Carnegie Mellon University
Department of Computer Science
15-415/615 - Database Applications
C. Faloutsos & A. Pavlo, Spring 2014

Homework 5

IMPORTANT - what to hand in:

- Please submit your answers in
  - **hard copy** in class at **1:30pm on Thursday, March 18th**. The hardcopy should contain the answers to all the questions (including SQL statements and database output, whenever stated by the question)
  - And **electronically** on the "blackboard", by the same date/time.

  The electronic submission should be an archive called `[andrew_id].zip`, containing a folder called `queries` and a file for each question, called `q#.sql`, where '#' is the question number. In each such file you should add the SQL statements that you used in order to answer the question. For your convenience, mirror the structure of the file `http://www.cs.cmu.edu/~epapalex/15415S14/db_hw5_s14.zip`, where you can replace the place-holder scripts `/queries/q#.sql` with your real answers.

Reminders:

- **Plagiarism:** Homework may be discussed with other students, but all homework is to be completed **individually**.
- **Typeset** all of your answers whenever possible. Illegible handwriting or ambiguous answers may get no points, at the discretion of the graders.
- For faster grading, please solve each of the 4 questions on a **separate** page, i.e., 4 pages in total for this homework. Type **course-id, hw-id, question-id**, and your **name and Andrew ID on each** of the 4 pages.
- **Late Homeworks**: If you are turning your homework in late, please email it
  - to all TAs
  - with the subject line exactly `15-415 Homework Submission (HW 5)`
  - the count of slip-days you are using, and the count you have left.

For your information:

- Graded out of **100** points;
- **4** questions total

- Rough time estimate: 4-6 hours (1 - 1.5 hour for each question on average - some questions require less time, some more.)

Revision : 2014/03/29 15:40

Question	Points	Score
Analysis of a simple query	20	
Analysis of a slightly more complex query	20	
Analyze a simple JOIN query	30	
Analyze a slightly more complicated join query	30	
Total:	100	

Homework 5 continues...

Introduction The purpose of this homework is to make you familiar with the query exectution engine of Postgres. In particular, you will have to analyze a few queries, and answer questions regarding their performance when turning different knobs of the execution engine.

In order to answer the questions, you might find the following documentation links useful:

- Documentation of `EXPLAIN ANALYZE`:

  http://www.postgresql.org/docs/9.2/static/sql-explain.html.

- Making sense of the `EXPLAIN ANALYZE` output:

  http://www.postgresql.org/docs/9.2/static/performance-tips.html.

- Postgres query planner documentation:

  http://www.postgresql.org/docs/9.2/static/runtime-config-query.html.

- How to create an index:

  http://www.postgresql.org/docs/9.2/static/sql-createindex.html.

IMPORTANT:

Even though we will be using the same database schema as in homework 2, you should re-run the database setup using the files for homework 5, since the loading script for homework 2 was creating indexes that should not be in place for this homework. You can download the new files from

`http://www.cs.cmu.edu/~epapalex/15415S14/db_hw5_s14.zip`.

In the following lines, for your convenience, we copy the directions from HW2.

Database details - (same directions as in HW2, feel free to skip)

The database contains two tables:

- `movies`(mid, title, year, num_ratings, rating)
- `play_in`(mid, name, cast_position)

The tables contain the obvious information: which actor played in what movie, at what position; for each movie, we have the title (eg., 'Gone with the wind'), year of production, count of rating reviews it received, and the average score of those ratings (a float in the range 0 to 10, with '10' meaning 'excellent').

We will use Postgres, which is installed in the andrew machines. [1]

You need to do the following set up steps.

1. **Get recommended machine and port number:** On the "blackboard" grades table there are two columns, called `ghc##` and `PGPORT`. The first is the number of the `ghc` machine we recommend for you, and the other is the recommended port number for your Postgres server instance. The goal is to avoid conflicts, by having each of you

[1] You may develop your queries on your own, local installation of Postgres, but your solutions will be run and graded on the public installation.

Homework 5 continues...

running on a different machine, and listening on a different port. To log in to the recommended machine, type

```
ssh ghc##.ghc.andrew.cmu.edu
```

replacing `##` by the two digit number you found on "blackboard".

In the remote case of conflicts, choose a port number close to the one assigned to you and try again. The GHC machines from 02 to 81 are located in GHC 3000, 5201, and 5205 and can be accessed through `ssh`. There are also machines in GHC 5208 which can be accessed physically.

2. **Get and edit setup script**: Download the HW5 .zip from here:

   `http://www.cs.cmu.edu/~epapalex/15415S14/db_hw5_s14.zip`.

   Unzip it and copy the folder `db_hw5_s14` (make sure the name is exactly this one) it on your *home directory* of your andrew account. Edit `setup_db.sh` to modify the line that assigns a value to `PGPORT` to the value that you found on blackboard.

3. **Set up the db:** If your default shell is not bash, type `bash` and then

   ```
   bash setup_db.sh.
   ```

   This will start the Postgres server, create a database with your username, create the schema for this homework, and add data to the database.

4. **Test script:** Run

   ```
   bash run_queries.sh.
   ```

   This shell script executes all the queries for this homework (which you have to implement). Inside the folder you unzipped, there is a folder called `queries` which contains place-holder .sql files for all the queries that you have to implement. Replace the place-holders with *your* queries, and submit them in a zip file, as we said in the first page.

5. **Optional testing:** If you want to test/debug queries separately, you can also use the script `test.sh`, included in the homework material.

6. **Starting/stopping** the Postgres server: see the instructions at

   `http://www.cs.cmu.edu/~epapalex/15415S14/PostgreSQLReadme.htm`.

The full documentation for Postgres is at http://www.postgresql.org/docs/.

Homework 5 continues. . .

Question 1: Analysis of a simple query .... [20 points]

Submit on separate page Consider the following query:

```
SELECT title, year
FROM movies
WHERE rating>=9;
```

In this question, we will see that indexes can help speeding up queries, when used appropriately.

Before running any queries, make sure that you have dropped *all indexes* that you might have created!

(a) [**3 points**] Provide the execution plan of the query. Give SQL statement and the execution plan.

(b) Based on the execution plan:
   i. [**1 point**] What is the estimated total cost of the query? (in arbitrary units)
   ii. [**1 point**] What is the total actual time it took to run? (in ms)

(c) [**2 points**] Build an index on attribute `rating` on the `movies` table. Provide the SQL statement.

(d) [**5 points**] Provide the execution plan of the query after you build the index.

(e) [**3 points**] Has the estimated total cost 1) increased, 2) decreased, or 3) stayed the same? (answer 1,2, or 3).

(f) [**2 points**] Has the total actual time it takes to execute the query 1) increased, 2) decreased, or 3) stayed the same? (answer 1,2, or 3).

(g) Let's see what happens if you build an index on attribute `num_ratings` on the `movies` table (Don't drop the index on `rating`).
   i. [**1 point**] Do you expect it to affect the cost of executing the query? (**Yes** or **No**).
   ii. [**2 points**] Build the index (no need to provide the statement). Get the new estimated total cost from the execution plan. Has it 1) increased, 2) decreased, or 3) stayed the same? (answer 1,2, or 3).

Solution:

(a) `EXPLAIN ANALYZE SELECT title, year from movies where rating>=9;`

```
                                               QUERY PLAN
---------------------------------------------------
 Seq Scan on movies
  (cost=0.00..26.68 rows=125 width=422)
  (actual time=0.028..0.483 rows=4 loops=1)
```

Question 1 continues. . .

```
     Filter: (rating >= 9::double precision)
     Rows Removed by Filter: 2676
 Total runtime: 0.502 ms
(4 rows)
```

(b) 1. 26.68

2. 0.483ms

(c) `CREATE INDEX rating_idx on movies(rating);`

(d)
```
                                                       QUERY PLAN
---------------------------------------------------
 Index Scan using rating_idx on movies
  (cost=0.00..8.27 rows=1 width=20)
  (actual time=0.016..0.019 rows=4 loops=1)
   Index Cond: (rating >= 9::double precision)
 Total runtime: 0.036 ms
(3 rows)
```

(e) 2

(f) 2

(g) 1. No.

2. 3 (the estimated total cost is still 8.27)

Question 2: Analysis of a slightly more complex query[20 points]

Submit on separate page

Consider the following query:

```
SELECT title, year
FROM movies
WHERE rating>=9 AND num_ratings >=100;
```

You might notice that now we have two attributes on the where clause. In this question we will, again, see how indexes can improve performance.

Before running any queries, make sure that you have dropped *all indexes* that you might have created!

(a) [**3 points**] Provide the execution plan of the query. Give SQL statement and the execution plan.

(b) Based on the execution plan, answer the following questions:

   i. [**1 point**] What is the estimated total cost of the query? (in arbitrary units)

   ii. [**1 point**] What is the total actual time it took to run? (in ms)

(c) Build an index on `rating` on the `movies` table.

   i. [**2 points**] Provide the SQL statement.

   ii. [**3 points**] *Based on the new execution plan*: Has the estimated total cost 1) increased, 2) decreased, or 3) stayed the same? (answer 1,2, or 3)

(d) Delete the index on `rating` and build and index on `num_ratings` on the `movies` table.

   i. [**2 points**] Provide the SQL statements.

   ii. [**3 points**] What is the new estimated total cost? (in arbitrary units)

(e) Create indexes on both `rating` and `num_ratings`. (no need to provide the statement)

   i. [**2 points**] *Based on the new execution plan*: Compared to the case where you only have an index on `rating`, has the total cost 1) increased, 2) decreased, or 3) stayed the same? (an: 1,2, or 3)

   ii. [**2 points**] *Based on the new execution plan*: Compared to the case where you only have an index on `num_ratings`, has the total cost 1) increased, 2) decreased, or 3) stayed the same? (an: 1,2, or 3)

   iii. [**1 point**] *Based on the new execution plan*: Compared to the case where you have no indexes, has the total cost 1) increased, 2) decreased, or 3) stayed the same? (an: 1,2, or 3)

Solution:

(a)
```
EXPLAIN ANALYZE SELECT title, year FROM movies
WHERE rating>=9 AND num_ratings >=100;
```

Question 2 continues…

```
                                                  QUERY PLAN
---------------------------------------
 Seq Scan on movies
  (cost=0.00..62.20 rows=1 width=20)
  (actual time=0.011..0.362 rows=4 loops=1)
   Filter: ((rating >= 9::double precision) AND (num_ratings >= 100))
   Rows Removed by Filter: 2676
 Total runtime: 0.377 ms
(4 rows)
```

(b) 1. 62.20
2. 0.362 ms

(c) 1. `CREATE INDEX rating_idx on movies(rating);`
2. 2

(d) 1. `DROP INDEX rating_idx;`
`CREATE INDEX num_ratings_idx on movies(num_ratings);`
2. 62.20

(e) The new cost is 8.27.

1. 3
2. 2
3. 2

Homework 5 continues...

Question 3: Analyze a simple JOIN query . . . . [30 points]

Submit on separate page

Consider the following query:

```
SELECT name, title
FROM movies m , play_in p
WHERE m.mid = p.mid;
```

In this question, we will see the difference in performance when using an efficient join algorithm.

Before running any queries, make sure that you have dropped *all indexes* that you might have created!

(a) Disable hash and merge-sort join algorithms

  i. [**4 points**] Provide the statements that you used.

  ii. [**3 points**] **According to the documentation**, what is the join algorithm that the query planner is going to use?

(b) [**3 points**] Provide the execution plan of the query. Give SQL statement and the execution plan.

(c) Based on the execution plan, answer the following questions:

  i. [**3 points**] What is the estimated total cost of the query? (in arbitrary units)

  ii. [**2 points**] What is the total actual time it took to run? (in ms)

(d) [**3 points**] According to the execution plan, which join algorithm is being used?

(e) [**5 points**] Force the query planner to use *hash join*. Provide the SQL statement.

(f) [**2 points**] Provide the new execution plan after changing the join algorithm. Give SQL statement and the execution plan.

(g) [**5 points**] Based on the execution plan *after changing the join algorithm*, has the estimated total cost 1) increased, 2) decreased, or 3) stayed the same? (answer 1,2, or 3)

Solution:

(a) 1.
```
set enable_hashjoin = false;
set enable_mergejoin = false;
```
  2. The default algorithm used is the nested loop join.

(b)
```
EXPLAIN ANALYZE SELECT name, title
FROM movies m , play_in p
WHERE m.mid = p.mid;
```

Question 3 continues. . .

```
                                    QUERY PLAN

-------------------------------------------------------
 Nested Loop  (cost=0.00..6379.73 rows=74772 width=30)
 (actual time=0.065..73.428 rows=74772 loops=1)
   ->  Seq Scan on movies m  (cost=0.00..48.80 rows=2680 width=20)
    (actual time=0.006..0.396 rows=2680 loops=1)
   ->  Index Only Scan using play_in_pkey on play_in p
    (cost=0.00..2.08 rows=28 width=18) (actual time=0.006..0.020 rows
=28 loops=2680)
         Index Cond: (mid = m.mid)
         Heap Fetches: 74772
 Total runtime: 76.783 ms
(6 rows)
```

(c) 1. 6379.73
    2. 73.428

(d) Nested Loop.

(e) `set enable_hashjoin = true;`

(f)
```
EXPLAIN ANALYZE SELECT name, title
FROM movies m , play_in p
WHERE m.mid = p.mid;

               QUERY PLAN
---------------------------------
 Hash Join  (cost=82.30..2536.07 rows=74772 width=30)
  (actual time=0.968..72.362 rows=74772 loops=1)
   Hash Cond: (p.mid = m.mid)
   ->  Seq Scan on play_in p  (cost=0.00..1238.72 rows=74772 width=18)
    (actual time=0.007..22.769 rows=74772 loops=1)
   ->  Hash  (cost=48.80..48.80 rows=2680 width=20)
     (actual time=0.937..0.937 rows=2680 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 142kB
         ->  Seq Scan on movies m  (cost=0.00..48.80 rows=2680 width=20)
          (actual time=0.005..0.361 rows=2680 loops=1)
 Total runtime: 81.689 ms
(7 rows)
```

(g) 2 (The new cost is 2536.07).

Grading info: -1 for not including SQL statements

Question 4: Analyze a slightly more complicated join query[30 points]

Submit on separate page

Consider the following query:

```
SELECT name, title
FROM movies m , play_in p
WHERE m.mid = p.mid
ORDER BY name;
```

The above join query has also an order by statement added to it. In this question we are going to

1. Investigate how sorting is executed by the query planner.
2. Investigate whether indexes can help the nested loop join algorithm for this particular join query (notice that the join query is on the primary key of the `movies` table).

Before running any queries, make sure that you have dropped *all indexes* that you might have created! Also, make sure that you disable hash and merge sort joins, as you did in the previous question

(a) [**3 points**] Provide the execution plan of the query. Give SQL statement and the execution plan.

(b) Based on the execution plan, answer the following questions:

 i. [**1 point**] What is the estimated total cost of the query? (in arbitrary units)

 ii. [**1 point**] What is the total actual time it took to run? (in ms)

(c) [**5 points**] Where is the sorting done? (Disk or memory)

(d) [**5 points**] Create an index on the joining attribute `mid` on the `movies` table. Provide the SQL statement.

(e) [**5 points**] Based on the execution plan after adding the index, compared to the case where you have no indexes, has the estimated total cost 1) increased, 2) decreased, or 3) stayed the same? (answer 1,2, or 3)

(f) [**5 points**] Add index on the joining attribute `mid` on the `play_in` table (and keep the previous index that you have already added). Provide the SQL statement

(g) [**5 points**] Based on the execution plan after adding the second index, compared to the case where you have index on the `movies` table, has the estimated total cost 1) increased, 2) decreased, or 3) stayed the same? (answer 1,2, or 3)

Solution:

Question 4 continues…

(a)
```
EXPLAIN ANALYZE SELECT name, title
FROM movies m , play_in p
WHERE m.mid = p.mid
ORDER BY name;


             QUERY PLAN

------------------------------------------------
 Sort  (cost=14224.61..14411.54 rows=74772 width=30)
  (actual time=718.415..1002.984 rows=74772 loops=1)
   Sort Key: p.name
   Sort Method: external merge  Disk: 2960kB
   ->  Nested Loop  (cost=0.00..6379.73 rows=74772 width=30)
    (actual time=0.014..78.539 rows=74772 loops=1)
         ->  Seq Scan on movies m  (cost=0.00..48.80 rows=2680 width=20)
          (actual time=0.005..0.507 rows=2680 loops=1)
         ->  Index Only Scan using play_in_pkey on play_in p
         (cost=0.00..2.08 rows=28 width=18) (actual time=0.006..0.02
0 rows=28 loops=2680)
               Index Cond: (mid = m.mid)
               Heap Fetches: 74772
 Total runtime: 1063.125 ms
(9 rows)
```

(b)
1. 14411.54
2. 1002.984ms

(c) Disk (`Sort Method:  external merge`).

(d) `CREATE INDEX mid_index_m on movies(mid);`

(e) 3.

This happens because the nested loop algorithm is already taking advantage on the existing index on `mid` on table `movies` (which is created by default, since `mid` is the primary key). Thus, creating this index is redundant!

(f) `CREATE INDEX mid_index_p on play_in(mid);`

(g) 3.

As we saw before, the nested loop algorithm is already using one index to do the join. Therefore, creating that second index does not make a difference.

Grading info: Due to non deterministic behavior of Postgres, full credit was given for answers that are different from the solution. -1 for not having SQL statement