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.

Answer: There are two indexes:

```
  index name | type
-------------+---------
 ratings_uid | B-tree
 ratings_mid | B-tree
```

(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.

Answer:

```
SELECT relname, relkind, relpages FROM pg_class WHERE relname LIKE 'ratings%';
   relname   | relkind | relpages
-------------+---------+----------
 ratings     | r       |      223
 ratings_uid | i       |       79
 ratings_mid | i       |       98
(3 rows)
```

(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.

Answer:

```
(SELECT attname, n_distinct
ROM pg_stats
WHERE tablename='ratings' and n_distinct>0)
 UNION
(SELECT attname, CEIL( -n_distinct*35000)
FROM pg_stats
WHERE tablename='ratings' and n_distinct<0);

  attname  | n_distinct
-----------+------------
 mid       |         83
 rating    |          5
 timestamp |       7029
 uid       |       2519
(4 rows)
```

(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?

Answer:

```
select count(*) from (select distinct mid from ratings) as foo;
 count
-------
    93
(1 row)

select count(*) from (select distinct rating from ratings) as foo;
 count
-------
     5
(1 row)

select count(*) from (select distinct timestamp from ratings) as foo;
 count
-------
 12063
(1 row)

select count(*) from (select distinct uid from ratings) as foo;
 count
-------
  2605
(1 row)
```

We see that the PostgreSQL catalog statistics are not *exact*, but good approximations of actual values.

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.

Answer:

```
explain select * from ratings where rating=1;
                          QUERY PLAN
-------------------------------------------------------------
```

```
 Seq Scan on ratings  (cost=0.00..660.50 rows=1727 width=24)
   Filter: (rating = 1)
(2 rows)

select count(*) from ratings where rating=1;
 count
-------
  1650
(1 row)
```

Estimated result cardinality = 1727 Actual number of rows = 1650

(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?

Answer:

```
cost=0.00..660.50
```

The first number (0.00) is the estimated start-up cost (Time expended before output scan can start, e.g., time to do the sorting in a sort node.) The second number (660.50) is the estimated total cost (If all rows were to be retrieved, though they might not be: for example, a query with a LIMIT clause will stop short of paying the total cost of the Limit plan node's input node.)

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

Answer: The estimated cost is:

```
(disk pages read * seq\_page\_cost) + (rows scanned * cpu\_tuple\_cost) +
(rows scanned * cpu\_operator\_cost)
  = (223 * 1.0 (default)) + (35000 * 0.01 (default))
    + (35000 * 0.0025 (default))
  = 660.50
```

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

Answer:

```
        Seq Scan, Filter: rating = 1
           |
        ratings
```

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.

```
create index ratings_rating on ratings(rating);
VACUUM;
ANALYZE;
```

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

Answer: 79 pages.

```
SELECT relname, relkind, relpages FROM pg_class WHERE relname LIKE 'ratings%';
    relname     | relkind | relpages
----------------+---------+----------
 ratings        | r       |      223
 ratings_uid    | i       |       79
 ratings_mid    | i       |       98
 ratings_rating | i       |       79
(4 rows)
```

(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?

Answer: Bitmap Index Scan followed by Bitmap Heap Scan Estimated total cost: 272.10

```
explain select * from ratings where rating=1;
                                   QUERY PLAN
--------------------------------------------------------------------------------
 Bitmap Heap Scan on ratings  (cost=28.82..272.10 rows=1622 width=24)
   Recheck Cond: (rating = 1)
   ->  Bitmap Index Scan on ratings_rating  (cost=0.00..28.42 rows=1622 width=0)
         Index Cond: (rating = 1)
(4 rows)
```

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

Answer: B-tree index (default method) created on attribute `rating` helps to avoid sequential scanning which reduces cost.

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

```
cluster ratings_rating on ratings;
VACUUM;
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?

Answer: Index Scan Estimated total cost: 67.10

```
explain select * from ratings where rating=1;
                                   QUERY PLAN
----------------------------------------------------------------------------
 Index Scan using ratings_rating on ratings  (cost=0.00..67.10 rows=1820 width=24)
   Index Cond: (rating = 1)
(2 rows)
```

(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?

Answer: The clustered index on attribute `rating` groups records with the same `rating` in consecutive pages in disk. So, once the first record with `rating = 1` is found, the disk is read sequentially. On the other hand, the B tree index only provides pointers to pages with records `rating = 1` which may reside in many different pages. This increases the number of accesses, i.e. the total cost.

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?

Answer: 1598

```
                                    QUERY PLAN
-------------------------------------------------------------------------------------
 Seq Scan on ratings  (cost=0.00..660.50 rows=1598 width=24)
   Filter: ("timestamp" < ’2002-01-01 00:00:00’::timestamp without time zone)
(2 rows)
```

(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?

Answer: The optimizer first gets the histogram information on `timestamp`.

```
 select histogram_bounds from pg_stats where tablename=’ratings’ and attname=’timestamp’;
       histogram_bounds
-----------------------------------------------
 {"2000-01-08 06:03:00","2004-04-18 10:45:00","2009-02-06 12:20:00","2009-04-12 22:32:00",
 "2009-05-12 10:45:00","2009-06-20 23:44:00","2009-07-19 09:26:00","2009-08-15 02:41:00",
 "2009-09-12 11:31:00","2009-10-20 23:44:00","2009-12-30 20:07:00"}
(1 row)
```

Next, the optimizer computes the *selectivity* (between 0 and 1) by calculating the bucket as well as the location wihtin that bucket in which the given timestamp `’2002-01-01 00:00:00’` could reside. The optimizer assumes linear distribution within histogram columns. The selectivity is then multiplied by the cardinality of `ratings` to obtain the expected cardinality (estimated number of rows).

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

Answer: Since the query plan selected for this query is Sequential Scan, the tuples will be returned in the order they are stored in disk.

(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?

Answer: The efficiency of the above query can be improved by creating a (clustered) index on `timestamp`. The order of the tuples returned by the query would change with a clustered index, because this time tuples are grouped on disk based on their `timestamp`.

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):

```
drop index ratings_rating;
VACUUM;
ANALYZE;

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.
Answer:

```
                                       QUERY PLAN
-------------------------------------------------------------------------------------
 Unique  (cost=835.67..842.85 rows=1435 width=7)
   ->  Sort  (cost=835.67..839.26 rows=1435 width=7)
         Sort Key: users.lastname
         ->  Hash Join  (cost=76.61..760.43 rows=1435 width=7)
               Hash Cond: (ratings.uid = users.uid)
               ->  Seq Scan on ratings  (cost=0.00..660.50 rows=1435 width=4)
                     Filter: (rating = 1)
               ->  Hash  (cost=44.05..44.05 rows=2605 width=15)
                     ->  Seq Scan on users  (cost=0.00..44.05 rows=2605 width=15)
(9 rows)
                        Unique
                          |
                        Sort
                        Sort Key:users.lastname
                          |
                     Hash Join
```

```
                Hash Cond: ratings.uid = users.uid
              /                      \
           Seq Scan   Hash
           Filter: (rating = 1)        \
             |                       Seq Scan
          ratings             |
                           users
```

(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?
Answer: Join algorithm: Hash Join
Estimated cost: 842.85
Estimated result cardinality: 1435

(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.
Answer:

```
set enable_hashjoin=false;

                                          QUERY PLAN
-----------------------------------------------------------------------------------------------
 Unique  (cost=939.35..946.52 rows=1435 width=7)
   ->  Sort  (cost=939.35..942.94 rows=1435 width=7)
         Sort Key: users.lastname
         ->  Merge Join  (cost=735.74..864.11 rows=1435 width=7)
               Merge Cond: (users.uid = ratings.uid)
               ->  Index Scan using users_uid on users  (cost=0.00..100.33 rows=2605 width=15)
               ->  Sort  (cost=735.74..739.33 rows=1435 width=4)
                     Sort Key: ratings.uid
                     ->  Seq Scan on ratings  (cost=0.00..660.50 rows=1435 width=4)
                           Filter: (rating = 1)
(10 rows)
```

Join algorithm: Merge Join
Estimated cost: 946.52

```
                   Unique
                     |
                   Sort
                   Sort Key:users.lastname
                     |
                Merge Join
                Merge Cond: users.uid = ratings.uid
              /    \
            Sort Index Scan
            Sort Key:ratings.uid |
             |                     users_uid
           Seq Scan   |
           Filter: (rating = 1)         users
             |
ratings
```

(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?
Answer:

```
set enable_mergejoin=false;

                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Unique  (cost=1250.12..1257.29 rows=1435 width=7)
   ->  Sort  (cost=1250.12..1253.70 rows=1435 width=7)
         Sort Key: users.lastname
         ->  Nested Loop  (cost=0.00..1174.87 rows=1435 width=7)
               ->  Seq Scan on ratings  (cost=0.00..660.50 rows=1435 width=4)
                     Filter: (rating = 1)
               ->  Index Scan using users_uid on users  (cost=0.00..0.35 rows=1 width=15)
                     Index Cond: (users.uid = ratings.uid)
(8 rows)
```

Join algorithm: Nested Loop Join
Estimated cost: 1257.29

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

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

```
set enable_hashjoin=true;
set enable_mergejoin=true;
```

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?

Answer: Join algorithm: Hash Join
Estimated cost: 1552.86

```
                                   QUERY PLAN
---------------------------------------------------------------------------------
 Sort  (cost=1547.22..1552.86 rows=2256 width=11)
   Sort Key: (avg(users.age))
   ->  HashAggregate  (cost=1393.36..1421.56 rows=2256 width=11)
         ->  Hash Join  (cost=76.61..1218.36 rows=35000 width=11)
               Hash Cond: (ratings.uid = users.uid)
               ->  Seq Scan on ratings  (cost=0.00..573.00 rows=35000 width=4)
               ->  Hash  (cost=44.05..44.05 rows=2605 width=19)
                     ->  Seq Scan on users  (cost=0.00..44.05 rows=2605 width=19)
(8 rows)

set enable_hashjoin=false;
```

Join algorithm: Merge Join

```
                                            QUERY PLAN
---------------------------------------------------------------------------------------------------
 Sort  (cost=2606.44..2612.08 rows=2256 width=11)
   Sort Key: (avg(users.age))
   ->  HashAggregate  (cost=2452.58..2480.78 rows=2256 width=11)
         ->  Merge Join  (cost=0.00..2277.58 rows=35000 width=11)
               Merge Cond: (users.uid = ratings.uid)
               ->  Index Scan using users_uid on users  (cost=0.00..100.33 rows=2605 width=19)
               ->  Index Scan using ratings_uid on ratings  (cost=0.00..1733.25 rows=35000 width=4)
(7 rows)
```

(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?

Answer:

```
set enable_hashjoin=true;
```

Join algorithm: Merge Join
Estimated cost: 2277.58

```
                                      QUERY PLAN
-----------------------------------------------------------------------------------------
 Merge Join  (cost=0.00..2277.58 rows=35000 width=49)
   Merge Cond: (users.uid = ratings.uid)
   ->  Index Scan using users_uid on users  (cost=0.00..100.33 rows=2605 width=25)
   ->  Index Scan using ratings_uid on ratings  (cost=0.00..1733.25 rows=35000 width=24)
(4 rows)

set enable_mergejoin=false;
```

Join algorithm: Nested Loop Join

```
                                   QUERY PLAN
----------------------------------------------------------------------------------
 Nested Loop  (cost=0.00..3061.91 rows=35000 width=49)
   ->  Index Scan using users_uid on users  (cost=0.00..100.33 rows=2605 width=25)
   ->  Index Scan using ratings_uid on ratings  (cost=0.00..0.96 rows=14 width=24)
```

```
          Index Cond: (ratings.uid = users.uid)
(4 rows)
```

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

Answer: The optimizer picks a Hash join for Query-6.1 and a Merge join for Query-6.2 as a result of the order in which the output of the queries are sorted. In Query-6.1, the output has to be sorted *after* the join since the `avgAge` on which the output tuples are sorted is not considered in the join condition (ratings.uid = users.uid). On the other hand, in Query-6.2, the output is sorted on `uid` which is a part of the join condition. Therefore, the sorting can be done *during* the join, and hence the Merge join was used.

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?

Answer: 130465.05

```
                                          QUERY PLAN
-------------------------------------------------------------------------------------------
 Seq Scan on users  (cost=0.00..130465.05 rows=1 width=8)
   Filter: (((lastname)::text = 'Rich'::text) AND (10 < (subplan)))
   SubPlan
     ->  Aggregate  (cost=50.05..50.06 rows=1 width=0)
           ->  Bitmap Heap Scan on ratings  (cost=4.36..50.01 rows=14 width=0)
                 Recheck Cond: (uid = $0)
                 ->  Bitmap Index Scan on ratings_uid  (cost=0.00..4.36 rows=14 width=0)
                       Index Cond: (uid = $0)
(8 rows)
```

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.

Answer:

```
CREATE VIEW MS  AS
SELECT uid, count(*) AS numratings
FROM ratings GROUP BY uid;
```

(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.

Answer:

```
SELECT users.uid
FROM users, MS
WHERE MS.uid = users.uid
AND lastname = 'Rich' AND numratings>10;

 uid
------
 2398
 1393
 2402
 2385
(4 rows)
```

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

Answer: 965.02

```
                               QUERY PLAN
-------------------------------------------------------------------------
 Hash Join  (cost=886.08..965.02 rows=1 width=8)
   Hash Cond: (ratings.uid = users.uid)
   ->  HashAggregate  (cost=835.50..879.71 rows=2526 width=4)
```

```
            Filter: (count(*) > 10)
            ->  Seq Scan on ratings  (cost=0.00..573.00 rows=35000 width=4)
      ->  Hash  (cost=50.56..50.56 rows=1 width=8)
            ->  Seq Scan on users  (cost=0.00..50.56 rows=1 width=8)
                  Filter: ((lastname)::text = 'Rich'::text)
(8 rows)
```

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` ≤ 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?
Answer: 4290788.15

```
                                           QUERY PLAN
-----------------------------------------------------------------------------------------------
 Sort  (cost=4290785.98..4290788.15 rows=868 width=434)
   Sort Key: ratecounts.uid
   ->  Subquery Scan ratecounts  (cost=1480.86..4290743.61 rows=868 width=434)
         Filter: (5 >= (subplan))
         ->  HashAggregate  (cost=1480.86..1513.42 rows=2605 width=15)
               ->  Hash Join  (cost=76.61..1218.36 rows=35000 width=15)
                     Hash Cond: (public.ratings.uid = public.users.uid)
                     ->  Seq Scan on ratings  (cost=0.00..573.00 rows=35000 width=4)
                     ->  Hash  (cost=44.05..44.05 rows=2605 width=15)
                           ->  Seq Scan on users  (cost=0.00..44.05 rows=2605 width=15)
         SubPlan
           ->  Aggregate  (cost=1646.52..1646.53 rows=1 width=0)
                 ->  HashAggregate  (cost=1568.36..1613.95 rows=2605 width=15)
                       Filter: ($0 < count(*))
```

```
                              ->  Hash Join  (cost=76.61..1218.36 rows=35000 width=15)
                                    Hash Cond: (public.ratings.uid = public.users.uid)
                                    ->  Seq Scan on ratings  (cost=0.00..573.00 rows=35000 width=4)
                                    ->  Hash  (cost=44.05..44.05 rows=2605 width=15)
                                          ->  Seq Scan on users  (cost=0.00..44.05 rows=2605 width=15)
(19 rows)
```

(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.

Answer:

```
CREATE VIEW RCount  AS
SELECT uid, count(*) AS numratings
FROM ratings GROUP BY uid;

CREATE VIEW MS  AS
SELECT R1.uid, count(*) AS quasiRank
FROM RCount AS R1 LEFT OUTER JOIN RCount AS R2
ON R1.numratings < R2.numratings
GROUP BY R1.uid;
```

(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.

Answer:

```
SELECT users.uid, lastname, numratings
FROM users, RCount, MS
WHERE MS.quasiRank <= 5 AND MS.uid=users.uid AND RCount.uid=users.uid
ORDER BY users.uid;

 uid  |  lastname  | numratings
------+------------+------------
  361 | D'Cruze    |         25
  383 | Derrick    |         25
  750 | Heikkinen  |         25
 1044 | Martin     |         25
 1150 | Montgomery |         25
 1153 | Maller     |         25
 1312 | Pervan     |         26
 1671 | Valtonen   |         25
 1688 | Ursin      |         29
 1964 | Colonello  |         25
 2400 | Rains      |         25
(11 rows)
```

(d) [**3 points**] According to EXPLAIN, what is the estimated total cost of executing the best plan for Query-8.2?
Answer: 178153.27

```
                                                  QUERY PLAN
----------------------------------------------------------------------------------------------------------------
 Sort  (cost=178152.79..178153.27 rows=194 width=23)
   Sort Key: users.uid
   ->  Hash Join  (cost=178077.17..178145.41 rows=194 width=23)
         Hash Cond: (public.ratings.uid = users.uid)
         ->  HashAggregate  (cost=748.00..779.58 rows=2526 width=4)
               ->  Seq Scan on ratings  (cost=0.00..573.00 rows=35000 width=4)
         ->  Hash  (cost=177326.67..177326.67 rows=200 width=19)
               ->  Hash Join  (cost=177316.92..177326.67 rows=200 width=19)
                     Hash Cond: (public.ratings.uid = users.uid)
                     ->  HashAggregate  (cost=177240.30..177244.80 rows=200 width=4)
                           Filter: ((count(*) - 1) <= 5)
                           ->  Nested Loop  (cost=1555.36..161129.10 rows=2148161 width=4)
                                 Join Filter: (((count(*)) < r2.numratings) OR (public.ratings.uid = r2.uid))
                                 ->  HashAggregate  (cost=748.00..779.58 rows=2526 width=4)
                                       ->  Seq Scan on ratings  (cost=0.00..573.00 rows=35000 width=4)
                                 ->  Materialize  (cost=807.36..832.62 rows=2526 width=12)
                                       ->  Subquery Scan r2  (cost=748.00..804.84 rows=2526 width=12)
                                             ->  HashAggregate  (cost=748.00..779.58 rows=2526 width=4)
                                                   ->  Seq Scan on ratings  (cost=0.00..573.00 rows=35000 width=4)
                     ->  Hash  (cost=44.05..44.05 rows=2605 width=15)
                           ->  Seq Scan on users  (cost=0.00..44.05 rows=2605 width=15)
(21 rows)
```

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.