Branch: master

Find file Copy path

fa19-hw1 / README.md

es1024 Release (hw1) - 09-02-2019

3445fb8 3 days ago

1 contributor

Raw Blame History

298 lines (190 sloc) 17.7 KB

Homework 1: SQL queries and Scalable Algorithms

This homework is due: **Friday, 9/13/2019, 11:59 PM**.

Overview

In this homework, we will exercise your newly acquired SQL skills. You will be writing queries against Postgres using public data.

Prerequisites

You should watch both the SQL I and SQL II lectures before working on this homework.

Fetching the Skeleton Code

This homework assumes you have gone through and completed HW0. See the HW0 README if you have not completed HW0.

First, open a terminal and start the CS186 docker container:

```
docker start -ai cs186
```

While inside the container, navigate to the shared directory:

```
cd /cs186
```

Clone this repo. **Make sure you do this inside the container, *especially* if you are on Windows.**

```
git clone https://github.com/berkeley-cs186/fa19-hw1.git
```

If you get an error like `Could not resolve host: github.com`, try restarting your docker machine (exit the container and run `docker-machine restart`) or restarting your computer.

Now, navigate into the newly created directory:

```
cd fa19-hw1
```

Now, you should be ready to start this homework. Note that any changes you make inside the new `/cs186/fa19-hw1` directory will be saved in your machine's filesystem, but they will not be backed up in any way. You are responsible to ensure the safety of your files by backing them up somehow, as discussed in HW0.

Starting Postgres

Your image includes an installation of postgresql with the lahman database pre-loaded. At this point, all you need to do is start the postgres server. The following command will do the trick:

```
ubuntu@3c0823881763:/cs186$ sudo service postgresql start
```

(you will need to run this command each time you start up this docker container). In a minute or so the postgres server will be up. To see if it is up and working, try to run the postgres command-line interface `psql` :

```
ubuntu@3c0823881763:/$ psql
```

If you get a response like this:

```
psql: FATAL:  the database system is starting up
```

then just wait a few seconds and try again. Depending on the speed of your machine it may take a few seconds to a minute to get postgresql up and running.

Once everything is working, you will get a prompt like this:

```
psql (9.5.14)
Type "help" for help

ubuntu=#
```

At the prompt, type `\q` or `<ctrl>-d` to exit the `psql` prompt, and return back to the bash shell inside your docker container.

Creating databases and using `psql`

Postgres enables you to have multiple distinct databases supported by the same DBMS server. Each one has a different name. To create your own database, you use the shell command `createdb <mydbname>` . To connect to a particular database, give its name as an argument to the `psql` command:

```
ubuntu@3c0823881763:/$ createdb test
ubuntu@3c0823881763:/$ psql test
```

The `psql` interface to postgres has a number of built-in commands, all of which begin with a backslash. You can use the `\?` to get a list of options.

For now, use the `\d` command to see a description of your current relations. Use SQL's `CREATE TABLE` to create new relations. You can also enter `INSERT`, `UPDATE`, `DELETE`, and `SELECT` commands at the `psql` prompt. Remember that each command must be terminated with a semicolon (`;`).

Type `\help` at the psql prompt to get more help options on SQL statements.

When you're done, use `\q` or `ctrl-d` to exit `psql`.

If you messed up creating your database, you can issue the `dropdb` command to delete it.

```
ubuntu@3c0823881763:/$ createdb tst  # oops!
ubuntu@3c0823881763:/$ dropdb tst   # drops the db named 'tst'
```

Getting started

Follow the steps above to test that Postgres is set up properly, and you are able to create and drop databases.

At this point you can connect to the baseball database that is pre-loaded for you in the docker image:

```
ubuntu@3c0823881763:/$ psql baseball
baseball=# \d
```

Try running a few sample commands in the `psql` console and see what they do:

```
baseball=# \d people
```

```
baseball=# SELECT playerid, namefirst, namelast FROM people;
```

```
baseball=# SELECT COUNT(*) FROM fielding;
```

For queries with many results, you can use arrow keys to scroll through the results, or the spacebar to page through the results (much like the UNIX `less` command). Press `q` to stop viewing the results.

Notes on using postgres in this container

Your databases are being created inside the docker container, so **be aware that any database you create in a container, or any changes you make to the `baseball` database, will be reverted when you terminate the container.**

This is an unusual way to set up a docker container for a database, but good for our read-only uses in this homework.

One aspect of this approach is that any SQL `CREATE VIEW` statements you may make for convenience will be lost if you terminate the contianer (which you may need to do if something goes wrong: see Resetting the Docker container). So be sure you copy the SQL for any view definitions you create into a file under `/cs186` that you can reload next time. For example, you might save some `CREATE VIEW` commands in a file like `/cs186/trythis.sql`. Then you can always reload those commands into `psql` like this:

```
ubuntu@3c0823881763:/$ psql baseball < /cs186/trythis.sql
```

Understanding the Schema

In this homework we will be working with the commonly-used Lahman baseball statistics database. (Our friends at the San Francisco Giants tell us they use it!) The database contains pitching, hitting, and fielding statistics for Major League Baseball from 1871 through 2017. It includes data from the two current leagues (American and National), four other "major" leagues (American Association, Union Association, Players League, and Federal League), and the National Association of 1871-1875.

The database is comprised of the following main tables:

```
People - Player names, date of birth (DOB), and biographical info
Batting - batting statistics
Pitching - pitching statistics
Fielding - fielding statistics
```

It is supplemented by these tables:

```
AllStarFull - All-Star appearance
HallofFame - Hall of Fame voting data
Managers - managerial statistics
Teams - yearly stats and standings
BattingPost - post-season batting statistics
```

```
PitchingPost - post-season pitching statistics
TeamFranchises - franchise information
FieldingOF - outfield position data
FieldingPost- post-season fielding data
ManagersHalf - split season data for managers
TeamsHalf - split season data for teams
Salaries - player salary data
SeriesPost - post-season series information
AwardsManagers - awards won by managers
AwardsPlayers - awards won by players
AwardsShareManagers - award voting for manager awards
AwardsSharePlayers - award voting for player awards
Appearances - details on the positions a player appeared at
Schools - list of colleges that players attended
CollegePlaying - list of players and the colleges they attended
```

For more detailed information, see the docs online.

Writing Queries

We've provided a skeleton solution file, `hw1.sql`, to help you get started. In the file, you'll find a `CREATE VIEW` statement for each part of the first 4 questions below, specifying a particular view name (like `q2i`) and list of column names (like `playerid`, `lastname`). The view name and column names constitute the interface against which we will grade this assignment. In other words, *don't change or remove these names*. Your job is to fill out the view definitions in a way that populates the views with the right tuples.

For example, consider Question 0: "What is the highest `era` (earned run average) recorded in baseball history?".

In the `hw1.sql` file we provide:

```
CREATE VIEW q0(era) AS
    SELECT 1 -- replace this line
;
```

You would edit this with your answer, keeping the schema the same:

```
-- solution you provide
CREATE VIEW q0(era) AS
 SELECT MAX(era)
 FROM pitching
;
```

To complete the homework, create a view for `q0` as above (via copy-paste), and for all of the following queries, which you will need to write yourself.

You may need to reference SQL documentation for concepts not covered in class: reference

1. **Basics**

    i. In the `people` table, find the `namefirst` , `namelast` and `birthyear` for all players with weight greater than 300 pounds.

    ii. Find the `namefirst` , `namelast` and `birthyear` of all players whose `namefirst` field contains a space.

    iii. From the `people` table, group together players with the same `birthyear` , and report the `birthyear` , average `height` , and number of players for each `birthyear` . Order the results by `birthyear` in *ascending* order.

    Note: some birthyears have no players; your answer can simply skip those years. In some other years, you may find that all the players have a `NULL` height value in the dataset (i.e. `height IS NULL`); your query should return `NULL` for the height in those years.

    iv. Following the results of Part iii, now only include groups with an average height > `70` . Again order the results by `birthyear` in *ascending* order.

2. **Hall of Fame Schools**

    i. Find the `namefirst` , `namelast` , `playerid` and `yearid` of all people who were successfully inducted into the Hall of Fame in *descending* order of `yearid` .

    Note: a player with id `drewj.01` is listed as having failed to be inducted into the Hall of Fame, but does not show up in the `people` table. Your query may assume that all players inducted into the Hall of Fame appear in the `people` table.

    ii. Find the people who were successfully inducted into the Hall of Fame and played in college at a school located in the state of California. For each person, return their `namefirst` , `namelast` , `playerid` , `schoolid` , and `yearid` in *descending* order of `yearid` . Break ties on `yearid` by `schoolid, playerid` (ascending). (For this question, `yearid` refers to the year of induction into the Hall of Fame).

Note: a player may appear in the results multiple times (once per year in a college in California).

iii. Find the `playerid`, `namefirst`, `namelast` and `schoolid` of all people who were successfully inducted into the Hall of Fame -- whether or not they played in college. Return people in *descending* order of `playerid`. Break ties on `playerid` by `schoolid` (ascending). (Note: `schoolid` will be `NULL` if they did not play in college.)

3. SaberMetrics

   i. Find the `playerid`, `namefirst`, `namelast`, `yearid` and single-year `slg` (Slugging Percentage) of the players with the 10 best annual Slugging Percentage recorded over all time. For statistical significance, only include players with more than 50 at-bats in the season. Order the results by `slg` descending, and break ties by `yearid, playerid` (ascending).

   Baseball note: Slugging Percentage is not provided in the database; it is computed according to a simple formula you can calculate from the data in the database.

   SQL note: You should compute `slg` properly as a floating point number---you'll need to figure out how to convince SQL to do this!

   ii. Following the results from Part i, find the `playerid`, `namefirst`, `namelast` and `lslg` (Lifetime Slugging Percentage) for the players with the top 10 Lifetime Slugging Percentage. Note that the database only gives batting information broken down by year; you will need to convert to total information across all time (from the earliest date recorded up to the last date recorded) to compute `lslg`.

   Order the results by `lslg` descending, and break ties by `playerid` (ascending order).

   NOTE: Make sure that you only include players with more than 50 at-bats across their lifetime.

iii. Find the `namefirst`, `namelast` and Lifetime Slugging Percentage (`lslg`) of batters whose lifetime slugging percentage is higher than that of San Francisco favorite Willie Mays. You may include Willie Mays' playerid in your query (`mayswi01`), but you *may not* include his slugging percentage -- you should calculate that as part of the query. (Test your query by replacing `mayswi01` with the playerid of another player -- it should work for that player as well! We may do the same in the autograder.)

    NOTE: Make sure that you still only include players with more than 50 at-bats across their lifetime.

Just for fun: For those of you who are baseball buffs, variants of the above queries can be used to find other more detailed SaberMetrics, like Runs Created or Value Over Replacement Player. Wikipedia has a nice page on baseball statistics; most of these can be computed fairly directly in SQL.

Also just for fun: SF Giants VP of Baseball Operations, Yeshayah Goldfarb, suggested the following:

> Using the Lahman database as your guide, make an argument for when MLBs "Steriod Era" started and ended. There are a number of different ways to explore this question using the data.

(Please do not include your "just for fun" answers in your solution file! They will break the autograder.)

4. **Salaries**

    i. Find the `yearid`, min, max, average and standard deviation of all player salaries for each year recorded, ordered by `yearid` in *ascending* order.

    ii. For salaries in 2016, compute a histogram. Divide the salary range into 10 equal bins from min to max, with `binid`s 0 through 9, and count the salaries in each bin. Return the `binid`, `low` and `high` values for each bin, as well as the number of salaries in each bin, with results sorted from smallest bin to largest.

    Note: `binid` 0 corresponds to the lowest salaries, and `binid` 9 corresponds to the highest. The ranges are left-inclusive (i.e. `[low, high)`) -- so the `high` value is excluded. For example, if bin 2 has a `high` value of 100000, salaries of 100000 belong in bin 3, and bin 3 should have a `low` value of 100000.

Note: The `high` value for bin 9 may be inclusive).

iii. Now let's compute the Year-over-Year change in min, max and average player salary. For each year with recorded salaries after the first, return the `yearid`, `mindiff`, `maxdiff`, and `avgdiff` with respect to the previous year. Order the output by `yearid` in *ascending* order. (You should omit the very first year of recorded salaries from the result.)

iv. In 2001, the max salary went up by over $6 million. Write a query to find the players that had the max salary in 2000 and 2001. Return the `playerid`, `namefirst`, `namelast`, `salary` and `yearid` for those two years. If multiple players tied for the max salary in a year, return all of them.

Note on notation: you are computing a relational variant of the argmax for each of those two years.

v. Each team has at least 1 All Star and may have multiple. For each team in the year 2016, give the `teamid` and `diffAvg` (the difference between the team's highest paid all-star's salary and the team's lowest paid all-star's salary). Order your final solution by `teamid`. NOTE: Due to some discrepancies in the database, please draw your team names from the All-Star table (so use allstarfull.teamid in the SELECT statement for this).

Submitting the Assignment

See the main readme for submission instructions. The homework number for this homework is hw1.

Congratulations! You finished your first homework!

Testing

You can run your answers through postgres directly using:

```
ubuntu@3c0823881763:/$ psql baseball < hw1.sql
```

This can help you catch any syntax errors in your SQL.

To help debug your logic, we've provided output from each of the views you need to define in questions 1-4 for the data set you've been given. Your views should match ours, but note that your SQL queries should work on ANY data set. **We will test your queries on a (set of) different database(s), so it is *NOT* sufficient to simply return these results in all cases!**

To run the test, from within the `hw1` directory:

```
ubuntu@3c0823881763:/$ ./test.sh
```

Become familiar with the UNIX diff command, if you're not already, because our tests saves the `diff` for any query executions that don't match in `diffs/`. If you care to look at the query outputs directly, ours are located in the `expected_output` directory. Your view output should be located in your solution's `your_output` directory once you run the tests.

Note: For queries where we don't specify the order, it doesn't matter how you sort your results; we will reorder before comparing. Note, however, that our test query output is sorted for these cases, so if you're trying to compare yours and ours manually line-by-line, make sure you use the proper ORDER BY clause (you can determine this by looking in `test.sh`).

Grading

- A bit over 50% of your grade will be made up of tests released to you (the tests that are run when you run `./test.sh`).
- A bit under 50% of your grade will be made up of hidden, unreleased tests that we will run on your submission after the deadline.