CS121 Midterm Fall 2014

CS121 Midterm Instructions

This midterm has a time limit of <u>6 hours</u>, in multiple sittings. If you go beyond the time limit, just indicate what work you did after time ran out. You will get 50% credit for approximately the first hour of work over the time limit, and no credit thereafter.

> NOTE: Start your timer when you get to the actual problems. The description is a bit long, so feel free to read it carefully without being rushed, and then you can start your timer when you get to the actual problems.

Feel free to consult any course material, including the MySQL online manual, the Database System Concepts book, any lecture slides, your graded homework assignments, or the solution sets for the assignments. Do not talk to anyone else about the midterm contents until both of you have actually completed it and turned it in. Do not look for answers to the problems on the Internet.

You may use your MySQL database for this midterm, if you would find it helpful. In general, minor issues with SQL syntax will not be penalized (e.g. missing a single-quote somewhere, a misspelled keyword, forgetting a parenthesis), but larger or more fundamental syntactic or structural issues will be penalized (e.g. putting the **`WHERE`** clause before the **`DELETE`** instead of after it, or confusing **`WHERE`** with **`HAVING`**, or just making up your own syntax).

If you run into any ambiguities in the midterm, identify the issue, state your assumptions, and then give your answer. I try to make things as abundantly clear as possible, but it doesn't always happen. If you identify a real problem then we will take this into account while grading.

Relational Algebra Guidelines

Feel free to hand-write your answers to the relational algebra problems. Students who type their answers generally spend much more time on formatting things than they intended to. One suggestion is to hand-write your answers, and then if you have time (and also motivation), you can go back and type them up at the end.

Of course, however you decide to write up your answers, make sure your work is neat and readable. Sloppy or unreadable work will lose points. Submit a hard copy of hand-written work; don't scan it.

For multiple-step operations that modify data, make sure to respect all referential integrity constraints. You will lose points if you are not careful to do this.

SQL Guidelines

For problems involving SQL, please restrain yourself to only valid MySQL syntax, with the following exception:

> **You may not use vendor-specific extensions to the `UPDATE` statement.** Use the version of **`UPDATE`** covered in the book and in class. (For example, do not use MySQL's "multiple table" version of the **`UPDATE`** statement.) These extensions vary from system to system, so it is important to be able to follow the standard in this area.

If your SQL is sloppy then you will lose points. Make sure to clearly and cleanly format your SQL commands. Do not put more than 80 chars on a line; wrap long statements at appropriate points.

For SQL that creates or modifies schema or data, if you have multiple steps, make sure to respect all referential integrity constraints. You will lose points if you are not careful to do this.

CS121 Midterm Fall 2014

Part A: Game Score Database

The Northeast Retirement Home has quite a lively group of old folks living there; they are all absolutely avid gamers. They love to play a variety of games with each other – Scrabble, Canasta, Bingo (they're still old people, after all…) – and they are highly competitive. They are always boasting about who has the best scores, and who has won the latest showdown in the game room.

The only problem is, they're *old*, so they can't always remember who has the highest scores, and who won at what game! So, the caretakers decided it's time to put together a database to keep track of everything, to end the invariable disputes about who won the last game of Dominoes or Backgammon.

Here is the schema for the gaming database. Note that all xxxx_id columns are integer keys, generated by the database itself.

geezer(<u>*person_id*</u>, *person_name*, *gender*, *birth_date*, *prescriptions*)

- This relation keeps track of people at the retirement home, and is not just used for keeping track of gaming, but also for more general purposes.
- *person_id* is an auto-incrementing integer value assigned by the database
- *gender* is a single character, either "M" for male or "F" for female
- *birth_date* is a date value, with no time component
- *prescriptions* is a string description of any prescriptions the person has
- The only attribute that should allow *null* values in this table is *prescriptions*. All other attributes are non-*null*.

game_type(<u>*type_id*</u>, *type_name*, *game_desc*, *min_players*, *max_players*)

- This relation keeps track of basic information about each game that the denizens of the retirement home like to play.
- *type_id* is an auto-incrementing integer value assigned by the database
- *type_name* is a short string identifying the game, e.g. "bingo". *type_name* is also a candidate key for this relation.
- The only attribute that may be *null* is *max_players*, which indicates that a game doesn't have a maximum number of players (e.g. Bingo has no maximum). For games that require an exact number of players, such as Backgammon, *min_players* and *max_players* will be set to the same value. Obviously, *max_players* must always be at least *min_players*, and *min_players* will always be at least 1.

game(<u>*game_id*</u>, *type_id*, *game_date*)

- This relation tracks specific games that are played. Each game is assigned an auto-generated integer ID value.
- Each tuple also includes the game's type, and when the game occurred. This type ID references the primary key of the *game_type* relation.
- The game date includes both a date and time value.
- No attributes in this relation are allowed to be *null*.

game_score(<u>*game_id*</u>, <u>*person_id*</u>, *score*)

- This relation records the final score that each person achieved in a particular game.
- All score values are integers.
- This relation references both the *geezer* and the *game* relations

Relational Algebra Operations *(34 points)*

Scoring: Questions 1-2 are worth 4 points each. Questions 3-6 are worth 5 points each. Question 7 is worth 6 points.

1. Retrieve the IDs and names of all people who have played every type of game.

2. Write a query that retrieves the type ID and type name of each game, along with the number of residents who have played that game. (If a resident has played the game multiple times, they are only counted once.) Make sure to include games that nobody has played; their counts should be 0.

3. Pinochle has fallen out of favor with the old folks; they have sworn to never speak of it again. Remove all information relating to this game (*type_name* = "pinochle") from the database.

4. One (and exactly one) of the people in the retirement home is named "Ted Codd", and he knows a few things about relational databases, so he decides to embellish his scores. What relational algebra operation would he write to increase all of his game scores by 30 points?

5. A game of Dominoes has been played between Skeeter Williams and Marcus Smith, and the database needs to be updated with the results. You must also generate the unique ID to store in the game relation, and then use this in the *game_score* relation. You can generate a new ID by finding current maximum game ID, and then adding one to it.

   Here are some additional details:
   - Skeeter's ID is 987, and Marcus' ID is 227.
   - Skeeter's score was 332, and Marcus' score was 457.
   - The game occurred on January 15, 2012 at 3:30pm.
   - The game of Dominoes has the *type_name* "dominoes".

6. Write a query that reports the IDs of any game that didn't have a correct number of players for the game's type. In other words, we want to find IDs of games where the number of players doesn't fall within the minimum and maximum number of players for that game's type. Note that some game types have no maximum number of players. Also, there might be specific games, with no corresponding records in *game_score*; your result should include those game IDs as well.

7. Write a query that reports, for each kind of game, the person who has played that kind of game more times than anyone else. The schema of the result should be (*person_id*, *person_name*, *type_id*, *type_name*, *num_times*), where *num_times* is the total number of times the person has played that type of game.

SQL DDL and DML Operations *(40 points)*

Problem 1 *(8 points)*

In the file **`make-game-db.sql`**, write SQL **`CREATE TABLE`** statements to set up the above database. Your answer should include all the following:

- Use the relation and attribute names given above, for your table and column names.
- Choose an appropriate SQL type for each column. Besides the guidelines given above, here are some additional guidelines:
  - As stated before, all "*xxxx_id*" columns are integer keys. Where an integer ID is a primary key referenced by other tables, have the database auto-generate it. (You do not auto-generate integer values for foreign keys; this wouldn't make any sense.)
  - Person names will be 100 characters or less. A person's prescription could be up to 1000 characters in length.
  - Game type-names can be up to 20 characters, and can vary widely in length. Game descriptions can be up to 1000 characters, and again will vary widely in length.
  - The *game_date* column is non-*null*, and should default to the current date and time.
- Specify all appropriate primary key, candidate key, and foreign key constraints.
- Specify **`NOT NULL`** constraints where appropriate.
- Include the following **`CHECK`** constraints:
  - Geezer genders should be "M" or "F"
  - A game type's *min_players* value should be at least 1.
  - A game type's *max_players* value should either be *null*, or it should be at least the *min_players* value. (Remember that the min and max can be the same, if a game has a specific number of players.)

`CREATE TABLE` commands must also respect referential integrity constraints. Referenced relations must always be created before referencing relations. Make sure your series of DDL commands takes this into account. (If you include **`DROP TABLE`** statements as well, these should also respect referential integrity, although referencing relations are dropped before referenced relations.)

NOTE: The remaining questions should be put into the file **`game-db-queries.sql`**.

Problem 2 *(4 points)*

Retrieve the IDs and names of all people who have played every type of game.

Problem 3 *(5 points)*

Create a view "*top_scores*" that reports, for each type of game, the person (or people) with the highest score achieved for that kind of game. The view should include the game type ID and type name, the person's ID and name, and the score that was achieved. Your view doesn't need to include types of games that haven't been played by anyone.

Problem 4 *(5 points)*

Write a query that reports which types of games have been more popular over the last two weeks (i.e. starting from two weeks ago, up until now). A type of game is considered "more popular" if it has been played more than the average number of times the various kinds of games have been played. The result should just include the *type_id* of the kind of game.

Problem 5 *(5 points)*

Ted Codd's earlier shenanigans have been discovered, but (luckily for him!) the home only thinks he altered his Cribbage scores since those scores are above the maximum allowed score on the game. As a penalty, the retirement home is going to throw out all records of Cribbage games that Ted Codd participated in. Write the sequence of operations for removing all Cribbage games (*type_name* = "cribbage") that "Ted Codd" participated in. You are not allowed to rely on cascading delete operations for this question.

(Hint: You might need a temporary table to correctly implement this operation. If you use one, get rid of it when you're done.)

Problem 6 *(6 points)*

Now that Codd has pissed off all the other Cribbage players, he wants to make it up to them by using his powers for good. *(At least, for their good…)* What SQL commands would he use to add the string "Extra pudding on Thursdays!" to all Cribbage players' *prescription*?

You can use the **`CONCAT()`** MySQL function to concatenate strings. There is one wrinkle – **`CONCAT(NULL, 'abc')`** produces **`NULL`**. Make sure your answer will update all Cribbage players' prescriptions, whether their original prescription was **`NULL`** or not. You could use one of MySQL's control-flow functions[1] to work around this limitation, or you could use multiple **`UPDATE`** statements, but don't add extra pudding to anybody's prescription twice…

Problem 7 *(7 points)*

The retirement home wants to generate an overall ranking of how many times the residents have won the various multiplayer games they play. For each game that a resident has won, they will receive 1 point; for each game ending in a tie, they will receive 0.5 points. (Obviously, a loss means no points.)

This query should only include multiplayer games, i.e. where *game_type.min_players* > 1.

The result should have the schema (*person_id*, *person_name*, *total_points*), and should be ordered in descending order of total points.

*(Hint: A **`CASE`** expression is probably the easiest way to convert wins/ties to 1 point/0.5 points.)*

If you don't know where to start, write out the sequence of steps you need to go through to compute the answer, and then translate that sequence of operations into SQL.

[1] http://dev.mysql.com/doc/refman/5.5/en/control-flow-functions.html

Part B: Web Logs and Visits *(26 points)*

Typically, websites log every single HTTP request into a log file, which can then be used to analyze the behavior of the website's visitors. One important metric is how often a user visits a website, and how long each visit lasts. This presents a challenge, since the log doesn't contain a record of each visit; it simply records each page or file requested from the server, along with the IP address and the timestamp of that request.

Therefore, a "visit" is usually defined as a collection of HTTP requests from the same client, where each request in the visit is within some maximum time interval of another request in the same visit. For example, a sequence of HTTP requests from a particular IP address that are all within 30 minutes of each other are considered to be part of the same "visit," but if 30 or more minutes elapses between the last request in that visit, and a new request from the same IP address, then the new request is considered to be part of a different visit. (One could imagine that the user browsed the website until they got bored, then they went and did something else, and then finally came back a little while later to look at the website again.)

It is definitely possible for a given visit to last longer than 30 minutes, if each request in that visit is within 30 minutes of some other request in the visit.

For this problem you will work with this database schema:

weblog(*ip_addr*, *logtime*, *method*, *resource*, *protocol*, *response*, *bytes_sent*)

Note that there is no primary key for this schema. The attributes have the following meanings:

- *ip_addr* is the IPv4 address or hostname of the client making the request. We assume that each distinct value in this column corresponds to one user.
- *logtime* is a timestamp of when the HTTP request was made.
- *method* is the HTTP request method (e.g. "GET", "POST")
- *resource* is the web page being requested from the server
- *protocol* is the HTTP protocol version (e.g. "HTTP/1.0")
- *response* is the HTTP status code (e.g. 200 or 404)
- *bytes_sent* is the number of bytes the server sent to the client

As far as computing visits, we really only care about the *ip_addr* and *logtime* attributes; everything else is peripheral.

The database schema is provided in **`weblog.sql`**. The files **`visits_1.sql`**, **`visits_2.sql`** and **`visits_3.sql`** give simple examples of different scenarios. The file **`visits_full.sql`** contains six hours of actual web log data from a web server; don't use this until you are confident in your answers to the below questions, since you may end up with a runaway query.

Scoring: Problem 1 is 8 points, Problem 2 is 8 points, and Problem 3 is 10 points.

*Put your answers for this section into the file **`visits.sql`**.*

1. Write a SQL query that reports how many visits appear in the *weblog* table.

   Don't worry about writing an optimal query; just write the clearest query that computes the correct result. You will worry about optimizing the query in the next problem. (The solution query takes just under 1 minute to run.)

   Include at least a brief comment describing how your query computes the number of visits.

2. You will probably notice that the SQL query for this operation runs very slowly. Explain why this is the case.

   Enumerate all possible ways to improve the performance of the query, assuming you only have the ability to rewrite the query or issue additional DDL operations. (I.e. you can't upgrade the hardware, change fundamental constants of the universe, etc.) Make sure your explanations are specific to this query; we aren't looking for a general "how to make databases faster" answer.

   Finally, improve the performance of this query so that it runs in under 1 second. (Note: you are specifically not required to make the query as fast as possible!) Make sure to include the SQL for how you achieve this in your answer. Explain your answer.

3. Even an optimized version of the above SQL query won't scale as the number of log records increases. Write a user-defined function **`num_visits()`** that computes the total number of visits in the weblog table, using a cursor to perform one pass over the table. Make sure to correctly handle the situation where the table is empty; return 0 in this case.

```
DROP FUNCTION IF EXISTS num_visits;



-- Set the "end of statement" character to ! so that semicolons in the
-- function body won't confuse MySQL.
DELIMITER !

CREATE FUNCTION num_visits() RETURNS INTEGER
BEGIN
    RETURN 0;  -- TODO:  Write your implementation!
END !

DELIMITER ;



-- Compute the number of visits in the weblog table!
SELECT num_visits();
```