INFS2200 Report - 4582038

Task 1.

Q1.

Statement:

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

Answer:

- The following constraints defined in Table 1 have been created by running prjScript.sql:
  - FK_FILMID1
  - PK_FILMID
  - PK_ACTORID

Q2.

Statements:

```sql
PK_CATEGORYID:

ALTER TABLE CATEGORY ADD CONSTRAINT PK_CATEGORYID PRIMARY KEY (category_id);

PK_LANGUAGEID:

ALTER TABLE LANGUAGE ADD CONSTRAINT PK_LANGUAGEID PRIMARY KEY (language_id);

UN_DESCRIPTION:

ALTER TABLE FILM ADD CONSTRAINT UN_DESCRIPTION UNIQUE (description);

CK_FNAME:

ALTER TABLE ACTOR MODIFY first_name VARCHAR2(45) NOT NULL;

CK_LNAME:

ALTER TABLE ACTOR MODIFY last_name VARCHAR2(45) NOT NULL;

CK_CATNAME:

ALTER TABLE CATEGORY MODIFY name VARCHAR2(25) NOT NULL;

CK_LANNAME:

ALTER TABLE LANGUAGE MODIFY name VARCHAR2(20) NOT NULL;

CK_TITLE:

ALTER TABLE FILM MODIFY title VARCHAR2(255) NOT NULL;

CK_RELEASEYR:

ALTER TABLE FILM ADD CONSTRAINT CK_RELEASEYR
CHECK (release_year <= 2020);

CK_RATING:

ALTER TABLE FILM ADD CONSTRAINT CK_RATING
CHECK (rating IN ('G', 'PG', 'PG-13', 'R', 'NC-17'));
```

CK_SPLFEATURES:

```sql
ALTER TABLE FILM ADD CONSTRAINT CK_SPLFEATURES
CHECK (special_features IN ('Trailers', 'Commentaries', 'Deleted Scenes', 'Behind the Scenes', NULL));
```

FK_LANGUAGEID:

```sql
ALTER TABLE FILM ADD CONSTRAINT FK_LANGUAGEID
FOREIGN KEY (language_id) REFERENCES LANGUAGE (language_id);
```

FK_ORLANGUAGEID:

```sql
ALTER TABLE FILM ADD CONSTRAINT FK_ORLANGUAGEID
FOREIGN KEY (original_language_id) REFERENCES LANGUAGE (language_id);
```

FK_ACTORID:

```sql
ALTER TABLE FILM_ACTOR ADD CONSTRAINT FK_ACTORID
FOREIGN KEY (actor_id) REFERENCES ACTOR (actor_id);
```

FK_CATEGORYID:

```sql
ALTER TABLE FILM_CATEGORY ADD CONSTRAINT FK_CATEGORYID
FOREIGN KEY (category_id) REFERENCES CATEGORY (category_id);
```

FK_FILMID2:

```sql
ALTER TABLE FILM_CATEGORY ADD CONSTRAINT FK_FILMID2
FOREIGN KEY (film_id) REFERENCES FILM (film_id);
```

<u>Task 2.</u>

Q1.

Statement:

```
CREATE SEQUENCE "FILM_ID_SEQ" MINVALUE 20010 MAXVALUE 999999999999
INCREMENT BY 10 START WITH 20010;
```

Q2.

Statement:

```
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;
/
```

Q3.

Statement:

```
CREATE OR REPLACE TRIGGER "BI_FILM_DESP"
    BEFORE INSERT ON "FILM"
    FOR EACH ROW
    WHEN (NEW.rating IS NOT NULL AND NEW.language_id IS NOT NULL AND NEW.original_language_id IS NOT NULL)
DECLARE
    rating_count int;
    new_language varchar2(20);
    release_language varchar2(20);
BEGIN
    SELECT COUNT(*) INTO rating_count
    FROM FILM
    WHERE FILM.rating = :NEW.rating;

    SELECT name INTO release_language
    FROM LANGUAGE
    WHERE :NEW.original_language_id = LANGUAGE.language_id;

    SELECT name INTO new_language
    FROM LANGUAGE
    WHERE :NEW.language_id = LANGUAGE.language_id;

    SELECT (:NEW.description || :NEW.rating || '-' || TO_CHAR(rating_count) ||
    ': Originally in ' || release_language || '. Re-released in ' || new_language || '.') INTO :NEW.description
    FROM DUAL;
END;
/
```

Task 3.

Q1.

Statement:

```
SELECT title, length
FROM FILM, FILM_CATEGORY, CATEGORY
WHERE FILM.film_id = FILM_CATEGORY.film_id
    AND FILM_CATEGORY.category_id = CATEGORY.category_id
    AND CATEGORY.name = 'Action'
    AND length = (SELECT MIN(length)
                  FROM FILM, FILM_CATEGORY, CATEGORY
                  WHERE FILM.film_id = FILM_CATEGORY.film_id
                  AND FILM_CATEGORY.category_id = CATEGORY.category_id
                  AND CATEGORY.name = 'Action');
```

Output:

```
TITLE                     LENGTH
------------------------- ------
STAR HEARTBREAKERS         00046
SPEAKEASY SILVERADO        00046
CLASH JAWS                 00046
PIZZA BIRDCAGE             00046
FIRE AUTUMN                00046
METROPOLIS DREAM           00046
VANISHING AIRPORT          00046
PEACH PARADISE             00046
DOORS AIRPORT              00046
HOLLOW DYNAMITE            00046
```

Q2.

Statement:

```
CREATE VIEW MIN_ACTION_ACTORS AS
    SELECT DISTINCT(A.actor_id), A.first_name, A.last_name
    FROM ACTOR A, (SELECT actor_id
                   FROM FILM_ACTOR, FILM, FILM_CATEGORY, CATEGORY
                   WHERE FILM.film_id = FILM_CATEGORY.film_id
                   AND FILM.film_id = FILM_ACTOR.film_id
                   AND FILM_CATEGORY.category_id = CATEGORY.category_id
                   AND CATEGORY.name = 'Action'
                   AND length = (SELECT MIN(F.length)
                                 FROM FILM F, FILM_CATEGORY, CATEGORY
                                 WHERE FILM.film_id = FILM_CATEGORY.film_id
                                 AND FILM_CATEGORY.category_id = CATEGORY.category_id
                                 AND CATEGORY.name = 'Action')) B
    WHERE A.actor_id = B.actor_id;
```

Q3.

Statement:

```
CREATE VIEW V_ACTION_ACTORS_2012 AS
    SELECT DISTINCT(A.actor_id), A.first_name, A.last_name
    FROM ACTOR A, (SELECT actor_id
                   FROM FILM_ACTOR, FILM, FILM_CATEGORY, CATEGORY
                   WHERE FILM.film_id = FILM_CATEGORY.film_id
                   AND FILM.film_id = FILM_ACTOR.film_id
                   AND FILM_CATEGORY.category_id = CATEGORY.category_id
                   AND CATEGORY.name = 'Action'
                   AND FILM.release_year = '2012') B
    WHERE A.actor_id = B.actor_id;
```

Q4.

Statement:

```
CREATE MATERIALIZED VIEW MV_ACTION_ACTORS_2012
    BUILD IMMEDIATE
    AS
    SELECT DISTINCT(A.actor_id), A.first_name, A.last_name
    FROM ACTOR A, (SELECT actor_id
                   FROM FILM_ACTOR, FILM, FILM_CATEGORY, CATEGORY
                   WHERE FILM.film_id = FILM_CATEGORY.film_id
                   AND FILM.film_id = FILM_ACTOR.film_id
                   AND FILM_CATEGORY.category_id = CATEGORY.category_id
                   AND CATEGORY.name = 'Action'
                   AND FILM.release_year = '2012') B
    WHERE A.actor_id = B.actor_id;
```

Q5.

Statements To Retrieve Information:

```
SET TIMING ON;

SELECT * FROM V_ACTION_ACTORS_2012;
SELECT * FROM MV_ACTION_ACTORS_2012;

SET TIMING OFF;

EXPLAIN PLAN FOR SELECT * FROM V_ACTION_ACTORS_2012;
SELECT PLAN_TABLE_OUTPUT FROM TABLE (DBMS_XPLAN.DISPLAY);

EXPLAIN PLAN FOR SELECT * FROM MV_ACTION_ACTORS_2012;
SELECT PLAN_TABLE_OUTPUT FROM TABLE (DBMS_XPLAN.DISPLAY);
```

Virtual View:

- Elapsed:

```
Elapsed: 00:00:00.20
```

- Query Execution Plan:

PLAN_TABLE_OUTPUT

Plan hash value: 3856098404

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		528	32208	222 (2)	00:00:01
1	VIEW	V_ACTION_ACTORS_2012	528	32208	222 (2)	00:00:01
2	HASH UNIQUE		528	77616	222 (2)	00:00:01
3	NESTED LOOPS		528	77616	221 (2)	00:00:01
4	NESTED LOOPS		528	77616	221 (2)	00:00:01
* 5	HASH JOIN		528	45408	221 (2)	00:00:01
* 6	HASH JOIN SEMI		21	1260	151 (1)	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	129K	3299K	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

Materialised View:

- Elapsed Time:

```
Elapsed: 00:00:00.18
```

- Query Execution Plan:

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

Answer:

- As shown by the two elapsed time recordings, the materialised view slightly reduced the required time to process the query.
- Additionally, as shown by its respective query execution plan (QEP), the query on the virtual view is translated to another query on base tables with numerous joins and tables which significantly increases CPU cost and query process time. In comparison, a single full table access can be performed on the materialised view (allowing the query to be performed directly on the view). This significantly reduced the number of necessary operations which in turn reduced CPU cost and runtime of the query, when compared to the virtual view.

Task 4.

Q1.

Statement:

```
SELECT *
FROM FILM F
WHERE INSTR(F.description, 'Boat') IS NOT NULL AND INSTR(F.description, 'Boat') > 0
ORDER BY F.title
FETCH FIRST 100 ROWS ONLY;
```

Output (top 3 and bottom 3 tuples):

```
FILM_ID TITLE                    DESCRIPTION              RELEASE_YEAR LANGUAGE_ID ORIGINAL_LANGUAGE_ID RENTAL_DURATION RENTAL_RATE LENGTH REPLACEMENT_COST RATING
------- ------------------------ ------------------------ ------------ ----------- -------------------- --------------- ----------- ------ ---------------- -------
SPECIAL_FEATURES
------------------------
  05880 ACADEMY REDS             A Beautiful Story of a Cr        1974         001                                  007     0000.99  00142         00017.99 R
                                 ocodile and a Lumberjack
                                 who must Vanquish a Foren
                                 sic Psychologist in a Jet
                                  Boat
Commentaries

  04952 ADAPTATION JEOPARDY      A Thrilling Saga of a Stu        1958         006                                  004     0002.99  00129         00028.99 R
                                 dent and a Car who must B
                                 attle a Technical Writer
                                 in A U-Boat
Commentaries

  06107 AFFAIR PANIC             An Intrepid Yarn of a Tea        1975         001                                  004     0000.99  00136         00023.99 NC-17
                                 cher and a Student who mu
                                 st Pursue a Madman in a J
                                 et Boat
```

```
  09446 BEETHOVEN CIRCUS         An Intrepid Story of a Hu        1975         001                                  007     0004.99  00057         00027.99 PG
                                 nter and a Database Admin
                                 istrator who must Face a
                                 Monkey in a Jet Boat
Commentaries

FILM_ID TITLE                    DESCRIPTION              RELEASE_YEAR LANGUAGE_ID ORIGINAL_LANGUAGE_ID RENTAL_DURATION RENTAL_RATE LENGTH REPLACEMENT_COST RATING
------- ------------------------ ------------------------ ------------ ----------- -------------------- --------------- ----------- ------ ---------------- -------
SPECIAL_FEATURES
------------------------
  18513 BEETHOVEN MOCKINGBIRD    A Taut Story of a Woman a        1978         004                                  004     0000.99  00105         00016.99 NC-17
                                 nd a Man who must Find a
                                 Database Administrator in
                                  A U-Boat
Trailers

  15677 BEHAVIOR SLIPPER         An Amazing Display of a H        1955         001                                  006     0004.99  00084         00024.99 PG
                                 unter and a Waitress who
                                 must Redeem a Pioneer in
                                 a Jet Boat
Deleted Scenes

100 rows selected.
```

Q2.

Statement:

```
CREATE INDEX IDX_BOAT ON FILM(INSTR(description, 'Boat'));
```

Justification:

- As the query searches for tuples which satisfy a single condition (description contains ‘Boat’) a single column index using the function INSTR(description, ‘Boat’) indicates if the description contains ‘Boat’. This simplifies the where condition of the query to check the index rather than performing INSTR for each tuple in FILM each time the query is queried.

Q3.

Statements To Retrieve Information:

```
DROP INDEX IDX_BOAT;
SET TIMING ON;

SELECT *
FROM FILM F
WHERE INSTR(F.description, 'Boat') IS NOT NULL AND INSTR(F.description, 'Boat') > 0
ORDER BY F.title
FETCH FIRST 100 ROWS ONLY;

SET TIMING OFF;

EXPLAIN PLAN FOR SELECT *
FROM FILM F
WHERE INSTR(F.description, 'Boat') IS NOT NULL AND INSTR(F.description, 'Boat') > 0
ORDER BY F.title
FETCH FIRST 100 ROWS ONLY;
SELECT PLAN_TABLE_OUTPUT FROM TABLE (DBMS_XPLAN.DISPLAY);

CREATE INDEX IDX_BOAT ON FILM(INSTR(description, 'Boat'));

SET TIMING ON;

SELECT *
FROM FILM F
WHERE INSTR(F.description, 'Boat') IS NOT NULL AND INSTR(F.description, 'Boat') > 0
ORDER BY F.title
FETCH FIRST 100 ROWS ONLY;

SET TIMING OFF;

EXPLAIN PLAN FOR SELECT *
FROM FILM F
WHERE INSTR(F.description, 'Boat') IS NOT NULL AND INSTR(F.description, 'Boat') > 0
ORDER BY F.title
FETCH FIRST 100 ROWS ONLY;
SELECT PLAN_TABLE_OUTPUT FROM TABLE (DBMS_XPLAN.DISPLAY);
```

Before Index:

- Elapsed Time:

```
Elapsed: 00:00:00.50
```

- Execution Plan:

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

-------------------------------------------------------------------------------
| Id  | Operation                | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT         |      |   100 | 63900 |   138   (2)| 00:00:01 |
|*  1 |  VIEW                    |      |   100 | 63900 |   138   (2)| 00:00:01 |
|*  2 |   WINDOW SORT PUSHED RANK|      |  1000 |   142K|   138   (2)| 00:00:01 |
|*  3 |    TABLE ACCESS FULL     | FILM |  1000 |   142K|   137   (1)| 00:00:01 |
-------------------------------------------------------------------------------

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

   1 - filter("from$_subquery$_002"."rowlimit_$$_rownumber"<=100)
   2 - filter(ROW_NUMBER() OVER ( ORDER BY "F"."TITLE")<=100)
   3 - filter(INSTR("F"."DESCRIPTION",'Boat')>0 AND
              INSTR("F"."DESCRIPTION",'Boat') IS NOT NULL)
```

After Index:

- Elapsed Time:

```
Elapsed: 00:00:00.45
```

- Execution Plan:

```
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 |
-----------------------------------------------------------------------------------------------

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

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

Answer:

- As shown by the above elapsed times, the index speeds up query processing.
- Additionally, as seen by the corresponding execution plans, the index significantly reduced CPU cost. This can be attributed to the fact that the indexed table can be accessed by index instead of a full table access (accessing only tuples which meets the condition).
- Thus, both the recorded elapsed times and execution plans indicate a notable decrease in query processing time.

Q4.

Statement:

```
SELECT COUNT(*)
FROM (SELECT film_id
      FROM FILM G, (SELECT F.release_year, F.rating, F.special_features, COUNT(*)
                    FROM FILM F
                    GROUP BY F.release_year, F.rating, F.special_features
                    HAVING COUNT(*) >= 41) H
      WHERE G.release_year = H.release_year AND
            G.rating = H.rating AND
            G.special_features = H.special_features);
```

Output:

```
  COUNT(*)
----------
       265
```

Q5.

Answer:

- The most suitable index type to create on the columns is a composite secondary index.
- A composite index should be used because multiple columns are used within the group by clause, allowing for easier grouping of equivalent element tuples.
- A secondary index should be used since the table is ordered on the primary key and as such another primary/clustering index cannot be created.
- Additionally, another possible option would be a bitmap index, however due to the possibly large number of unique values it is not applicable in this situation.
- A function-based index was considered but was rejected as it would significantly affect insertion, update and deletion procedures as all entries in the index would need to be updated.
- As such, a composite secondary index seems the most applicable in this situation.

Task 5.

Q1.

Statements:

```
ANALYZE INDEX PK_FILMID VALIDATE STRUCTURE;

---- a, b)

SELECT HEIGHT, LF_BLKS
FROM INDEX_STATS;

---- c)

SELECT BLOCKS
FROM USER_TABLES
WHERE table_name = 'FILM';
```

Outputs:

a, b)

```
    HEIGHT    LF_BLKS
---------- ----------
         2         37
```

c)

```
    BLOCKS
----------
       496
```

Answers:

a) Height of B+ Tree = 2
b) Number of leaf blocks in the B+ tree index = 37
c) Number of block access for a full table scan on ‘film’ = 496

Q2.

Statement To Retrieve QEP:

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

Output:

```
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):
---------------------------------------------------

   2 - access("FILM_ID">19990)

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

18 rows selected.
```

Answer:

- As shown by the above QEP, the query first performs an index range scan on the primary key of FILM for film_ids > 100. Then locate the related tuples in FILM using the indices. Then finally performs the select statement and returns the requested fields from the resultant tuples.

Q3.

Statement To Retrieve QEP:

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

Output:

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

--------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      | 19901 |  2837K|   137   (1)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| FILM | 19901 |  2837K|   137   (1)| 00:00:01 |
--------------------------------------------------------------------------

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

   1 - filter("FILM_ID">100)

13 rows selected.
```

Answer:

- The query first performs a full table access on FILM checking all tuples against the arguments in the where clause. Then performs the select statement returning the necessary fields of the resultant tuples.
- An obvious difference between the plans obtained in Q3 and Q2 is that Q2 utilises the primary index to search for relevant tuples while Q3 just performs a full table search checking each tuple.
- In Q2, the index range scan performs tree height – 1 + 2 * leaf blocks (at most) block accesses (in this case 2*37 accesses). The true number of block accesses index range scan would perform is tree height – 1 + # of matching leaf blocks + # of matching data blocks. On the other hand in Q3, the full table scan performs exactly 496 block accesses. The main difference between the two plans is the resultant average number of block accesses.
- Another significant difference, is the number of expected rows which must be accessed and returned.
- Note:
    - For Q2, the index range scan might the same block multiple times if the number of valid row ids is significant, which would increase the number of block access performed.

Q4.

Statement To Retrieve QEP:

```
EXPLAIN PLAN FOR SELECT * FROM FILM WHERE FILM_ID > 19990;
```

Output:

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

-----------------------------------------------------------------------------------------
| Id  | Operation                           | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |           |    10 |  1460 |     3   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| FILM      |    10 |  1460 |     3   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN                  | PK_FILMID |    10 |       |     2   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------

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

   2 - access("FILM_ID">19990)

14 rows selected.
```

Answer:

- The query first performs an index range scan on PK_FILM for values > 19990. It uses the indices to performing multi-block read on a few tuples which satisfy the where clause, then tries to access rows in block order. Then performs the select statement returning the necessary fields of the resultant tuples.
- The main observable difference between the plans produced by Q3 and Q4 is that Q4 utilises the primary index whilst Q3 does not.
- The number of block accesses performed by index range scan, in Q4, is tree height – 1 + # of matching leaf blocks + # of matching data blocks. While Q3 performs exactly 496 block accesses. Thus Q4's QEP has a significantly reduced cost compared to Q3's QEP.
- Q4 utilises the index because it is trying to access a significantly smaller subset of tuples than those accessed in Q3.

Q5.

Statement To Retrieve QEP:

```
EXPLAIN PLAN FOR SELECT * FROM FILM WHERE FILM ID = 100;
```

Output:

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

---------------------------------------------------------------------------------
| Id  | Operation                   | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |           |     1 |   146 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| FILM      |     1 |   146 |     2   (0)| 00:00:01 |
|*  2 |   INDEX UNIQUE SCAN         | PK_FILMID |     1 |       |     1   (0)| 00:00:01 |
---------------------------------------------------------------------------------

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

   2 - access("FILM_ID"=100)

14 rows selected.
```

Answer:

- The query first uses PK_FILMID in a unique scan operation to evaluate the where clause criteria, returning exactly one row-id from the index.
- Rows are then accessed using the index.
- Then the select statement returns the resultant rows.
- Differences between the QEP of Q5 and Q3:
    - Q5 performs tree height + 1 block access and only returns one row (since FILM_ID is the primary key) unlike Q3 which accesses every row and block. Similar to Q4, Q5 has a significantly lower cost than Q3.
    - Similarly to Q4, Q5 utilises the index because it is trying to access a significantly smaller subset of tuples than those accessed in Q3.