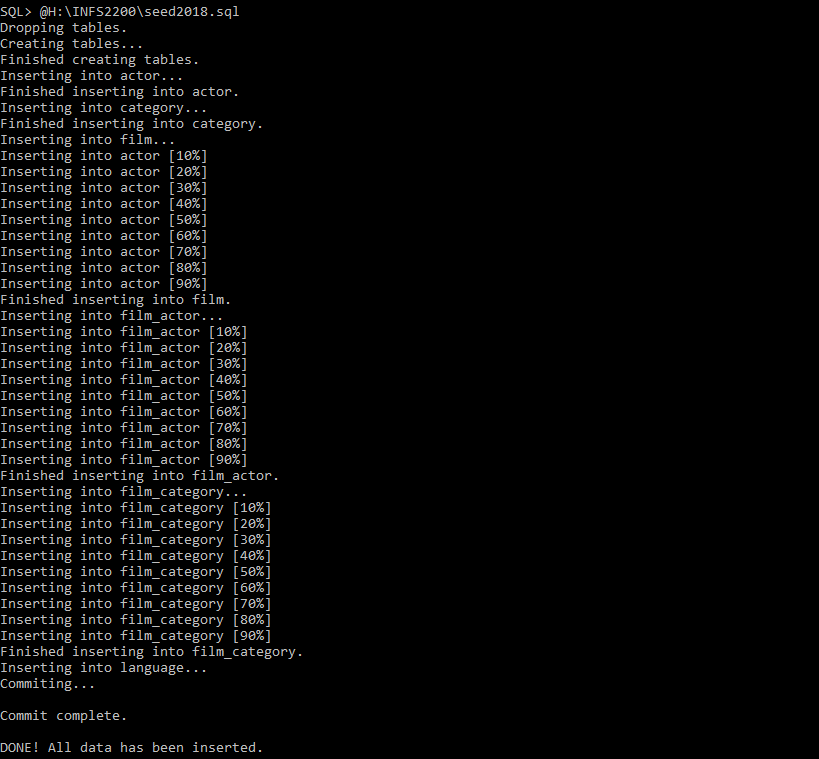
# SECTION 2. ASSIGNMENT TASKS

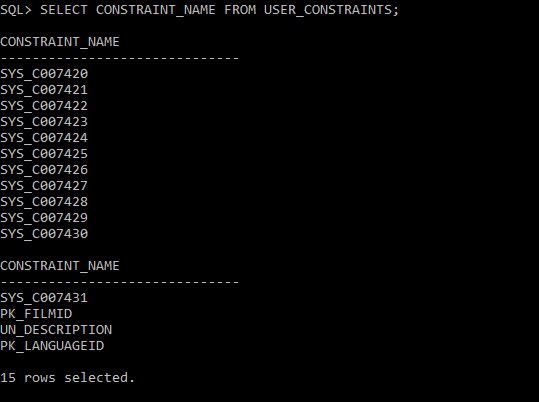
## Task 0 – Database

@H:\INFS2200\seed2018.sql



## Task 1 –Constraints

SELECT CONSTRAINT\_NAME FROM USER\_CONSTRAINTS;



ALTER TABLE ACTOR ADD CONSTRAINT PK\_ACTORID PRIMARY KEY (actor\_id);

ALTER TABLE category ADD CONSTRAINT PK\_CATEGORYID PRIMARY KEY (category\_id);

ALTER TABLE ACTOR MODIFY first\_name CONSTRAINT CK\_FNAME NOT NULL;

ALTER TABLE ACTOR MODIFY last\_name CONSTRAINT CK\_LNAME NOT NULL;

ALTER TABLE FILM MODIFY title CONSTRAINT CK\_TILE NOT NULL;

ALTER TABLE CATEGORY  MODIFY name CONSTRAINT CK\_CATANAME NOT NULL;

ALTER TABLE FILM  MODIFY rental\_rate CONSTRAINT CK\_RENTALRATE NOT NULL;

ALTER TABLE FILM ADD CONSTRAINT CK\_RATING CHECK ((rating = 'G') OR (rating = 'PG') OR (rating = 'PG-13') OR (rating = 'R') OR (rating = 'NC-17'));

ALTER TABLE FILM ADD CONSTRAINT CK\_SPLFEATURES CHECK ((special\_features = 'Trailers') OR (special\_features IS NULL) OR (special\_features = 'Commentaries') OR (special\_features = 'Deleted Scenes') OR (special\_features = 'Behind the Scenes'));

ALTER TABLE FILM ADD CONSTRAINT FK\_LANGUAGE FOREIGN KEY (language\_id) REFERENCES language (language\_id);

ALTER TABLE FILM ADD CONSTRAINT FK\_ORLANGUAGEID FOREIGN KEY (original\_language\_id ) REFERENCES language (language\_id);

ALTER TABLE FILM\_ACTOR ADD CONSTRAINT FK\_ACTORID FOREIGN KEY (ACTOR\_ID) REFERENCES ACTOR (ACTOR\_ID);

ALTER TABLE FILM ADD CONSTRAINT CK\_RELEASEYR CHECK (RELEASE\_YEAR <= 2018);

## Task 2 –Triggers

1. Write a trigger, named BI\_FILM\_ID that automatically populates the film\_id when a new film is added. The sequence, named FILM\_ID\_SEQ, should start from 22,000 and increment by 2.

CREATE SEQUENCE "FILM\_ID\_SEQ" MINVALUE 2020 INCREMENT BY 2 START WITH 22000;

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;

 /

SHOW ERRORS;

2. Write an SQL trigger, that should be named BI\_FILM\_LANG, to append text to the description of every new film inserted into the database. It is based on the language (language\_id) and the original language (original\_language\_id) of the film.

CREATE OR REPLACE TRIGGER "BI\_FILM\_LANG"

BEFORE INSERT ON "FILM"

FOR EACH ROW

DECLARE

  OL\_NAME VARCHAR2(20);

  NL\_NAME VARCHAR2(20);

  N\_DES VARCHAR(295);

BEGIN

  IF ((:NEW.original\_language\_id IS NOT NULL) AND (:NEW.language\_id IS NOT NULL)) THEN

    SELECT name INTO OL\_NAME FROM LANGUAGE WHERE language\_id = :NEW.original\_language\_id;

    SELECT name INTO NL\_NAME FROM LANGUAGE WHERE language\_id = :NEW.language\_id;

    SELECT CONCAT(CONCAT(CONCAT('Originally in ', OL\_NAME), '. Re-released in '), NL\_NAME) INTO N\_DES FROM DUAL;

    :NEW.description := CONCAT(:NEW.description, N\_DES);

  END IF;

end;

/

SHOW ERRORS;

## Task 3 – Views

1. Write a SQL statement to find the ‘Comedy’ films with the longest running time. Your query should output the titles and lengths of the films.

SELECT title, length

FROM FILM

WHERE film\_id IN (

SELECT film\_id FROM FILM\_CATEGORY WHERE category\_id IN (

SELECT category\_id

FROM CATEGORY

WHERE name = 'Comedy'))

AND LENGTH = (SELECT MAX(length) FROM FILM

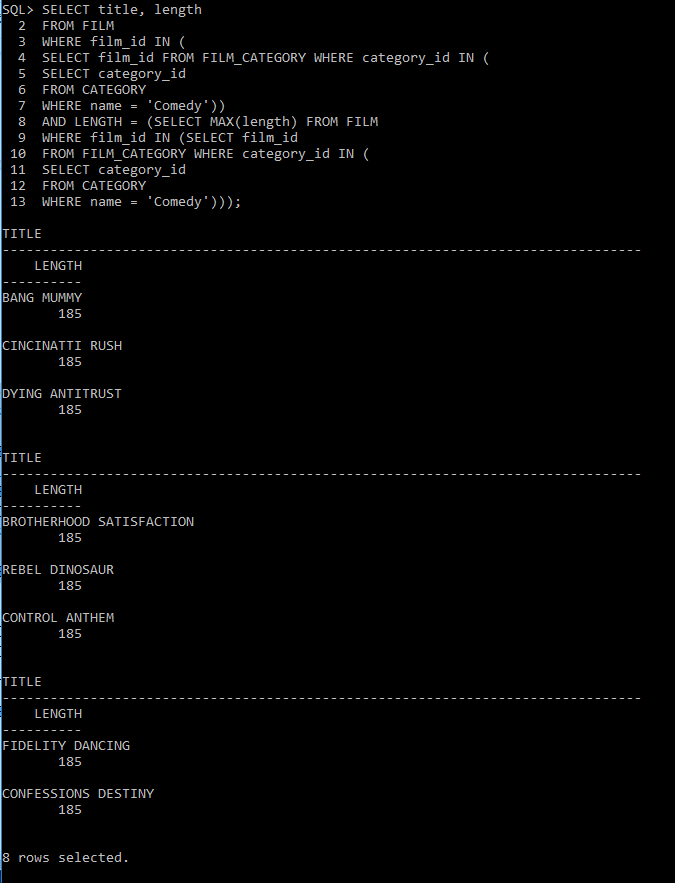
WHERE film\_id IN (SELECT film\_id

FROM FILM\_CATEGORY WHERE category\_id IN (

SELECT category\_id

FROM CATEGORY

WHERE name = 'Comedy')));



2. Write a SQL statement to create a (virtual) view called MAX\_COMEDY\_ACTORS that contains all the actors that have acted in the films that you obtained in Task 3.1. The view should include the columns actor id, first name and last name. Note: Actors may act in multiple films, but should only appear once in the view.

CREATE OR REPLACE VIEW MAX\_COMEDY\_ACTORS AS

SELECT \* FROM ACTOR

  WHERE actor\_id IN (SELECT actor\_id

FROM FILM\_ACTOR

   WHERE film\_id IN (SELECT film\_id

FROM FILM

WHERE film\_id IN (

SELECT film\_id FROM FILM\_CATEGORY WHERE category\_id IN (

SELECT category\_id FROM CATEGORY WHERE name = 'Comedy'))

AND LENGTH = (SELECT MAX(length) FROM FILM

WHERE film\_id IN (SELECT film\_id

FROM FILM\_CATEGORY WHERE category\_id IN (

SELECT category\_id FROM CATEGORY WHERE name = 'Comedy')))));

3. Write a SQL statement to create a (virtual) view called V\_COMEDY\_ACTORS\_2008 that lists the ids, first names and last names of any actors that starred in a Comedy film released in the year 2008. CREATE OR REPLACE VIEW V\_COMEDY\_ACTORS\_2008 AS

SELECT first\_name, last\_name FROM ACTOR WHERE ACTOR\_ID IN (SELECT DISTINCT ACTOR\_ID

                                                         FROM FILM\_ACTOR WHERE film\_id IN (SELECT FILM\_ID FROM FILM

                 WHERE (release\_year = 2008)

                 AND FILM\_ID IN (SELECT film\_id FROM FILM\_CATEGORY WHERE category\_id IN (

SELECT category\_id

FROM CATEGORY

WHERE name = 'Comedy'))));

4. Write a SQL statement to create a materialized view called MV\_COMEDY\_ACTORS\_2008 that lists the same information as in Task 3.3.

CREATE MATERIALIZED VIEW MV\_COMEDY\_ACTORS\_2008

BUILD IMMEDIATE

AS

SELECT first\_name, last\_name FROM ACTOR WHERE ACTOR\_ID IN (SELECT DISTINCT ACTOR\_ID

                                                         FROM FILM\_ACTOR WHERE film\_id IN (SELECT FILM\_ID FROM FILM

                 WHERE (release\_year = 2008)

                 AND FILM\_ID IN (

SELECT film\_id FROM FILM\_CATEGORY WHERE category\_id IN (

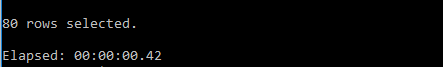
SELECT category\_id

FROM CATEGORY

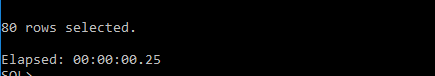
WHERE name = 'Comedy'))));

**TASK3.5**

SELECT \* FROM V\_COMEDY\_ACTORS\_2008;



SELECT \* FROM MV\_COMEDY\_ACTORS\_2008;



The first one is virtual view, and it takes longer time to query. A view uses a query to pull data from the underlying table. But when we creating Materialized view, it will create two objects, and it will physically store the view and its data. So that, in this case, the virtual view takes much longer time than materialized view.

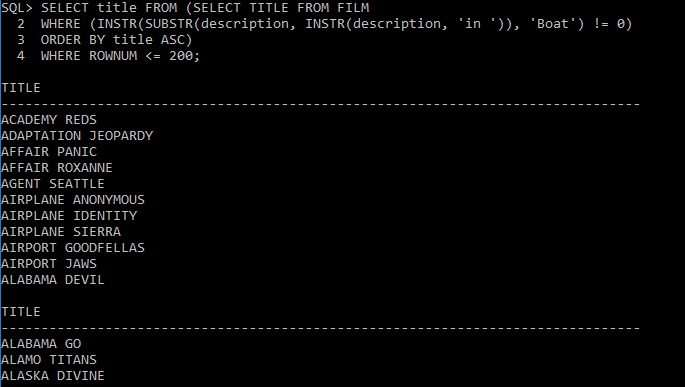
## Task 4 – Indexes

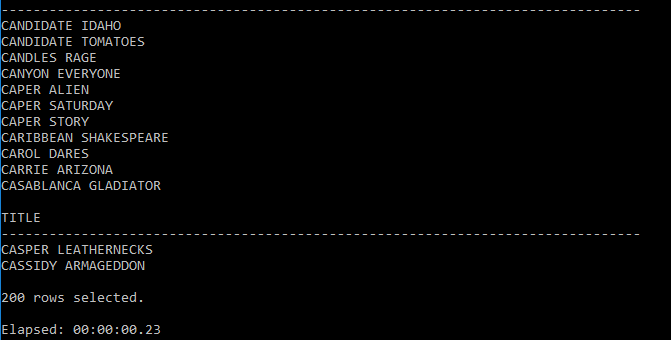
SELECT title FROM (SELECT TITLE FROM FILM

WHERE (INSTR(SUBSTR(description, INSTR(description, 'in ')), 'Boat') != 0)

ORDER BY title ASC)

WHERE ROWNUM <= 200;





TASK 4.2

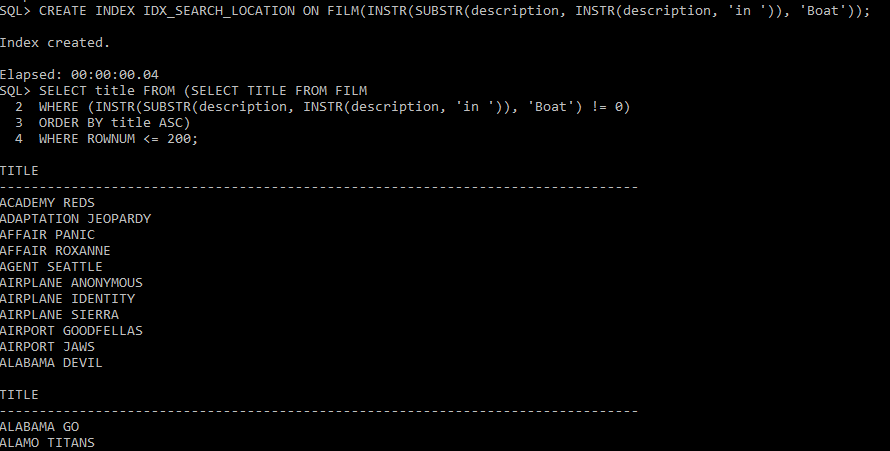
CREATE INDEX IDX\_SEARCH\_LOCATION ON FILM(INSTR(SUBSTR(description, INSTR(description, 'in ')), 'Boat'));

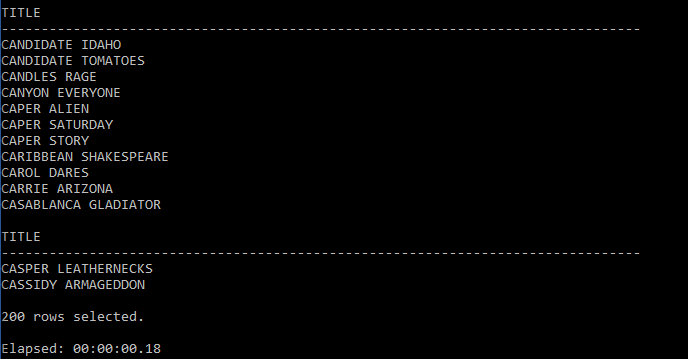
SELECT title FROM (SELECT TITLE FROM FILM

WHERE (INSTR(SUBSTR(description, INSTR(description, 'in ')), 'Boat') != 0)

ORDER BY title ASC)

WHERE ROWNUM <= 200;





Since we using functions to analysis each description costs time for searching each row. Hence, we create function-based index to reduce analysis time

TASK4.3

Without index:



With index:



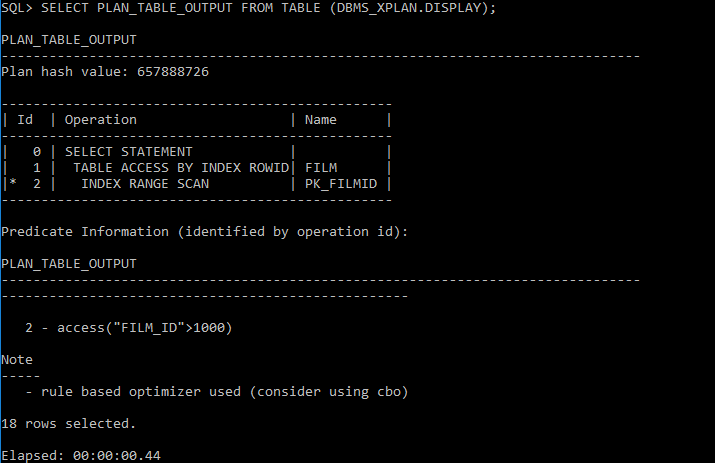
It is clearly that, the query with index will be faster than without index. An index is a data structure that stores the values for a specific column in a table.

## Task 5 – Execution Plan

TASK 5.1

EXPLAIN PLAN FOR SELECT /\*+RULE\*/ \* FROM FILM WHERE FILM\_ID > 1000;

SELECT PLAN\_TABLE\_OUTPUT FROM TABLE (DBMS\_XPLAN.DISPLAY);



The Rule-Based Optimizer is a set of rules to determine how to execute a query. If there is an index, then the Rule-Based Optimizer will use this index to search.

INDEX (UNIQUE SCAN) - Index PK\_FILMID is used in a unique scan operation to evaluate the WHERE clause criteria. It returns exactly one row id from the index.

• TABLE ACCESS BY INDEX ROWID - Rows are located using index; note that full table is not accessed as in the previous case.

• The SELECT statement returns rows satisfying the WHERE clause conditions.

• Cost–It shows the cost of the operation as estimated by the optimizer's query approach, which is a weighted value used to compare costs of execution plans. The value of this column is a function of the CPU\_COST and IO\_COST columns. It does not have any particular unit of measurement.

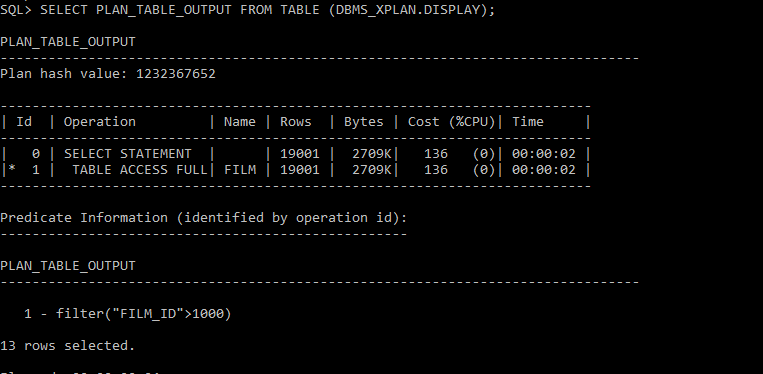
• Bytes–It represents the number of bytes accessed by the operation, as estimated by the query optimization approach.

• Time – It depicts elapsed time in seconds of the operation as estimated by query optimization.

TASK 5.2

EXPLAIN PLAN FOR SELECT \* FROM FILM WHERE FILM\_ID > 1000;

SELECT PLAN\_TABLE\_OUTPUT FROM TABLE (DBMS\_XPLAN.DISPLAY);

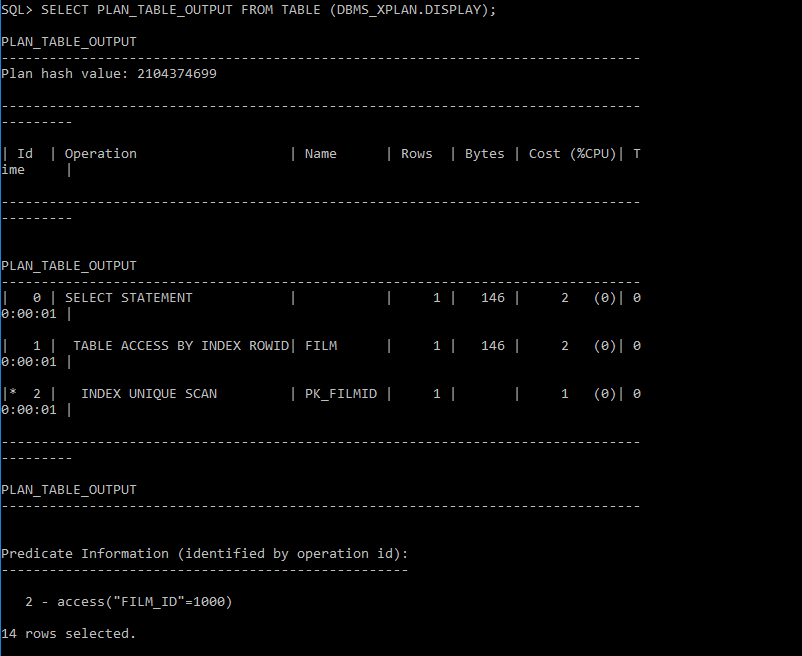


Cost-based optimizer is better than rule-based optimizer. Since the CBO determines which execution plan is most efficient by considering available access paths and by factoring in information based on statistics for the schema objects accessed by the sql statement.

TASK 5.3

EXPLAIN PLAN FOR SELECT \* FROM FILM WHERE FILM\_ID = 1000;

SELECT PLAN\_TABLE\_OUTPUT FROM TABLE (DBMS\_XPLAN.DISPLAY);



TASK 5.4

Task 5.1 used Rule-based optimizer, index PK\_FILMID is used to access the data which FILM\_ID is greater than 1000 in FILM table. Hence, FILM tables are not fully accessed. In task 5.2, table is fully accessed, all records are scanned.

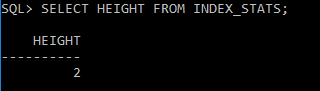
Task 5.5

In task 5.2, all records in FILM are scanned, since task 5.2 wants to generate the data which FILM\_ID is bigger than 1000. But in task 5.3, index PK\_FILMID is used in unique scan, so just one record is returned.

TASK 5.6.A

ANALYZE INDEX PK\_FILMID VALIDATE STRUCTURE;

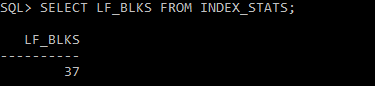
SELECT HEIGHT FROM INDEX\_STATS;



TASK 5.6.B

ANALYZE INDEX PK\_FILMID VALIDATE STRUCTURE;

SELECT LF\_BLKS FROM INDEX\_STATS;



TASK5.6.C

SELECT BLOCKS FROM USER\_TABLES WHERE TABLE\_NAME = 'FILM';

