COMP3311 19T3 Database Systems

Prac Exercise 04
SQL Queries, Views, and Aggregates (i)

Aims

This exercise aims to give you practice in:

- asking SQL queries on a relatively simple schema
- using SQL aggregation and grouping operators
- writing SQL view definitions
- porting SQL across multiple database management systems

This exercise will **not** explain how to do everything in fine detail. Part of the aim of the exercise is that you explore how to use the PostgreSQL and SQLite systems. The documentation for these systems contains much useful information: PostgreSQL Manual, SQLite Manual. You should become familiar with where to find useful information in the documentation ASAP; you will need to know how to use PostgreSQL for the assignments and how to use SQLite for the exam.

Background

Access logs for web servers contain a considerable amount of information that is potentially useful in tuning both the web server parameters and the applications that run on the web server. A web server access log contains one entry for each page that is fetched from that server, where a page may be an HTML document, a PHP script, an image, etc. Each log entry contains information about one page access, including:

- the IP address of the host from which the page request was made
- the precise date/time that the request was made
- the URL that was requested (path name relative to server document root)
- the status of the fetch operation (e.g. 200 = successful, 404 = not found)
- the number of bytes of data transferred to the requestor

Here is an example from the start of the March 2005 log from the Apache web server on `mahler`:

```
60.240.97.148 - - [01/Mar/2005:00:00:00 +1100] "GET /webcms/intro/view_intro.phtml?cid=845&color=%23DEB887 HTTP/1.1" 200 342
60.240.97.148 - - [01/Mar/2005:00:00:03 +1100] "GET /webcms/notice/view_notice.phtml?cid=845&color=%23DEB887&state=view HTTP/1.1" 200 3642
60.229.57.188 - - [01/Mar/2005:00:00:06 +1100] "GET /webcms/creation/index.phtml?tid=000000124004 HTTP/1.1" 200 881
60.229.57.188 - - [01/Mar/2005:00:00:06 +1100] "GET /webcms/login/invalid.phtml HTTP/1.1" 200 1401
60.229.57.188 - - [01/Mar/2005:00:00:07 +1100] "GET /webcms/login/login.phtml HTTP/1.1" 200 4883
60.229.57.188 - - [01/Mar/2005:00:00:09 +1100] "POST /webcms/login/log_in.phtml HTTP/1.1" 302 5
60.229.57.188 - - [01/Mar/2005:00:00:09 +1100] "GET /webcms/creation/index.phtml?tid=000000124013 HTTP/1.1" 200 720
60.229.57.188 - - [01/Mar/2005:00:00:09 +1100] "GET /webcms/creation/menu.phtml?tid=000000124013 HTTP/1.1" 200 1898
60.229.57.188 - - [01/Mar/2005:00:00:10 +1100] "GET /webcms/creation/welcome.phtml?tid=000000124013 HTTP/1.1" 200 5487
60.229.57.188 - - [01/Mar/2005:00:00:12 +1100] "GET /webcms/course/index.phtml?tid=000000124013&cid=860 HTTP/1.1" 200 806
```

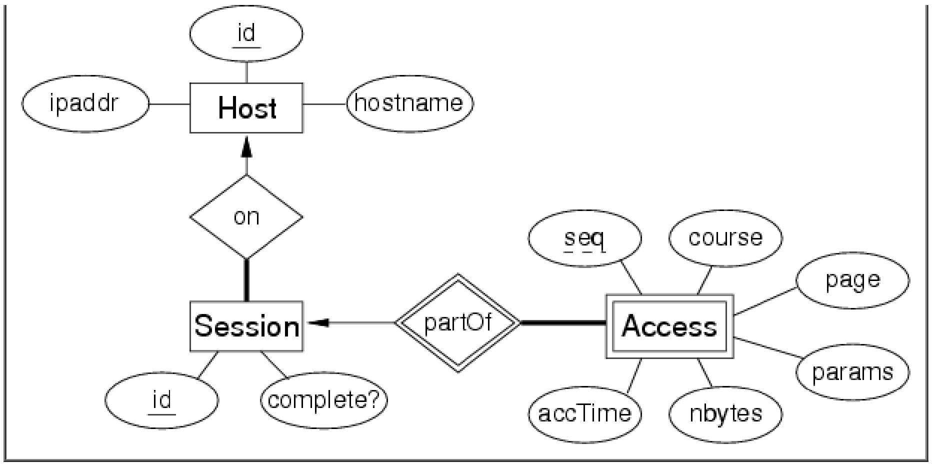
Some Web-based applications such as WebCMS introduce the notion of a "session" to a user's interaction with the web server. A user logs in to WebCMS, performs a series of page accesses (e.g. looks the the lecture notes, reads the message board, etc) and then logs out. All of these accesses are tied together by being part of a single user's interaction with the system. In an older version of WebCMS, sessions were implemented by passing a session identifier from one PHP script to the next, and checking this against a copy of the session identifier stored in the database. Thus, while the web log itself does not store information about users, it is possible to track an individual user's access to the system by finding all of the page accesses that make use of the same session identifier.

For the purposes of this exercise, imagine that we are interested in finding out the typical things that people do in a session with WebCMS. Some of this we can guess: they check the NoticeBoard, take a look whether there are any new lecture notes, read the current prac exercise, etc. Analysing the actual data in detail allows us to either confirm our hunches or discover new (unexpected) ways in which people use the system. Either way, this information could give us ideas on how to tune the performance of WebCMS.

It is very convenient to do this kind of analysis if the data is loaded into a relational database system, so the first step is to put the web log data into a relational form that captures the essential aspects of WebCMS sessions. Based on this, it is possible to define a database schema to represent the data from a WebCMS web log:

- the IP address and names of various host computers
- information about each session using WebCMS, including which host it was from and whether the user actually logged out via the WebCMS logout page (if they don't logout, their session is eventually timed-out)
- the details of each individual page access, including the name of the script, the parameters, the access time, and the session that the access was part of

We use the following ER design:



which has been converted to a relational schema.

```
create table Hosts (
        id          integer,
        ipaddr      varchar(20) unique,
        hostname    varchar(100),
        primary key (id)
);

create table Sessions (
        id          int,
        host        integer,
        complete    boolean,
        primary key (id),
        foreign key (host) references Host(id)
);

create table Accesses (
        session     int,
        seq         int,
        course      int,
        page        varchar(200),
        params      varchar(200),
        accTime     timestamp,
        nbytes      integer,
        primary key (session,seq),
        foreign key (session) references Session(id)
);
```

This schema is written in standard SQL, and so will load in any standard-conforming SQL database management system. In particular, it will load in both PostgreSQL and SQLite.

An Apache web-server on the CSE `mahler` runs the WebCMS system. The web server log for the first three days of March 2005 from this server was pre-processed to fit the schema above and is available for you load into a database.

The data has been placed into three files, each of which consists of a large number of SQL insert statements.

```
-rw-r--r--  1 YOU  YOU  5568874 31 Mar 16:58 Accesses.sql
-rw-r--r--  1 YOU  YOU   172536 31 Mar 16:09 Hosts.sql
-rw-r--r--  1 YOU  YOU   193407 31 Mar 21:03 Sessions.sql
```

Note that these data files are quite large, so you shouldn't leave them lying around under your home directory unless you have a large disk quota. However, it would be feasible to place them under your `/srvr/YOU/` directory since you have a reasonable disk quota on `grieg`, which is separate from and additional to your "home" quota.

We have also supplied a copy of the schema to build the database table structures and a template file for the exercises:

```
-rw-r--r--  1 YOU  YOU      657 31 Mar 16:10 schema.sql
-rw-r--r--  1 YOU  YOU     2243 31 Mar 16:11 weblog.sql
```

We have created a **ZIP file** containing all of the above files for you to download and use for this Prac. Since the data files are quite large (e.g. `Accesses.sql` is 5MB), you might want to put them under your `/srvr/YOU/` directory rather than under your home directory. Note that you can access `/srvr/YOU/` from an xterm on any CSE workstation without needing to log in to `grieg`.

The first thing to do is to make a directory for this Prac and extract a copy of the data files into this directory. On the CSE workstations, you can do this via:

```
% unzip /home/cs3311/web/19T3/pracs/04/weblog.zip
```

while you're in the directory where you want the files to be located.

Alternatively, if you're working on this at home, you should download the weblog.zip file your home machine and run the command:

```
% unzip weblog.zip
```

In the examples below, we assume that you have already done this.

Setting up the PostgreSQL database

Once you have copies of the schema and data files, you can setup a PostgreSQL database for this Prac via the following sequence of commands on your PostgreSQL server on `grieg`:

```
% createdb weblog
CREATE DATABASE
% psql weblog -f schema.sql
  ... should produce CREATE TABLE messages ...
% psql weblog -f Hosts.sql
  ... should produce lots of INSERT messages ...
% psql weblog -f Sessions.sql
  ... should produce lots of INSERT messages ...
% psql weblog -f Accesses.sql
  ... should produce lots of INSERT messages ...
  ... will take around 20-30 seconds to complete ...
```

Each `INSERT` statement should look like:

```
INSERT 0 1
```

The `1` means that one tuple was inserted. You can insert multiple tuples in a single SQL statement, so this number could potentially be different to `1`. The `0` means that no object ID was generated for the new tuple. PostgreSQL can generate a unique ID for each tuple if you ask.

An alternative way of achieving the above is:

```
% createdb weblog
CREATE DATABASE
% psql weblog
...
weblog=#  \i schema.sql
CREATE TABLE
CREATE TABLE
CREATE TABLE
weblog=#  \i Hosts.sql
... should produce lots of INSERT messages ...
weblog=#  \i Sessions.sql
... should produce lots of INSERT messages ...
weblog=#  \i Accesses.sql
... should produce lots of INSERT messages ...
```

If you don't want to look at at all of the `INSERT` messages, and you're using Linux or Mac OSX, then you can do the following:

```
% createdb weblog
CREATE DATABASE
% psql weblog -f schema.sql
  ... should produce CREATE TABLE messages ...
% (psql weblog -f Hosts.sql 2>&1) > .errs
  ... INSERT messages are added to file .errs ...
% (psql weblog -f Sessions.sql 2>&1) >> .errs
  ... INSERT messages are added to file .errs ...
% (psql weblog -f Accesses.sql 2>&1) >> .errs
  ... INSERT messages are added to file .errs ...
```

Note that the first loading command has only one > to create the `.errs` file, while the other loading commands use >> to append to the `.errs` file. A useful command, once you've run the above is:

```
% grep ERROR .errs
```

to check for any load-time errors. If there are any errors (and there shouldn't be), all of the tuples *except* the incorrect ones will have been loaded into the database. Using the line numbers in the error messages, you should be able to find any erroneous `INSERT` statements and correct them, and then re-run just those statements.

Once the database is loaded, access it via `psql` to check that everything was loaded ok:

```
% psql weblog
...
weblog=# \d
         List of relations
 Schema |   Name   | Type  | Owner
--------+----------+-------+-------
 public | accesses | table | YOU
 public | hosts    | table | YOU
 public | sessions | table | YOU
(3 rows)
```

```

weblog=# select count(*) from hosts;
 count
-------
  2213
(1 row)

weblog=# select count(*) from sessions;
 count
-------
  4610
(1 row)

weblog=# select count(*) from accesses;
 count
-------
 54490
(1 row)

weblog=# ...
```

Note that this is a non-trivial-sized database, and if you are not careful in how you phrase your queries, they make take quite a while to be answered. It might be useful to run your `psql` session with timing output turned on (use `psql`'s `\timing` command to do this). If a query takes too long to produce a result, see if you can phrase it differently to get the same answer, but using less time.

Another thing to note: the first time you access a table after creating it (e.g. to run the above counting queries), the query may be quite slow. On subsequent accesses to the table, it may be significantly faster. Try re-running the above queries to see if you observe this. Can you suggest why it's happening?

Setting up the SQLite Database

After copying the schema and data files, you can setup an SQLite database for this Prac via the following sequence of commands on any of the CSE workstations (or on your home machine if it has SQLite installed):

```
% sqlite3 weblog.db
SQLite version 3.8.7.1 2014-10-29 13:59:56
Enter ".help" for usage hints.
sqlite> .read /home/cs3311/web/19T3/pracs/04/schema.sql
sqlite> .read /home/cs3311/web/19T3/pracs/04/Hosts.sql
sqlite> .read /home/cs3311/web/19T3/pracs/04/Sessions.sql
sqlite> .read /home/cs3311/web/19T3/pracs/04/Accesses.sql
sqlite> .quit
%
```

There are a couple of obvious differences between PostgreSQL and SQLite on data loading. One is that SQLite doesn't give feedback for every tuple that's inserted. The other is that the two systems use different commands:

```
pgsql=# \i File
... read SQL commands from the file File ...
sqlite> .read File
... read SQL commands from the file File ...
```

Another difference is that you don't need to explicitly create the database in SQLite. In the example above, we ran the `sqlite3` command with an argument referring to a non-existent database file. After we run the `.read` statements and quit, the `weblog.db` file has been created and contains the tables and tuples from the SQL files. To remove a database: PostgreSQL has a specific `dropdb` command; for an SQLite database, you simply remove the file.

One less obvious difference is that if there are errors in the `INSERT`s ...

- PostgreSQL prints an error message for the erroneous `INSERT`, but executes all of the others
- SQLite halts execution after the first erroneous `INSERT`

Another clear difference is that SQLite takes significantly longer to do the insertions than PostgreSQL.

There is also a lurking problem with what SQLite has done that we will consider later.

Exercises

In the questions below, you are required to produce SQL queries to solve a range of data retrieval problems on this schema. For each problem, create a view called `Q`*n* which holds the "top-level" SQL statement that produces the answer (this SQL statement may make use of any views defined earlier in the Prac Exercise). In producing a solution for each problem, you may define as many auxiliary views as you like.

To simplify the process of producing these views, a template file (`weblog.sql`) is available. While you're developing your views, you might find it convenient to edit the views in one window (i.e. edit the `weblog.sql` file containing the views) and copy-and-paste the view definitions into another window running `psql`.

Note that the order of the results does not matter (except for the examples where you are imposing an order using `order by`). As long as you have the same set of tuples, your view is correct. Remember that, in theory, the output from an SQL query is a set.

Once you have completed each of the view definitions, you can test it simply by typing:

```
weblog=# select * from Qn;
```

and observing whether the result matches the expected result given below.

Queries in PostgreSQL

1. How many of the page accesses occurred on March 2?

   The results should look like:

   ```
      weblog=# select * from Q1;
       nacc
      -------
       20144
      (1 row)
   ```

2. How many times was the main WebCMS MessageBoard search facility used? You can recognise this by the fact that the page contains `messageboard` and the parameters string contains `state=search`.

   The result should look like:

   ```
   weblog=# select * from Q2;
    nsearches
   -----------
            0
   (1 row)
   ```

   (Note: if you get 1 as the count, you're probably picking up a search on one of the Web**G**MS messageboards, which is not a usage of the main WebCMS MessageBoard.)

3. On which (distinct) machines in the Tuba Lab were WebCMS sessions run that were not terminated correctly (i.e. were uncompleted)?

   The result should look like:

   ```
   weblog=# select * from Q3;
                hostname
   ----------------------------------
    tuba00.orchestra.cse.unsw.edu.au
    tuba04.orchestra.cse.unsw.edu.au
    tuba05.orchestra.cse.unsw.edu.au
    tuba06.orchestra.cse.unsw.edu.au
    tuba07.orchestra.cse.unsw.edu.au
    tuba16.orchestra.cse.unsw.edu.au
    tuba18.orchestra.cse.unsw.edu.au
    tuba20.orchestra.cse.unsw.edu.au
    tuba21.orchestra.cse.unsw.edu.au
   (9 rows)
   ```

   (Hint: the `Sessions.complete` attribute tells you whether a given session was completed)

4. What are the minimum, average and maximum number of bytes transferred in a single page access? Produce all three values in a single tuple, and make sure that they are all integers.

   The result should look like:

   ```
   weblog=# select * from Q4;
    min | avg  |  max
   -----+------+--------
      0 | 3412 | 425253
   (1 row)
   ```

5. How many of the sessions were run from CSE hosts? A CSE host is one whose host name ends in `cse.unsw.edu.au`. Ignore any machines whose hostname is not known.

   The result should look like:

   ```
   weblog=# select * from Q5;
    nhosts
   --------
      1380
   (1 row)
   ```

6. How many of the sessions were run from non-CSE hosts? A non-CSE host is one whose host name does not end in `cse.unsw.edu.au`. Ignore any machines whose hostname is not known.

   The result should look like:

   ```
   weblog=# select * from Q6;
    nhosts
   --------
   ```

```
   2785
(1 row)
```

7. How many page accesses were there in the longest session?

   The result should look like:

```
weblog=# select * from Q7;
 session | length
---------+--------
    2945 |    576
(1 row)
```

8. Each `Accesses` tuple indicates an access to a single WebCMS page/script. Produce a list of pages and their access frequency (i.e. how many times each is accessed). **Do not** use "`order by`" or "`limit`" in the view.

   The result should look like:

```
weblog=# select * from Q8 order by freq desc limit 10;
              page               | freq
---------------------------------+------
 notice/view_notice              | 9707
 course/index                    | 9288
 course/menu                     | 9133
 lecture/view_lecture            | 2969
 intro/view_intro                | 1627
 class/view_class                | 1303
 webgms/group/view_group         | 1205
 lab/view_lab                    | 1047
 messageboard/view_messagetopic  |  735
 webgms/overview/view_intro      |  692
(10 rows)
```

9. WebCMS is divided into modules, where the PHP scripts for each module is contained in a subdirectory. We can work out the module name by looking at the first component of the script name (e.g. in the sample output above, `notice`, `course`, `lecture`, etc. are modules). Produce a table of modules and their access frequency (i.e. how many times each is accessed). **Do not** incorporate "`order by`" or "`limit`" into the view.

   The result should look like:

```
weblog=# select * from Q9 order by freq desc limit 10;
    module    | freq
--------------+-------
 course       | 18602
 notice       |  9859
 webgms       |  8122
 lecture      |  3903
 messageboard |  2354
 creation     |  1884
 login        |  1776
 intro        |  1720
 class        |  1375
 lab          |  1216
(10 rows)
```

   Hint: you'll need to find out more about PostgreSQL string operators and regular expressions.
   Note: not all page URLs contain the '/' character; a page URL that looks simply like `'lecture'` should be treated as a reference to the lecture module.

10. The script that maps the web log into relational tuples isn't perfect. Has it produced any sessions that have no corresponding accesses? Write a view to print the session IDs of any such sessions.

    The result should look like:

```
weblog=# select * from Q10;
 session
---------
    3992
    3998
    4610
(3 rows)
```

11. Which hosts were not involved with any accesses to WebCMS?

    The result should look like:

```
weblog=# select * from Q11;
              unused
-----------------------------------
 tm.net.my
 203.199.51.148.static.vsnl.net.in
 mahler.cse.unsw.edu.au
(3 rows)
```

You should attempt the above exercises before looking at the PostgreSQL sample solutions.

Queries in SQLite

Now that you've attempted the above exercises in PostgreSQL, let's consider how things would work in SQLite. This provides an interesting exercise in SQL portability, since both databases support "standard SQL".

Make a copy of your `weblog.sql` file and start testing the views that you created for PostgreSQL for SQLite.

The first problem you will notice is that SQLite doesn't support

```
create or replace view ViewName as ...
```

The fix for this is simple enough. Change all of the view definitions to something like:

```
drop view if exists ViewName;
create view ViewName as ...
```

Another problem relating to view definitions is that SQLite does not support the naming of view attributes in the header of the `create view` statement. PostgreSQL allows you to define views as:

```
create view ViewName(attr1, attr2, ...)
as
select expr1, expr2, ...
```

This is just a convenient (but standard) shorthand for:

```
create view ViewName
as
select expr1 as attr1, expr2 as attr2, ...
```

and so you will need to change all of the view definitions to this form. (If you need a hint, some of the view definitions in the PostgreSQL solutions already use this form.)

Once you've made the above changes, many of the views will work correctly in both PostgreSQL and SQLite.

The next problem you will notice is that the "obvious" solution for view `Q3` doesn't seem to work. It produces *all* of the workstations rather than just the ones where sessions were not terminated correctly. The problem here is that SQLite doesn't support distinguished boolean values like `true` or `'t'` and `false` or `'f'`. It supports the syntax for a `boolean` data type, but actually uses 0 for `false` and 1 for `true`. Since these values are written into the `Sessions.sql` file, the only way to fix the problem is to modify this file and then rebuild the `weblog.db` database.

(As an aside: can you explain why the query returns all workstations on the original database, with `'t'` and `'f'` rather than `1` and `0`?)

Finally, while you are trying to write a solution for view `Q9`, you will discover that SQLite's string processing functions aren't quite as powerful as PostgreSQL's. As far as I can tell, there is no way to solve `Q9` in SQLite without re-doing the database structure and separating out the module name as a separate attribute in the table.

The goal now is to get as many queries as you can to work on both systems and to understand why some queries can't be made to work with the existing data.

Once you've attempted the exercises, compare your solutions against those in the SQLite sample solutions.