW.ORIGINAL_LANGUAGE_ID IS NOT NULL
AND :NEW.LANGUAGE_ID IS NOT NULL AND :NEW.RATING IS NOT
NULL)
    THEN
        SELECT NAME INTO og_lang FROM LANGUAGE
WHERE :NEW.ORIGINAL_LANGUAGE_ID = LANGUAGE.LANGUAGE_ID;
        SELECT NAME INTO lang FROM LANGUAGE
WHERE :NEW.LANGUAGE_ID = LANGUAGE.LANGUAGE_ID;
        SELECT count(*) INTO temp_seq from FILM WHERE RATING
= :NEW.RATING;
        :NEW.DESCRIPTION := CONCAT (:NEW.DESCRIPTION,
CONCAT(:NEW.RATING, CONCAT('-', CONCAT(temp_seq + 1, ':
')))); 
        :NEW.DESCRIPTION := CONCAT (:NEW.DESCRIPTIO  2  N,
CONCAT('Originally in ', CONCAT(og_lang, '. ')));
        :NEW.DESCRIPTION := CONCAT (:NEW.DESCRIPTION,
CONCAT('Re-released in ', lang));
    END IF;

END;
/
```

TASK 3 - VIEWS

1.

```
SELECT TITLE, LENGTH
FROM FILM F, FILM_CATEGORY CF, CATEGORY C
WHERE F.LENGTH = (SELECT MIN(LENGTH) FROM FILM)
AND (F.FILM_ID = CF.FILM_ID AND CF.CATEGORY_ID =
C.CATEGORY_ID)
AND C.NAME = 'Action';
```

2.

```
CREATE VIEW MIN_ACTION_ACTORS AS
SELECT DISTINCT A.ACTOR_ID, A.FIRST_NAME, A.LAST_NAME
FROM FILM_ACTOR FA, ACTOR A
WHERE FA.FILM_ID IN (SELECT F.FILM_ID
    FROM FILM F, FILM_CATEGORY CF, CATEGORY C
    WHERE F.LENGTH = (SELECT MIN(LENGTH) FROM FILM)
    AND (F.FILM_ID = CF.FILM_ID AND CF.CATEGORY_ID =
C.CATEGORY_ID)
    AND C.NAME = 'Action')
    AND A.ACTOR_ID = FA.ACTOR_ID;
```

3.

```
CREATE VIEW V_ACTION_ACTORS_2012 AS
SELECT DISTINCT A.ACTOR_ID, A.FIRST_NAME, A.LAST_NAME
FROM FILM_ACTOR FA, ACTOR A
WHERE FA.FILM_ID IN (SELECT F.FILM_ID
    FROM FILM F, FILM_CATEGORY CF, CATEGORY C
    WHERE F.RELEASE_YEAR = '2012'
    AND (F.FILM_ID = CF.FILM_ID AND CF.CATEGORY_ID =
C.CATEGORY_ID)
    AND C.NAME = 'Action')
    AND A.ACTOR_ID = FA.ACTOR_ID;
```

4.

```
CREATE MATERIALIZED VIEW MV_ACTION_ACTORS_2012
BUILD IMMEDIATE AS
SELECT DISTINCT A.ACTOR_ID, A.FIRST_NAME, A.LAST_NAME
FROM FILM_ACTOR FA, ACTOR A
WHERE FA.FILM_ID IN (SELECT F.FILM_ID
    FROM FILM F, FILM_CATEGORY CF, CATEGORY C
    WHERE  2   F.RELEASE_YEAR = '2012'
    AND (F.FILM_ID = CF.FILM_ID AND CF.CATEGORY_ID =
C.CATEGORY_ID)
    AND C.NAME = 'Action')
    AND A.ACTOR_ID = FA.ACTOR_ID;
```

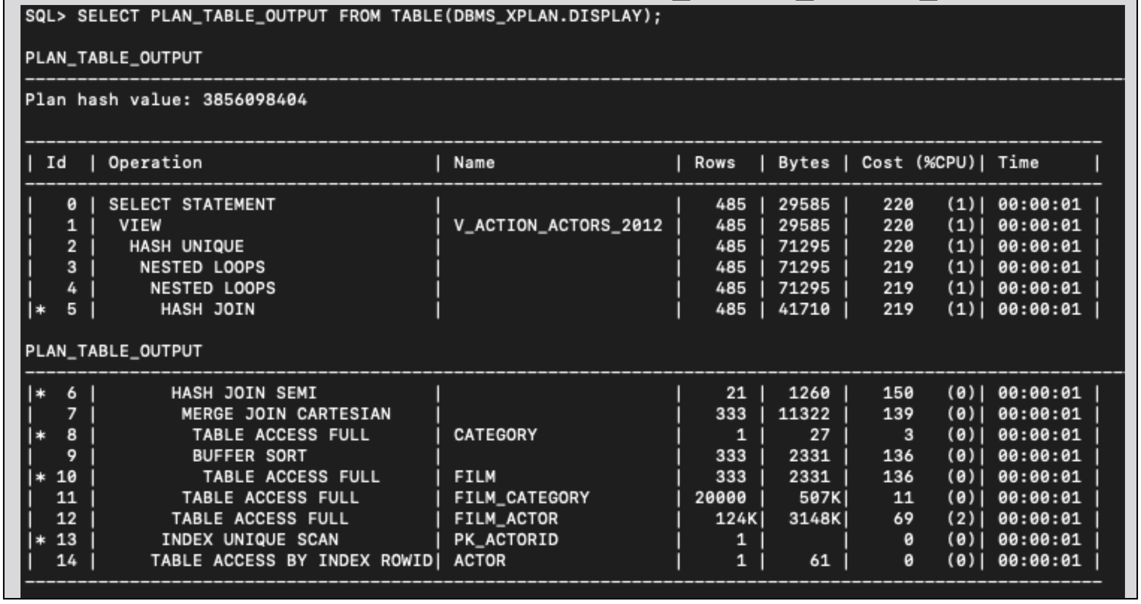
5.

Reported query execution time for V_ACTION_ACTORS_2012 and MV_ACTION_ACTORS_2012:

```
SQL> SET TIMING ON;

SQL> SELECT * FROM V_ACTION_ACTORS_2012;
Elapsed: 00:00:00.24

SQL> SELECT * FROM MV_ACTION_ACTORS_2012;
Elapsed: 00:00:00.08
```

Reported Execution Plan for V_ACTION_ACTORS_2012:

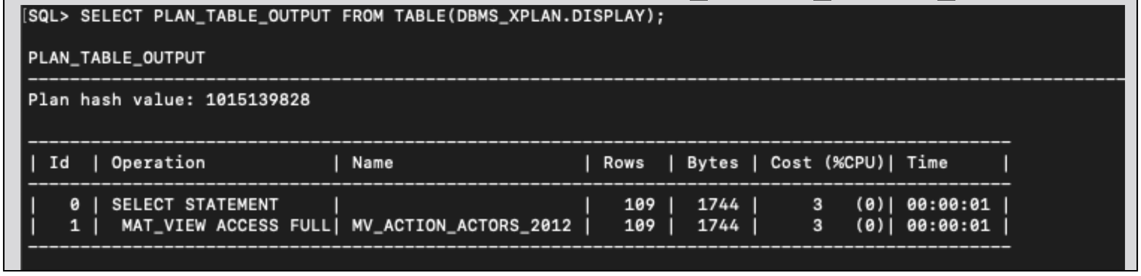
```
SQL> EXPLAIN PLAN FOR SELECT * FROM V_ACTION_ACTORS_2012;
```



```
SQL> SELECT PLAN_TABLE_OUTPUT FROM TABLE(DBMS_XPLAN.DISPLAY);

PLAN_TABLE_OUTPUT
----------------------------------------------------------------------------------------------------
Plan hash value: 3856098404

-----------------------------------------------------------------------------------------------
| Id  | Operation                     | Name                 | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                      |   485 | 29585 |   220   (1)| 00:00:01 |
|   1 |  VIEW                         | V_ACTION_ACTORS_2012 |   485 | 29585 |   220   (1)| 00:00:01 |
|   2 |   HASH UNIQUE                 |                      |   485 | 71295 |   220   (1)| 00:00:01 |
|   3 |    NESTED LOOPS               |                      |   485 | 71295 |   219   (1)| 00:00:01 |
|   4 |     NESTED LOOPS              |                      |   485 | 71295 |   219   (1)| 00:00:01 |
|*  5 |      HASH JOIN                |                      |   485 | 41710 |   219   (1)| 00:00:01 |

PLAN_TABLE_OUTPUT
----------------------------------------------------------------------------------------------------
|*  6 |       HASH JOIN SEMI          |                      |    21 |  1260 |   150   (0)| 00:00:01 |
|   7 |        MERGE JOIN CARTESIAN   |                      |   333 | 11322 |   139   (0)| 00:00:01 |
|*  8 |         TABLE ACCESS FULL     | CATEGORY             |     1 |    27 |     3   (0)| 00:00:01 |
|   9 |         BUFFER SORT           |                      |   333 |  2331 |   136   (0)| 00:00:01 |
|* 10 |          TABLE ACCESS FULL    | FILM                 |   333 |  2331 |   136   (0)| 00:00:01 |
|  11 |        TABLE ACCESS FULL      | FILM_CATEGORY        | 20000 |   507K|    11   (0)| 00:00:01 |
|  12 |       TABLE ACCESS FULL       | FILM_ACTOR           |   124K|  3148K|    69   (2)| 00:00:01 |
|* 13 |      INDEX UNIQUE SCAN        | PK_ACTORID           |     1 |       |     0   (0)| 00:00:01 |
|  14 |     TABLE ACCESS BY INDEX ROWID| ACTOR               |     1 |    61 |     0   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------
```

Reported Execution Plan for MV_ACTION_ACTORS_2012:

```
SQL> EXPLAIN PLAN FOR SELECT * FROM MV_ACTION_ACTORS_2012;
```



```
SQL> SELECT PLAN_TABLE_OUTPUT FROM TABLE(DBMS_XPLAN.DISPLAY);

PLAN_TABLE_OUTPUT
----------------------------------------------------------------------------------------------------
Plan hash value: 1015139828

---------------------------------------------------------------------------------------------
| Id  | Operation            | Name                  | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |                       |   109 |  1744 |     3   (0)| 00:00:01 |
|   1 |  MAT_VIEW ACCESS FULL| MV_ACTION_ACTORS_2012 |   109 |  1744 |     3   (0)| 00:00:01 |
---------------------------------------------------------------------------------------------
```

As can be seen above, query modification is conducted on the virtual view whereas the query is directly executed on the materialised view which in turn reduces the query time significantly. This is the advantage of a materialised view having a pre-calculated data stored in the view in comparison to a regular view.

TASK 4 - INDEXES

1.

```
SELECT *
FROM FILM
WHERE INSTR(DESCRIPTION, 'boat') > 0
ORDER BY TITLE ASC
FETCH NEXT 100 ROWS ONLY;
```

2.

```
CREATE INDEX IDX_BOAT ON FILM (INSTR(DESCRIPTION, ‘Boat’));
```

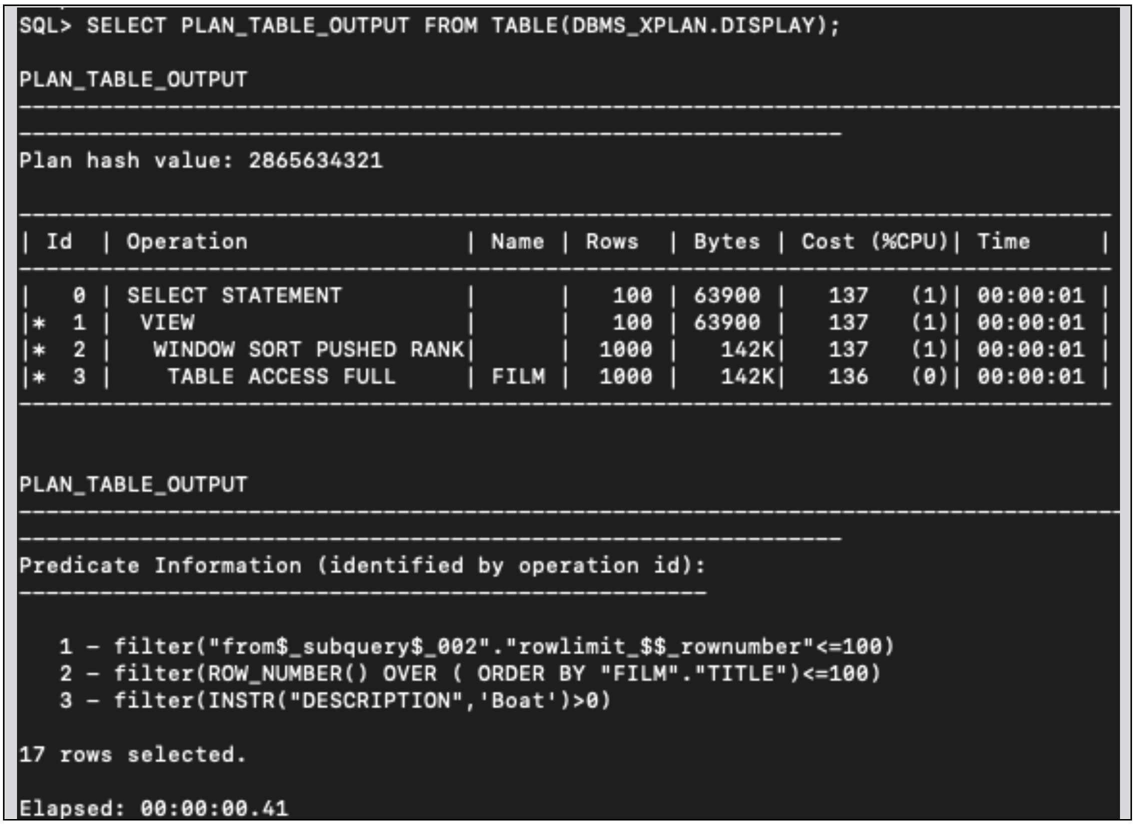
Simply creating an index on, say TITLE, would not improve the performance of the query as the search condition is based on the function of INSTR of the DESCRIPTION column. To improve the performance of this query, a function-based indexing is required as shown above.

3.

Setup Execution Plan:

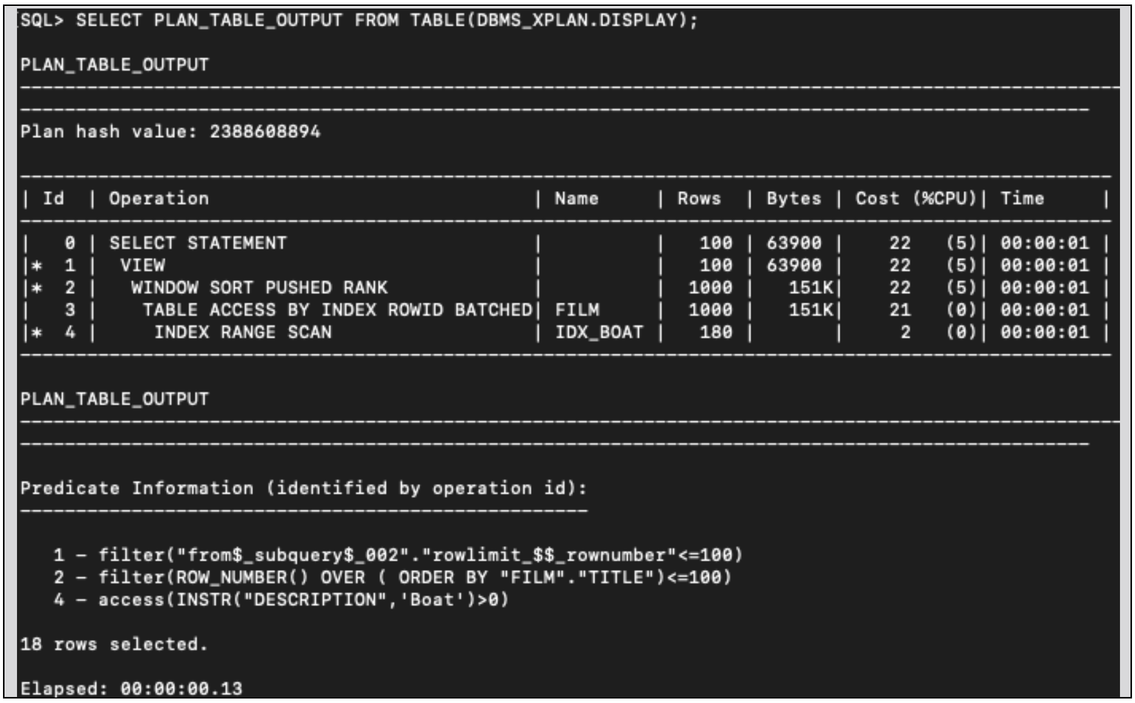
```
EXPLAIN PLAN FOR
SELECT *
FROM FILM
WHERE INSTR(DESCRIPTION, 'Boat') > 0
ORDER BY TITLE ASC
FETCH NEXT 100 ROWS ONLY;
```

Reported execution plan and time BEFORE index is created:



```
SQL> SELECT PLAN_TABLE_OUTPUT FROM TABLE(DBMS_XPLAN.DISPLAY);

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------
--------------------------------------------------------------
Plan hash value: 2865634321

-----------------------------------------------------------------------------------
| Id  | Operation                   | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |      |   100 | 63900 |   137   (1)| 00:00:01 |
|*  1 |  VIEW                       |      |   100 | 63900 |   137   (1)| 00:00:01 |
|*  2 |   WINDOW SORT PUSHED RANK|      |  1000 |   142K|   137   (1)| 00:00:01 |
|*  3 |    TABLE ACCESS FULL        | FILM |  1000 |   142K|   136   (0)| 00:00:01 |
-----------------------------------------------------------------------------------


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


   1 - filter("from$_subquery$_002"."rowlimit_$$_rownumber"<=100)
   2 - filter(ROW_NUMBER() OVER ( ORDER BY "FILM"."TITLE")<=100)
   3 - filter(INSTR("DESCRIPTION",'Boat')>0)

17 rows selected.

Elapsed: 00:00:00.41
```

Reported execution plan and time AFTER indexing:



```
SQL> SELECT PLAN_TABLE_OUTPUT FROM TABLE(DBMS_XPLAN.DISPLAY);

PLAN_TABLE_OUTPUT
----------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------
Plan hash value: 2388608894

--------------------------------------------------------------------------------------------------
| Id  | Operation                             | Name     | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                      |          |   100 | 63900 |    22   (5)| 00:00:01 |
|*  1 |  VIEW                                 |          |   100 | 63900 |    22   (5)| 00:00:01 |
|*  2 |   WINDOW SORT PUSHED RANK             |          |  1000 |   151K|    22   (5)| 00:00:01 |
|   3 |    TABLE ACCESS BY INDEX ROWID BATCHED| FILM     |  1000 |   151K|    21   (0)| 00:00:01 |
|*  4 |     INDEX RANGE SCAN                  | IDX_BOAT |   180 |       |     2   (0)| 00:00:01 |
--------------------------------------------------------------------------------------------------

PLAN_TABLE_OUTPUT
----------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------

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

   1 - filter("from$_subquery$_002"."rowlimit_$$_rownumber"<=100)
   2 - filter(ROW_NUMBER() OVER ( ORDER BY "FILM"."TITLE")<=100)
   4 - access(INSTR("DESCRIPTION",'Boat')>0)

18 rows selected.

Elapsed: 00:00:00.13
```

Both execution time and the cost of execution improved significantly after indexing on the INSTR function of the DESCRIPTION column.

4.

SQL statement:

```
SELECT COUNT(*)
FROM FILM
WHERE FILM.FILM_ID IN (SELECT F.FILM_ID
    FROM FILM F, FILM I
    WHERE F.RELEASE_YEAR = I.RELEASE_YEAR
    AND F.RATING = I.RATING
    AND F.SPECIAL_FEATURES = I.SPECIAL_FEATURES
    GROUP BY F.FILM_ID
    HAVING COUNT(*) >= 40);
```

5.

Indexes are often quite useful if the column to be indexed have a high degree of uniqueness such that a selection of large portion of the rows is avoided as much as possible. Release_year, rating and special_features columns are all arguably unsuitable index types due to the lack of such uniqueness, however, if a column had to be picked, release_year would be the best index candidate as the other alternatives are vastly limited by their constraints which enforces specific values are to be accepted.

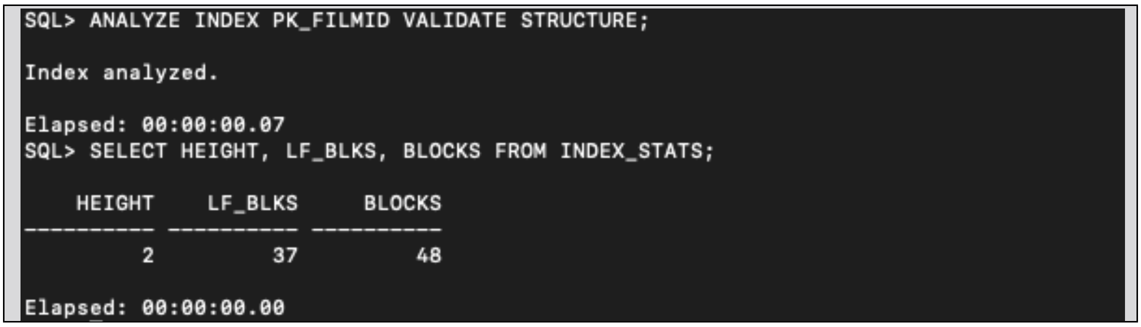
TASK 5 – EXECUTION PLAN

1.

SQL statement:

```
ANALYZE INDEX PK_FILMID VALIDATE STRUCTURE;
SELECT HEIGHT, LF_BLKS, BLOCKS FROM INDEX_STATS;
```

Output of query above:



```
SQL> ANALYZE INDEX PK_FILMID VALIDATE STRUCTURE;

Index analyzed.

Elapsed: 00:00:00.07
SQL> SELECT HEIGHT, LF_BLKS, BLOCKS FROM INDEX_STATS;

    HEIGHT    LF_BLKS     BLOCKS
---------- ---------- ----------
         2         37         48

Elapsed: 00:00:00.00
```

The output of the index analysis indicates:

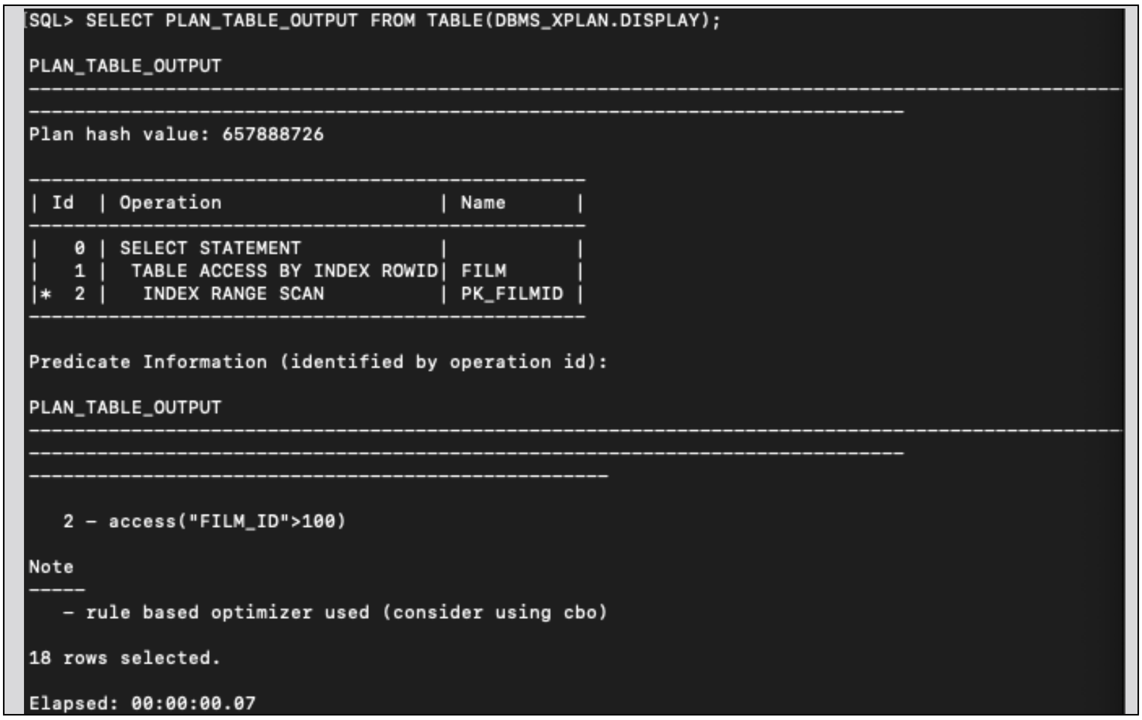
- Height of the B+ tree index is 2.
- There are 37 leaf blocks in the B+ tree index.
- 48 block accesses are needed for a full table scan.

2.

Rule-based execution plan query:

```
EXPLAIN PLAN FOR SELECT /*+RULE*/* FROM FILM WHERE FILM_ID >
100;
```

Execution plan table output:



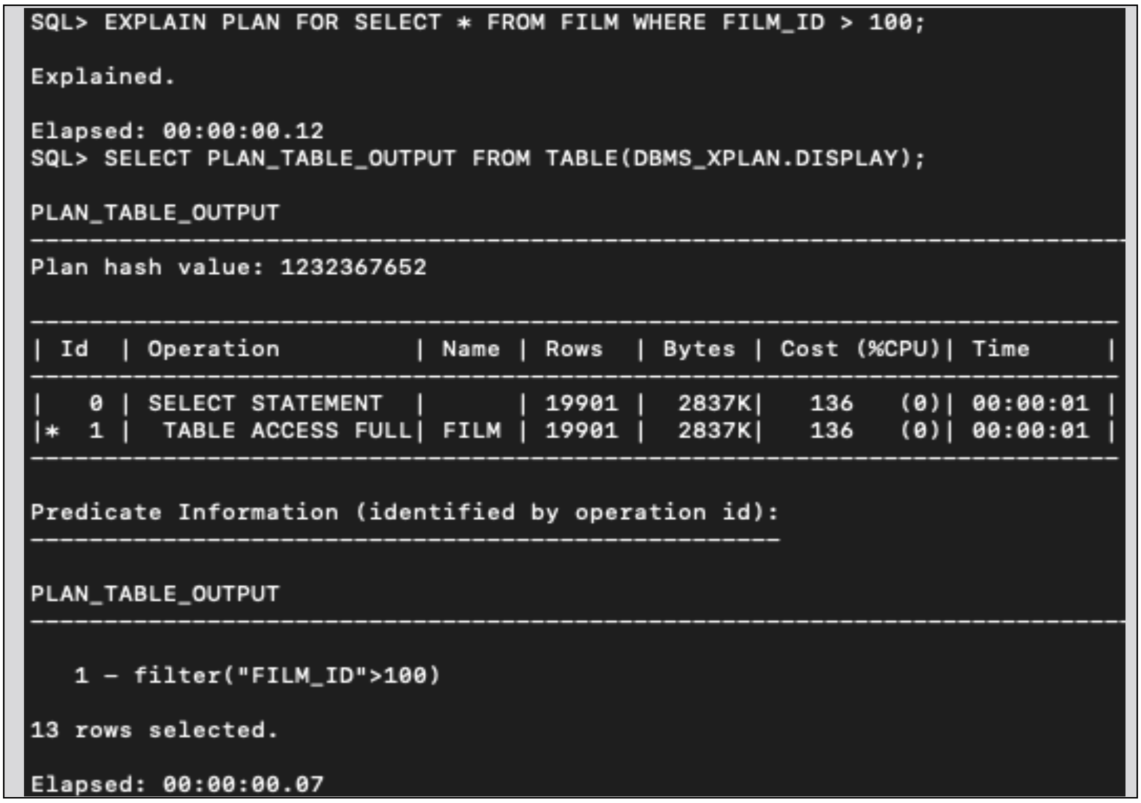
```
SQL> SELECT PLAN_TABLE_OUTPUT FROM TABLE(DBMS_XPLAN.DISPLAY);

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------
Plan hash value: 657888726

-------------------------------------------------
| Id  | Operation                   | Name      |
-------------------------------------------------
|   0 | SELECT STATEMENT            |           |
|   1 |  TABLE ACCESS BY INDEX ROWID| FILM      |
|*  2 |   INDEX RANGE SCAN          | PK_FILMID |
-------------------------------------------------

Predicate Information (identified by operation id):

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------
---------------------------------------------------

   2 - access("FILM_ID">100)

Note
-----
   - rule based optimizer used (consider using cbo)

18 rows selected.

Elapsed: 00:00:00.07
```

The plan above shows the execution of the above SQL statement as follows:

- INDEX (RANGE SCAN): Index PK_FILMID is used in a range scan operation to evaluate the WHERE clause criteria. It returns a range of ROW-ID from the index. A unique ROW-ID is not guaranteed.
- TABLE ACCESS BY INDEX ROWID – Looks up selected rows in an order determined by the ROWID obtained from the index, which specifies the data file, block and location of the row within the block.
- SELECT statement returns rows satisfying the WHERE clause conditions.

3.

Cost-based execution plan query and table output:



```
SQL> EXPLAIN PLAN FOR SELECT * FROM FILM WHERE FILM_ID > 100;

Explained.

Elapsed: 00:00:00.12
SQL> SELECT PLAN_TABLE_OUTPUT FROM TABLE(DBMS_XPLAN.DISPLAY);

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------
Plan hash value: 1232367652

-------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      | 19901 |  2837K|   136   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| FILM | 19901 |  2837K|   136   (0)| 00:00:01 |
-------------------------------------------------------------------------

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

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------

   1 - filter("FILM_ID">100)

13 rows selected.

Elapsed: 00:00:00.07
```

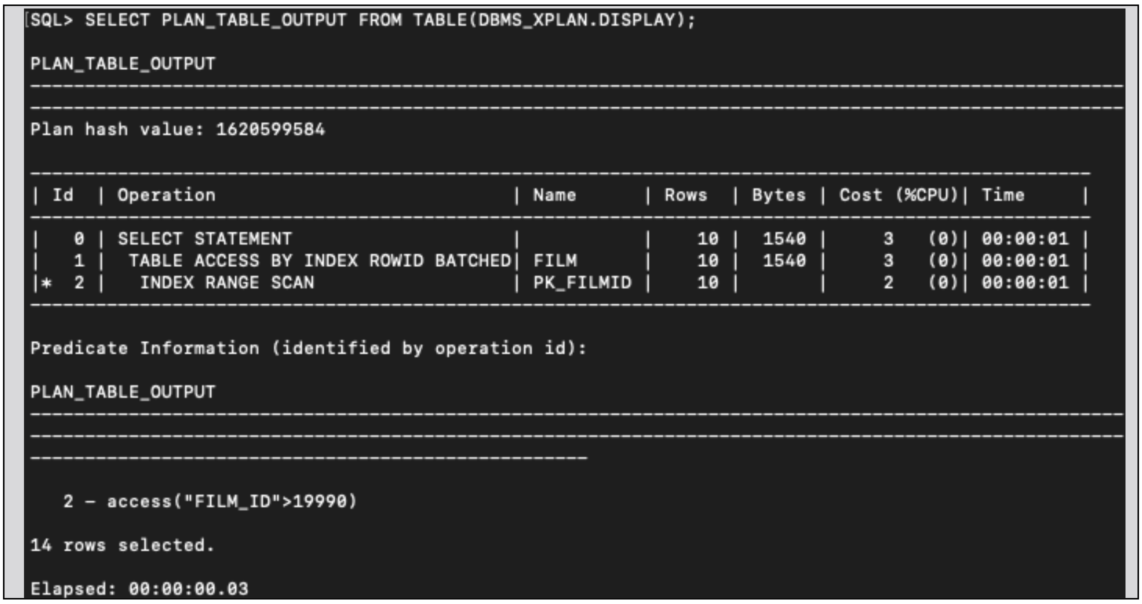
The query processing take place as follows:

- The table FILM is accessed using a full table scan. This means every row in the table FILM is accessed, and the WHERE clause criteria are evaluated for every row.
- The SELECT statement returns the rows meeting the WHERE clause criteria.
- Bytes: 2837K bytes accessed by the operation.
- Cost: The estimated cost of operation used to compare the weight of the costs of execution plans and has no particular unit of measurement.
- Time: Depicts the elapsed time in seconds of the operation.

The main differences between the rule-based and cost-based execution plans can be identified in the different steps performed during each query execution. The rule-based optimisation method employs the available index, PK_FILMID, during the table scan whereas the latter performs a full table scan as recommended by the Oracle optimiser. The number of index blocks accessed during the index range scan would equal the tree height (2 as obtained in Task 5.1) and TABLE ACCESS BY INDEX ROWID is an access to table FILM, therefore a total of 3 block accesses are performed. In above, all 19901 rows in the table are accessed with a 136 computational cost.

4.

Cost-based execution plan to search for FILM_ID > 19990:



```
[SQL> SELECT PLAN_TABLE_OUTPUT FROM TABLE(DBMS_XPLAN.DISPLAY);

PLAN_TABLE_OUTPUT
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
Plan hash value: 1620599584

-------------------------------------------------------------------------------------------------
| Id  | Operation                            | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                     |           |    10 |  1540 |     3   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| FILM      |    10 |  1540 |     3   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN                   | PK_FILMID |    10 |       |     2   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):

PLAN_TABLE_OUTPUT
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
---------------------------------------------------

   2 - access("FILM_ID">19990)

14 rows selected.

Elapsed: 00:00:00.03
```

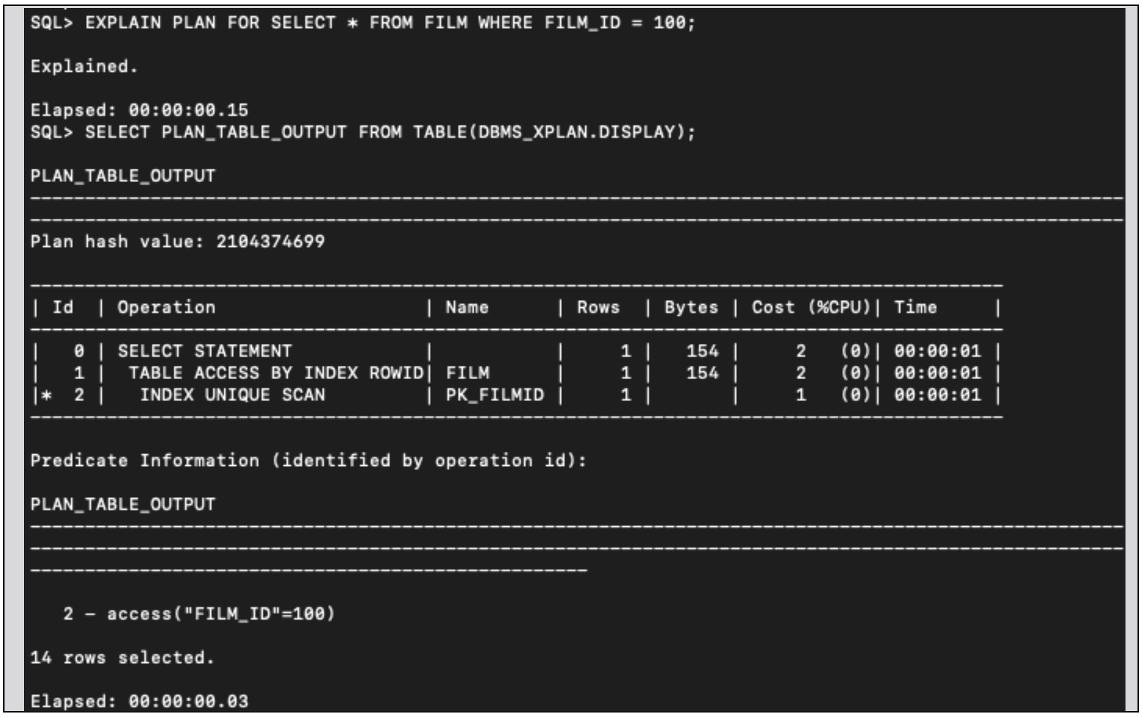
The query processing takes place as follows:

- INDEX (RANGE SCAN): Index PK_FILMID is used in a range scan operation to evaluate FILM_ID > 19990. It returns 10 rows of ROW-ID from the index.
- TABLE ACCESS BY INDEX ROWID BATCHED – Generally selects few ROWIDs from the index and then try to access the rows in blocks to minimise the number of block accesses.
- SELECT statement returns rows satisfying the WHERE clause conditions.

As shown previously, the Oracle optimiser recommended a full table scan for FILM_ID > 100 in contrast to the execution plan shown above which employs the index range scan for FILM_ID > 19990. In batched access, a few row ids from the index are retrieved and sorted in block order to improve clustering and reduce the number of times that the database must access a block. The noticeable difference between the two is arguably the number of rows, bytes, and cost of running the query as shown above. The cost of running a full scan is severe in all three aspects in comparison to the those of index scan, making it a much better method for performing the query. Also, the query above returns 10 rows because the maximum number of records in the database is 20000 and all records after 19990 was accessed.

5.

Cost-based execution plan to search for FILM_ID = 100:



```
SQL> EXPLAIN PLAN FOR SELECT * FROM FILM WHERE FILM_ID = 100;

Explained.

Elapsed: 00:00:00.15
SQL> SELECT PLAN_TABLE_OUTPUT FROM TABLE(DBMS_XPLAN.DISPLAY);

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------
Plan hash value: 2104374699

-------------------------------------------------------------------------------------------
| Id  | Operation                   | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |           |     1 |   154 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| FILM      |     1 |   154 |     2   (0)| 00:00:01 |
|*  2 |   INDEX UNIQUE SCAN         | PK_FILMID |     1 |       |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------
----------------------------------------------------

   2 - access("FILM_ID"=100)

14 rows selected.

Elapsed: 00:00:00.03
```

This plan shows the execution of the SQL statement and other statistics as follows:

- INDEX (UNIQUE SCAN): Index PK_FILMID is used in a unique scan operation to evaluate the WHERE clause criteria, that is FILM_ID=100. It returns exactly 1 ROW-ID from the index.
- TABLE ACCESS BY INDEX ROWID – Rows are located using index.
- SELECT statement returns rows satisfying FILM_ID = 100.
- Bytes: 154 bytes accessed during the operation.
- Cost: Cost of operations (1-2).

In unique scan, a single row is returned whereas all rows are returned in a full table scan, followed by much less cost and smaller size of bytes accessed during the execution of the query. Only a single row is accessed as we just want FILM_ID = 100 from the index which is a discriminating factor for the performance difference between the above query and observations made in task 5.3. This would also suggest there are 3 block accesses required for the query above which complies with task 5.1's statistics, in that the total number of accesses equal height+1 which is 3.