GROUP FELLOWSHIP OF THE RING

Assignment 2

Dennis Thinh Tan Nguyen William Diedrichsen Marstrand, Thor Valentin Olesen, Pernille Lous, Nicklas Johansen

Problem 1: Relational Algebra

Task 1

- (b)

Task 2

- (b)

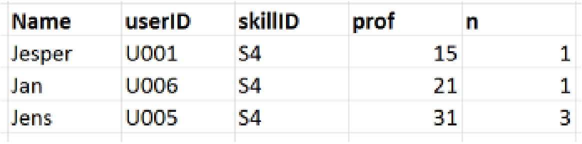
Task 3

a) The expression joins the two relations Endorsement and Member which gives us a relation with the following attributes: *userID, Name, skillID, prof* and *n* where *n* is less than four.

b) There are five columns with the following attributes: *userID*, *name*, *skillID*, *prof* and *n*.

c) There are 3 tuples

d)



Name	userID	skillID	prof	n
Jesper	U001	S4	15	1
Jan	U006	S4	21	1
Jens	U005	S4	31	3

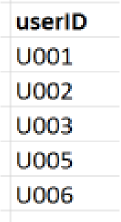
Task 4

a) The expression selects all *userID* and *skillID* from the table Endorsement and divides the relation with the *skillID* belonging to the *userID U005* in the table Endorsement.

b) One column with *UserID*.

c) 5 tuples

d)



userID
U001
U002
U003
U005
U006

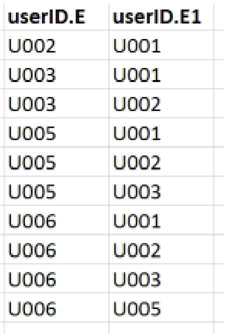
Task 5

a) Creates two temporary tables of Endorsement named E and E1.
Joins E and E1 where *skillID* in E is the same as *skillID* in E1 and *userID* in E is greater than *userID* in E1.
Then selects all userID in E and E1.

b) A table with two columns *SkillID* and *userID*

c) There are 10 tuples

d)



userID.E	userID.E1
U002	U001
U003	U001
U003	U002
U005	U001
U005	U002
U005	U003
U006	U001
U006	U002
U006	U003
U006	U005

Problem 2: Relational Tuple Calculus

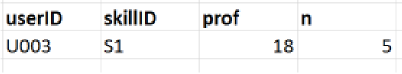
Task 1

a) The expression gets a tuple t from the table endorsement where the skillID is equal to S1.

b) 4 columns named *userID, skillID, prof* and *n.*

c) 1 tuple

d)



userID	skillID	prof	n
U003	S1	18	5

Task 2

a) This question contains errors and has there been omitted *(By request of TA.)*

Task 3

a) This expression gives a set of tuples where *userID* from endorsement is equal to *userID* from Members and *skillID* in endorsement tuple is equal to ”*S1*” and *prof* in endorsement is greater than 20. *name* in tuple t is equal to *name* in Member. Since there is in fact only one tuple with *skillID* equal to “*S1*”, but the tuple has a *prof* attribute less than 20, t will be an empty set.

b) Since t is an empty set the answer is 0

c) No tables has been created.

Problem 3: SQL Queries

Exercise 1:

```
        SELECT COUNT(*) FROM danishMovies;
```

Exercise 2:

```
        SELECT year, COUNT(imdbVotes) FROM movie GROUP BY year;
```

Exercise 3:

```
#-- Helper-query:

        SELECT DISTINCT role FROM involved;

        #--Only roles are actor and director

#-- Query:
        SELECT COUNT(*) FROM person WHERE name LIKE 'C%';

#-- Comment: DISTINCT could have been used upon duplicate occurrences.
```

Exercise 4:

```
        SET @pulpID = (SELECT id FROM movie WHERE title LIKE 'Pulp Fiction');
        SELECT name,birthdate FROM person
        INNER JOIN involved ON personId = id WHERE movieId = @pulpID AND birthdate IS NOT NULL ORDER BY
        birthdate;

        #--Some of the persons dosen't have a birthdate in the database. These are thrown away in the
        query.
```

Exercise 5:

```
        SET @UmaID = (select id from person where name like 'Uma Thurman'); #Uma ID
        SET @JohnID = (select id from person where name like 'John Travolta'); #John ID

        #--Exclusive starring

        SELECT title FROM
              (SELECT movieId FROM involved WHERE personId=@UmaID
               AND movieId IN
                    (SELECT movieId FROM involved WHERE personId=@JohnID))
                          AS ids LEFT JOIN movie ON ids.movieId=id;
```

Exercise 6:

```
SELECT name FROM person, involved AS personInv1
WHERE person.id = personInv1.personId AND personInv1.role = "Actor" AND personInv1.movieId =
(SELECT id FROM movie WHERE title = "Pulp Fiction")AND NOT EXISTS
(SELECT 1 FROM involved AS personInv2, involved AS personInv3, involved AS personInv4 WHERE
personInv2.role = "Actor" AND personInv3.role = "Actor" AND personInv4.role = "Actor" AND
personInv2.movieId = personInv1.movieId AND personInv2.personId != personInv1.personId AND
personInv3.personId = personInv1.personId AND personInv4.personId = personInv2.personId AND
personInv3.movieId != personInv1.movieId AND personInv4.movieId = personInv3.movieId);


#-- This should return an empty list but we seem to get an endless loop.
```

Exercise 7:

```
SET @SamuelID = (select id from person where name like 'Samadu Jackson');
SET @JohnID = (select id from person where name like 'John Travolta');

SELECT title, year FROM
        (select movieId from involved where personId=@SamuelID
        AND movieId IN
            (select movieId from involved where personId=@JohnID))
               AS ids LEFT JOIN movie on ids.movieId=id WHERE year >= 1980 ORDER BY title, year;
```

Exercise 8:

```
SELECT title, ImdbRank FROM movie WHERE year >= 1990
        AND year < 2000 ORDER BY imdbRank DESC LIMIT 5;
```

Exercise 9:

```
SELECT movie.language, ROUND(AVG(imdbRank), 2)  AS 'Average Rating'
        FROM movie WHERE movie.year = 1994 GROUP BY movie.language ORDER BY AVG(imdbRank) DESC;
```

Exercise 10:

```
SET @JohnId = (SELECT id FROM person WHERE name LIKE 'John Travolta');
SELECT TITLE, MAX(imdbRank) FROM movie WHERE id
        IN(SELECT movieId FROM involved WHERE personId = @JohnId);
```

Exercise 11:

```
SET @CCbirth = (SELECT birthdate FROM person WHERE name LIKE 'Charles Chaplin');
SET @CCdeath = (SELECT deathdate FROM person WHERE name LIKE 'Charles Chaplin');

SELECT count(*) FROM person WHERE gender = 'f' AND birthdate>@CCdeath OR deathdate<@CCbirth;
```

Exercise 12:

```
SELECT name, MIN(height) FROM person WHERE id IN (SELECT personId FROM involved WHERE movieId in
(SELECT movieId FROM genre WHERE genre='action') AND role='actor');
```

Exercise 13:

```
SELECT genre, AVG(imdbRank) FROM (SELECT * FROM genre INNER JOIN movie ON genre.movieId =
movie.id) as avgRankByGenre GROUP BY avgRankByGenre.genre;
```

Exercise 14:

```
SELECT genre.genre, sum(approvedMovies.count) votes FROM genre,
        (SELECT movie.id, userRating.count FROM movie,
        (SELECT movie.id, count(*) count FROM movie, ratings WHERE movie.id = ratings.movieId GROUP
        BY movie.id)
userRating WHERE movie.id = userRating.id AND userRating.count > 10) AS approvedMovies WHERE
approvedMovies.id = genre.movieId GROUP BY genre.genre ORDER BY votes DESC;
```

Exercise 15:

```
SELECT movie.title, count(mr2.toID) AS refNumber FROM movieref mr1, movieref mr2, movie
WHERE mr1.fromId = movie.id and mr1.toId = mr2.fromId GROUP BY mr1.fromId
ORDER BY refNumber desc LIMIT 1;
```

Exercise 16:

```
SELECT Count(*) AS "ActorAsDirector" FROM person
WHERE person.id IN (SELECT Distinct personID FROM involved WHERE role = 'director')
AND person.id IN (SELECT distinct personID FROM involved WHERE role = 'actor');
```

Exercise 17:

```
SELECT a.genre, b.genre, count(*) FROM genre a INNER JOIN genre b ON a.movieId = b.movieId
        AND a.genre < b.genre GROUP BY a.genre, b.genre ORDER BY count(*) DESC LIMIT 1
```

Problem 4: Indexing

To determine what tables would benefit from indexing, we have looked at the size of each table;

Size chart

Involved	1.997.903
Person	167.010
Movieref	140.820
Genre	129.962
Movie	59.284
Ratings	486
danishMovies	251
Popular	74

Example

This tells us that the involved table should probably be indexed if used extensively in our queries. By way of example, exercise 6 requires for us to find all people starred in the '*Pulp Fiction'* movie and filter out those who are starred together only in that movie. This requires extensive use of lookups in the involved table that holds the *id* for movies and people. The query takes around 1 minute without indexing. However, if we create an index on *movieId*, *personId* and a composite index of both, it would only take 0,01 seconds.

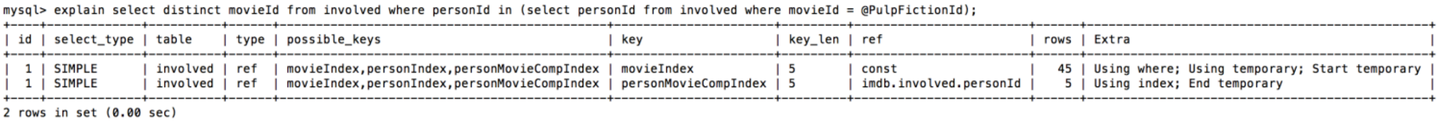
Index creation

Notation: `CREATE index <indexname> ON <tablename>(attribute);`

Indices:

```
CREATE index movieIndex ON involved (movieId);
CREATE index personIndex ON involved (personId);
CREATE index personMovieIndex ON involved (personId,movieId);
```

EXPLAIN



```
mysql> explain select distinct movieId from involved where personId in (select personId from involved where movieId = @PulpFictionId);
+----+-------------+----------+------+---------------------------------------------+----------------------+---------+------------------------+------+-----------------------------------------------+
| id | select_type | table    | type | possible_keys                               | key                  | key_len | ref                    | rows | Extra                                         |
+----+-------------+----------+------+---------------------------------------------+----------------------+---------+------------------------+------+-----------------------------------------------+
|  1 | SIMPLE      | involved | ref  | movieIndex,personIndex,personMovieCompIndex | movieIndex           | 5       | const                  |   45 | Using where; Using temporary; Start temporary |
|  1 | SIMPLE      | involved | ref  | movieIndex,personIndex,personMovieCompIndex | personMovieCompIndex | 5       | imdb.involved.personId |    5 | Using index; End temporary                    |
+----+-------------+----------+------+---------------------------------------------+----------------------+---------+------------------------+------+-----------------------------------------------+
2 rows in set (0.00 sec)
```

The *'EXPLAIN'* keyword shows us that the movieIndex and composite key of movieIndex and personIndex is used to execute the query.

Timing

Query for finding movies that people from Pulp Fiction have starred in:

```
	SET @PulpFictionId = (select id from movie where title = 'pulp fiction');
	SELECT DISTINCT movieId from involved WHERE personID IN
		(SELECT personId FROM involved WHERE movieId = @PulpFictionId);
```

	Time
Without Indexing	~73 seconds
With indexing	0,01 seconds
Time to create indices	10 seconds

To conclude, the index speeds up the query from 1 minute and 13 seconds to 0,01 seconds.