# Carnegie Mellon University
# 15-415 - Database Applications
# Fall 2009, Faloutsos
# Assignment 7: Query Optimization

**Due: 10/27, 1:30pm, Hard-copy only**

## 1 Reminders

- Weight: **15%** of the homework grade.
- Out of **100** points.
- Lead TA: Leman Akoglu
- Estimated time: **3-6 hours**.
- For any questions contact (`lakoglu@cs.cmu.edu`), or use the `blackboard` system.

## 2 What to hand in:

- Only a hard-copy please. Type your answers as much as you can.
- Illegible handwriting may get no points, at the discretion of the grader. Only drawings may be hand-drawn, as long as they are neat and legible.

## 3 Getting started

In this assignment we will (again!) work with a subset of the Netflix database which contains the following 3 tables:

```
movies (mid, title, year)

users (uid, lastname, firstname, age)

ratings (mid, uid, rating, timestamp)
```

where `mid` is the unique id for each movie, `title` is the movie's title, `year` is the movie's year-of-release; `uid` is the unique id for each user, `firstname` and `lastname` are the first and last name of the user, respectively and `age` is the user's age; the attribute `rating` is an integer ranging between 1 and 5 and `timestamp` denotes the time when the user rated the movie.

We have provided you with a script to create the schema and load the tables into the database. Please do the following:

- Log in to your account on `newcastle.db.cs.cmu.edu` (using the login and password from Assignment 3).
- Run ~`lakoglu415/setup_db.sh`, press ``y'' to continue when prompted.
- Run `pg_ctl start -o -i`, then press ``Enter'' and then run `psql`.
- Update the database statistics (i.e., `VACUUM`, `ANALYZE`).

*SANITY CHECK:*

```
select count(*) from movies;
select count(*) from users;
select count(*) from ratings;
```

*The commands above should return 97 records in table* `movies`*, 2605 records in table* `users` *and 35000 records in table* `ratings`*.*

## 4 Resources

The following documents are EXTREMELY useful for the purpose of this assignment.

- Check the statistics collected by PostgreSQL:

  `http://www.postgresql.org/docs/8.3/static/planner-stats.html`

- Syntax of EXPLAIN command:

  `http://www.postgresql.org/docs/8.3/static/sql-explain.html`

- How to use EXPLAIN command and understand its output:

  `http://www.postgresql.org/docs/8.3/static/performance-tips.html`

- Create an Index for a table:

  `http://www.postgresql.org/docs/8.3/static/sql-createindex.html`

- Create a Clustered-Index for a table:

  `http://www.postgresql.org/docs/8.3/interactive/sql-cluster.html`

- Understanding Joins:

  `http://stanford.edu/dept/itss/docs/oracle/10g/server.101/b10752/optimops.htm#39473`

## 5 Exercises

Q1 [**10 points**] *Examining the system catalogs.*

We will begin by inspecting the statistics PostgreSQL has for the table `ratings`.

(a) [**2 points**] How many indexes are built on `ratings`? Name them and write down their type.

(b) [**3 points**] What is the number of pages occupied by `ratings`? How many pages does its index use? Write down the query you used to find the above information.

(c) [**3 points**] What is the number of distinct values for each of the columns of `ratings`? Write down the query you used to find the above information and explain its output.

(d) [**2 points**] Write down your **own** queries to find the number of distinct values of each of the columns of `ratings` without using the PostgreSQL catalog. What are your observations?

Q2 [**10 points**] *Executing an exact match query.*

Use the PostgreSQL command EXPLAIN to examine how the optimizer treats the following query:

```
select * from ratings where rating=1;
```

(a) [**2 points**] What is the estimated result cardinality of this query? Run the query and report the number of rows it actually returns.

(b) [**2 points**] According to EXPLAIN, what is the estimated total cost of executing the best plan for this query? What do the two numbers mean?

(c) [**4 points**] How does the query optimizer derive the above cost values? Write down the formula and give a brief description.

(d) [**2 points**] Using our tree/relational-algebra notation, draw the execution plan selected by the optimizer.

Q3 [**15 points**] *Executing an Index Scan.*

Create a (non-clustered) index on the attribute `rating` of table `ratings`. Update the statistics by running VACUUM and then ANALYZE.

(a) [**1 points**] How many disk pages does the new index occupy?

(b) [**3 points**] Run EXPLAIN on the query from Exercise 2. Which access method does the query optimizer select now? What is the estimated total cost of executing the best plan for this query?

(c) [**4 points**] What is it that makes the best plan with the index cheaper?

Create a clustered index on the attribute `rating` of table `ratings`. Update the statistics by running VACUUM and then ANALYZE.

(d) [**3 points**] Run EXPLAIN again on the query from Exercise 2. Which access method does the query optimizer select now? What is the estimated total cost of executing the best plan for this query?

(e) [**4 points**] What is it that makes the best plan with the clustered index cheaper than the the best plan with the (non-clustered) index?

Q4 [**10 points**] *Executing a Range Scan.*

Now analyze the query plan that PostgreSQL comes up for the following query:

```
select * from ratings where timestamp < '2002-01-01 00:00:00';
```

Answer the following questions by inspecting the output of EXPLAIN.

(a) [**2 points**] How many tuples in table `ratings` that have `timestamp` < '2002-01-01 00:00:00' does the optimizer think there are?

(b) [**2 points**] How does the optimizer arrive at this estimate of the number of tuples? That is, what calculations does it perform, and where does the supporting data come from?

(c) [**3 points**] In what order will the tuples be returned by this plan? Why?

(d) [**3 points**] How would you improve the efficiency of the above query? What happens to the expected cost after your improvement? Does the order in which the tuples are returned change? Why?

Q5 [**10 points**] *Executing a Join.*

Drop the index on `rating` in table `ratings` from Exercise Q3. Update the statistics by running VACUUM and then ANALYZE and consider the following query (tough raters):

```
SELECT DISTINCT (lastname)
FROM ratings, users
WHERE ratings.uid = users.uid
       AND rating = 1;
```

Answer the following questions by inspecting the output of EXPLAIN.

(a) [**3 points**] Draw the plan selected by the optimizer (copying appropriate output messages from the PostgreSQL prompt). Draw the query tree using relational-algebra notation.

(b) [**2 points**] Which join algorithm is used by the query optimizer? What is its estimated cost? What is the estimated result cardinality of this query?

(c) [**3 points**] Disable the join algorithm selected by the optimizer (i.e., the answer to part (b) by using the `SET` command. Which join algorithm is used now? What is the total estimated cost? Draw the query tree again using relational algebra.

(d) [**2 points**] Now disable both join algorithms comprising your answers to parts (b) and (c). Which join algorithm is used now? What is the total estimated cost?

Q6 [**15 points**] *Understanding Join Selection*

Enable all the disabled join algorithms in Exercise Q5 and consider the following two queries:

Query-6.1

```
SELECT avg(age) as avgAge, lastname
FROM ratings,users
WHERE ratings.uid=users.uid
GROUP BY lastname
ORDER BY avgAge;
```

Query-6.2

```
SELECT *
FROM ratings,users
WHERE ratings.uid=users.uid
ORDER BY users.uid;
```

(a) [**5 points**] Before executing EXPLAIN on Query-6.1, state which method do you expect the query optimizer to select. Which join algorithm does the optimizer actually pick? What is the total estimated cost? Disable the join algorithm selected by the optimizer. Which join algorithm is used now?

(b) [**5 points**] Enable the join algorithm that you disabled in (a). Before executing EXPLAIN on Query-6.2, state which method do you expect the query optimizer to select. Which join algorithm does the optimizer actually pick? What is the total estimated cost? Disable the join algorithm selected by the optimizer. Which join algorithm is used now?

(c) [**5 points**] Why did the optimizer pick *different* join algorithms for the given queries in (a) and (b)?

Q7 [**15 points**]*Magic Sets (MS)*

Consider the following SQL query which finds the users with `lastname` "Rich" and who have rated more than 10 movies:

Query-7.1

```
SELECT uid
FROM users
WHERE lastname = 'Rich' AND
   10 < (SELECT COUNT(*)
   FROM ratings
   WHERE ratings.uid = users.uid)
```

There is a nested subquery in the WHERE clause which computes the total number of movies rated by a particular user. This type of subquery is also called a *correlated subquery* (CS) because it uses parameters (here the column `uid`) from a table outside of the subquery. We will be concentrating on the optimization of this class of queries.

(a) [**2 points**] According to EXPLAIN, what is the estimated total cost of executing the best plan for the above query?

As you might have guessed, Correlated Execution is commonly considered as an inferior strategy, as it involves blind per-row processing instead of a set-oriented strategy. Let's try to develop a better strategy for the above query.

Note that a nested query has the following structure:

```
for_each (x elementof X) {
      SubqueryResult = CS(x);
      Process(SubqueryResult);
}
```

Here, x is the correlation attribute (`uid` in Query-7.1) and X is the set of values with which the correlated subquery (CS) is invoked. Note that the primary aim of decorrelation is to decouple the execution of CS from the execution of the outer query block. We can do so by writing an SQL query which generates a view MS (Magic Set) which stores the computed value CS(x) of all the distinct values of x in X.

(b) [**5 points**] Write an SQL query to create a view `MS(uid, numratings)` which finds the total number of movies rated by each user.

(c) [**5 points**] Write a final SQL query (call it Query-7.2) using the view MS above which is semantically same as Query-7.1, i.e. which finds users whose `lastname` is "Rich" and who have rated more than 10 movies, but does not use any nested subqueries.

(d) [**3 points**] According to EXPLAIN, what is the estimated total cost of executing the best plan for Query-7.2?

Q8 [**15 points**]*Hands-on Query Optimization*

Lets define the total number of users who have more number of ratings than that of a particular user to be the `quasiRank` of that user. For example, user `Smith`'s `quasiRank` is 32 if there are 32 users with greater number of ratings than that of `Smith`. Note that there might be multiple users with the same `quasiRank` (those with the same number of ratings).

The following SQL query finds the `uid`, `lastname`, and number of ratings (`numratings`) of users with `quasiRank` $\leq 5$.

Query-8.1

```
SELECT uid, lastname, numratings
FROM   (SELECT users.uid, lastname, count(*) as numratings
        FROM users, ratings
        WHERE users.uid=ratings.uid
        GROUP BY users.uid, lastname) AS RATECOUNTS
WHERE 5 >= (SELECT count(*)
            FROM (SELECT users.uid, lastname, count(*) as numratings
                  FROM users, ratings
                  WHERE users.uid=ratings.uid
                  GROUP BY users.uid, lastname) AS RATECOUNTS2
            WHERE RATECOUNTS.numratings < RATECOUNTS2.numratings)
ORDER BY uid;
```

(a) [**2 points**] According to EXPLAIN, what is the estimated total cost of executing the best plan for the above query?

(b) [**5 points**] Write an SQL query to create a view `MS(uid, quasiRank)` which finds the count of more prolific users (with higher number of ratings) for each user. Note that the count for the (most prolific) user with the most number of ratings should be 0.

(c) [**5 points**] Write a final SQL query (call it Query-8.2) using the view MS above which will produce the same result as Query-8.1, but in a more efficient way without using any nested subqueries.

(d) [**3 points**] According to EXPLAIN, what is the estimated total cost of executing the best plan for Query-8.2?

*Remark: There is an even more efficient SQL query without using Magic Sets that will produce the same result as Query-8.1, but you do not need to worry about it in this assignment.*