Problem Set 1: SQL

Assigned: 9/8/2014

Due: 9/15/2014 11:59 PM

Submit to the 6.830 Stellar Site (`https://stellar.mit.edu/S/course/6/fa14/6.830/homework/`)

You may work in pairs on this problem set. Clearly indicate the name of your partner. Only one of you needs to submit on Stellar.

1 Introduction

The purpose of this assignment is to provide you with hands-on experience with the SQL programming language. SQL is a declarative language in which you specify the data you want in terms of logical expressions that specify the properties the returned data should satisfy but not how to actually compute the answer.

We will be using the SQLite open source database, which provides a standards-compliant SQL implementation. In reality, there are slight variations between the SQL dialects of different vendors—especially with respect to advanced features, built-in functions, and so on. The SQL tutorial at `http://sqlzoo.net/`, provides a good introduction to the basic features of SQL; after following this tutorial you should be able to answer most of the problems in this problem set (the last few questions may be a bit tricky). You may also wish to refer to Chapter 5 of "Database Management Systems." This assignment mainly focuses on *querying* data rather than modifying it. `http://sqlzoo.net/` includes a reference section that describe how to create tables and modify records; you should read it so that you are familiar with these aspects of the language.

SQLite a very easy database to use because a database in simply stored in a file. We have put a database file on athena for you; you may download this file to your local machine and run SQLite there, or log in to athena and use it. More details are given in Section 3 below.

2 MIMIC2 Data

In this problem set, you will write a series of queries using the `SELECT` statement over an anonymized medical patient database containing information about approximately 500 patients. `http://mimic.physionet.org/UserGuide/UserGuide.pdf`).

The MIMIC2 project, led by Prof. Roger Mark of MIT, consists of detailed data on about 42,000 patient stays in the various intensive care units (ICUs) at the Beth Israel/Deaconess Medical Center in Boston. These data include tabular data reporting prescriptions, the results of laboratory tests, clinical orders, billing codes, minute-by-minute summary data from bedside monitors, narrative text that includes nursing and doctors' notes, discharge summaries, radiology and pathology reports, and high-resolution waveform data on a fraction of these patients for whom such data were offloaded from the monitors and successfully matched to their clinical records. The entire database has been de-identified, so that it is (supposed to be) impossible to link the records back to specific real individuals. One may access the entire data set by passing the COUHES training course in responsible human subjects research and signing a limited data use agreement in which you promise not to further redistribute the data or try to contact patients if you discover flaws in our de-identification algorithms. For convenience, the project has also produced a subset of data about 500 deceased patients. Under U.S. laws, dead people have vastly reduced privacy rights, so we make these records available without having to do the training or sign the agreement. Nevertheless, we ask you not to further redistribute the data; others who are interested may obtain the most up to date version directly from the MIMIC project.

The tables that will be used in this problem set include `a_meddurations`, `chartevents`, `d_chartitems`, `d_meditems`, `d_patients`, `icustayevents`, `medevents` and `labevents`. Patient information is stored in `d_patients`, referred as the patient table throughout this document. As the database went through a careful de-identication process, the patient table only stores the patient identifier (Subject ID), gender (sex), date of birth (dob, shifted) and date of death (dod, shifted) . Medication(s) given to a patient are recorded in the `medevents`, `d_meditems`, `a_meddurations` and `additives` (not used in this problem set) tables. Patient medical chart data is recorded in the `chartevents`, `d_chartitems`, `a_chartdurations` and `formevents` tables (the latter two are not used in this problem set). The ICU (intensive care unit) stay events are recorded in the `icustayevents` table, in which `intime` and `outtime` denote the check-in and check-out times from the ICU.

Example "Data Definition Language" (DDL) commands used to create `d_patients` and `icustayevents` tables is as follows:

```
CREATE TABLE d_patients (
  subject_id INTEGER NOT NULL PRIMARY KEY, -- patient id
  sex VARCHAR(1) DEFAULT NULL, -- patient gender
  dob DATETIME NOT NULL, -- patient date of birth
  dod DATETIME DEFAULT NULL, -- patient date of death
  hospital_expire_flg VARCHAR(1) DEFAULT 'N' -- whether patient died at hospital
);

CREATE TABLE icustayevents (
  icustay_id int(11) NOT NULL PRIMARY KEY, -- ICU stay event id
  subject_id int(11) NOT NULL, -- patient id
  intime datetime NOT NULL, -- check in time
  outtime datetime NOT NULL, -- check out time
  los double NOT NULL, -- length of stay
  first_careunit int(11) DEFAULT NULL, -- first care unit stayed
  last_careunit int(11) DEFAULT NULL, -- last care unit stayed
  UNIQUE KEY icustayev_u1 (subject_id,intime),
  FOREIGN KEY subject_id REFERENCES d_patients(subject_id),
  FOREIGN KEY first_careunit REFERENCES careunits(cuid),
  FOREIGN KEY last_careunit REFERENCES careunits(last_careunit),
  KEY icustayevents_o1 (intime),
  KEY icustayevents_o2 (outtime)
)
```

`CREATE TABLE name` defines a new table called `name` in SQL. Within each command is a list of field names and types (e.g., `subject_id INTEGER` indicates that the table contains a field named `subject_id` with type `INTEGER`). Each field definition may also be followed by one or more modifiers, such as:

- `PRIMARY KEY`: Indicates this is a part of the primary key (i.e., is a unique identifier for the row). Implies `NOT NULL`.
- `NOT NULL`: Indicates that this field may not have the special value `NULL`.

In addition, `FOREIGN KEY (field) REFERENCES` indicates that this field is a foreign key which references an attribute (i.e., column) in another table. Values of this field must match (join) with a value in the referenced attribute. Phrases following the "`--`" in the above table definitions are comments.

Notice that the above tables reference each other via several foreign-key relationships. Medications are administered on a patient, indicated by the `subject_id` attribute of the `a_meddurations` table, which references the `subject_id` attribute of the `d_patient` table. Each patient can receive multiple medications and each medication can be administered on multiple patients. Because this is a many-to-many relationship, we need to represent it using the intermediate table `a_meddurations`. Each row of `a_meddurations` indicates that a particular patient (`subject_id`) received a particular medication (`itemid`, which is another foreign key referring to the `itemid` in `d_meditems`).

Figure 1 is a simplified "Entity-Relationship Diagram" that shows the relationships (diamonds) between the tables (squares). This is a popular approach for conceptualizing the schema of a database; we will not study such diagrams in detail in 6.830, but you should be able to read and recognize this type of diagram. For the rest of the tables used in this problem set, we show their schema via separate figures instead of in terms of `CREATE TABLE` commands. Figure 2 shows medication events related schemas. Figure 4 shows lab events related schemas. Figure 3 shows chart events related schemas.

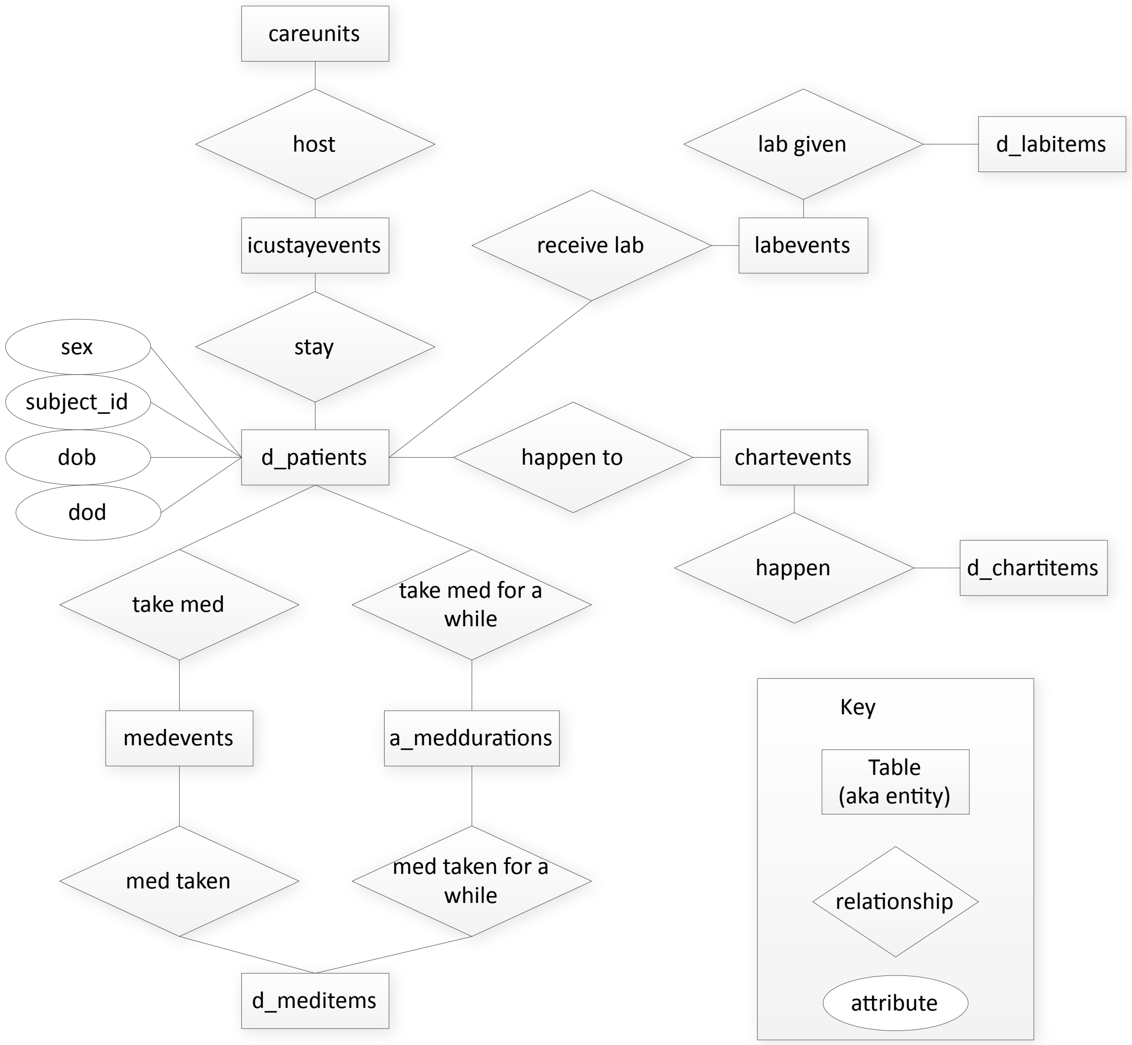

Figure 1: Simplified ER Diagram.

3 Running SQLite on the MIMIC2 Dataset

This section describes how to get SQLite database running on the MIMIC2 database. For users inexperienced in Unix-like operating systems, or without Linux or MacOS on their machine, we recommend that you use athena, either by going to one of the clusters, or by ssh'ing into one of the athena machines (e.g., athena.dialup.mit.edu).

Obtaining SQLite: You first need to obtain a copy of the sqlite3 binary, which is the program that reads (and manipulates) SQLite database.

If you are running on a *64 bit Linux* machine (such as athena.dialup.mit.edu, and most of the Athena workstations), we have put a pre-built binary at `http://db.csail.mit.edu/sqlite3.tar.gz` (this binary includes readline support, which makes using the interactive SQL shell much easier).

To download and decompress it, open a command-line and type something like

```
wget http://db.csail.mit.edu/sqlite3.tar.gz
tar -xvzf sqlite3.tar.gz
```

Alternatively, you can download a binary to your local machine by going to `http://www.sqlite.org/download.html` and downloading the "precompiled command-line shell" for your platform.

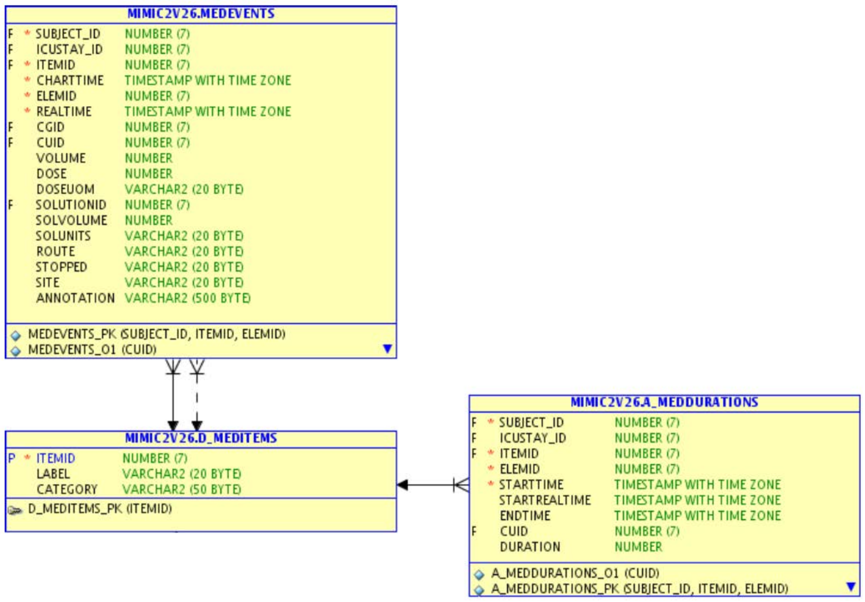

Figure 2: Schemas for medication events related tables.

Connecting to the database: Assuming you are in the same directory as the sqlite3 binary you just downloaded, and on a machine with AFS access (such as any Athena machine), you should now be able to type:

```
./sqlite3 /afs/csail.mit.edu/group/db/6830/mimic2.db
```

You should see a prompt that says:

```
sqlite>
```

Note that this database file is quite large (2.5 GB uncompressed, 500 MB compressed) and you may not have sufficient quota to copy it to your local athena directory. If you would like to download it to your local machine (to run sqlite3 locally on a non-afs equipped machine), we have placed a compressed copy at:

```
http://db.csail.mit.edu/mimic2.tgz
```

You will need to decompress this using the tar command line tool (which should be installed by default on Mac and Linux systems). From the command line you can download and decompress by typing something like:

```
wget http://db.csail.mit.edu/mimic2.tgz
tar -xvzf mimic2.tgz
```

Then to run SQLite, you would type (assuming sqlite3 is in the current directory):

```
./sqlite3 mimic2.db
```

Please contact us if you have trouble installing or running SQLite.

4 Using the Database

Once connected, you should be able to type SQL queries directly. All queries in SQLite must be terminated with a semi-colon. For example, to get a list of all records in the `d_patients` table, you would type (note that running this query will take a long time, since it will return many answers, so please don't do it – keep reading instead!):

```
SELECT * FROM d_patients;
```

A less expensive way to sample the contents of a table (which you may find useful) is to use the `LIMIT` SQL extension which specifies a limit to the number of records the database should return as answer to a query. This is not part of the SQL standard, but is supported by many relational DBMSes, including SQLite. For example, to view 20 rows from the `d_patients` table, use the following query:

```
SELECT * FROM d_patients LIMIT 20;
```

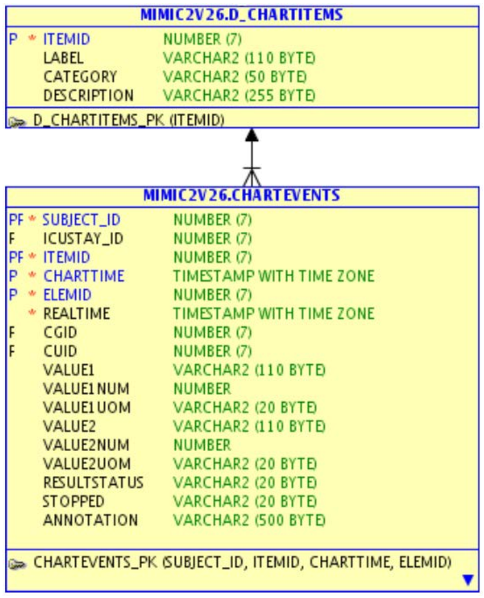
MIMIC2V26.D_CHARTITEMS
P * ITEMID NUMBER (7)
LABEL VARCHAR2 (110 BYTE)
CATEGORY VARCHAR2 (50 BYTE)
DESCRIPTION VARCHAR2 (255 BYTE)
D_CHARTITEMS_PK (ITEMID)
MIMIC2V26.CHARTEVENTS
PF * SUBJECT_ID NUMBER (7)
F ICUSTAY_ID NUMBER (7)
PF * ITEMID NUMBER (7)
P * CHARTTIME TIMESTAMP WITH TIME ZONE
P * ELEMID NUMBER (7)
* REALTIME TIMESTAMP WITH TIME ZONE
F CGID NUMBER (7)
F CUID NUMBER (7)
VALUE1 VARCHAR2 (110 BYTE)
VALUE1NUM NUMBER
VALUE1UOM VARCHAR2 (20 BYTE)
VALUE2 VARCHAR2 (110 BYTE)
VALUE2NUM NUMBER
VALUE2UOM VARCHAR2 (20 BYTE)
RESULTSTATUS VARCHAR2 (20 BYTE)
STOPPED VARCHAR2 (20 BYTE)
ANNOTATION VARCHAR2 (500 BYTE)
CHARTEVENTS_PK (SUBJECT_ID, ITEMID, CHARTTIME, ELEMID)

Figure 3: Schemas for chart events related tables.

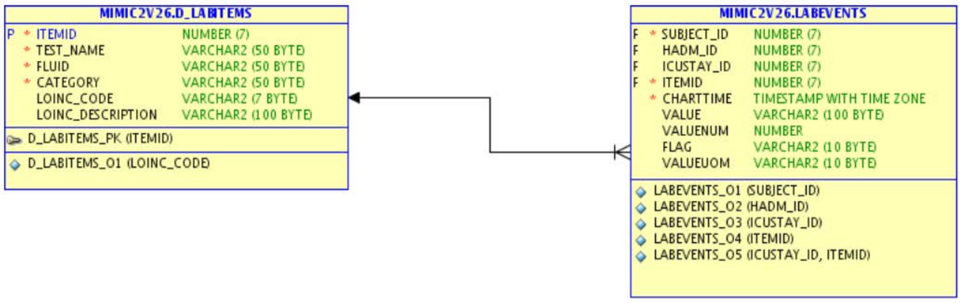
MIMIC2V26.D_LABITEMS
P * ITEMID NUMBER (7)
* TEST_NAME VARCHAR2 (50 BYTE)
* FLUID VARCHAR2 (50 BYTE)
* CATEGORY VARCHAR2 (50 BYTE)
LOINC_CODE VARCHAR2 (7 BYTE)
LOINC_DESCRIPTION VARCHAR2 (100 BYTE)
D_LABITEMS_PK (ITEMID)
D_LABITEMS_01 (LOINC_CODE)
MIMIC2V26.LABEVENTS
F * SUBJECT_ID NUMBER (7)
F HADM_ID NUMBER (7)
F ICUSTAY_ID NUMBER (7)
F * ITEMID NUMBER (7)
* CHARTTIME TIMESTAMP WITH TIME ZONE
VALUE VARCHAR2 (100 BYTE)
VALUENUM NUMBER
FLAG VARCHAR2 (10 BYTE)
VALUEUOM VARCHAR2 (10 BYTE)
LABEVENTS_01 (SUBJECT_ID)
LABEVENTS_02 (HADM_ID)
LABEVENTS_03 (ICUSTAY_ID)
LABEVENTS_04 (ITEMID)
LABEVENTS_05 (ICUSTAY_ID, ITEMID)

Figure 4: Schemas for lab events related tables.

The `LIMIT` clause above returns only the first 20 rows of the result for the query `SELECT * FROM d_patients`. However, keep in mind that the relational model does not specify an ordering for rows in a relation. Therefore, there is no guarantee on which 20 rows from the result are returned, unless the query itself includes an `ORDER BY` clause, which sorts results by any output column or an expression on columns used in the query. Nevertheless, `LIMIT` is very useful for playing around with a large database and sampling data from tables in it.

SQLite includes a number of simple commands to help you query the metadata of the database and to understand query performance. You can use the `.help` to see a list. The most useful are `.tables` to a see a list of tables and `.schema tablename` to see the schema of a given table.

Note that the output of SQLite can be a bit hard to read by default. You may find it more readable if you run the following commands from the sqlite prompt:

```
.mode column

.headers on
```

Note that in column mode, you may need to explicitly control the width of columns to prevent truncation. If you want the first two columns to each be 50 characters wide, you can write:

```
.width 50 50
```

5 Questions

For each question, please include both the **SQL query** and the **result** in your answer. Some of the more complex queries can take quite a while to run, so be patient!

Q1. Write a query (using the `SELECT` statement) that finds the maximum age of death of the patients who died in a hospital. You may find the aggregate function "`max`" useful.

Q2. The `labevents` table records the lab tests that patients are given. Its `itemid` field gives a numeric code that identifies the particular type of lab test recorded in each entry, and those codes are translated to readable text descriptions in the `d_labitems` table. Write a SQL statement that shows the 20 most frequently occurring types of blood tests and the number of occurrences of each test. Blood tests can be detected by filtering on the `fluid` column of the `d_labitems` table. You will need to write a "join query" that combines these two tables.

Subqueries and nesting: In SQL, a subquery is a query over the results of another query. You can use subqueries in the `SELECT` list, the `FROM` list, or as a part of the `WHERE` clause. For example, suppose we want to find all the info of the patient with the smallest id. To find the smallest id, we would write the query `SELECT min(subject_id) FROM d_patients`, but SQL doesn't provide a way to get the other attributes of that minimum-id patient without using a nested query. We can do this either with nesting in the `FROM` clause or in the `WHERE` clause. Using the nesting in the`FROM` clause, we would write:

```
SELECT *
FROM d_patients,
     (SELECT min(subject_id) AS minid
      FROM d_patients) AS nested
WHERE d_patients.subject_id = nested.minid;
```

Using nesting in the `WHERE` clause, we would write:

```
SELECT *
FROM d_patients
WHERE d_patients.subject_id = (SELECT min(subject_id) FROM d_patients);
```

As another example, if you were interested in finding information about medications given to a particular set of patients (e.g., all female patients) you could write the following query:

```
SELECT DISTINCT d_meditems.*
FROM d_meditems,medevents
WHERE d_meditems.itemid=medevents.itemid
AND medevents.subject_id IN (SELECT subject_id FROM d_patients WHERE sex = 'F');
```

Note that here we use `IN` instead of = because the subquery returns a set rather than a single value (= can only be used to compare two single values.)

It is usually the case that when confronted with a subquery of this form, it is possible to un-nest the query by rewriting it as a join, as in the example above.

Also note the use of the `DISTINCT` keyword, which is used to show all unique rows. To understand its utility, try running the above query with `DISTINCT` removed.

Q3. Show what the `SELECT ...WHERE ...IN` query above would look like when the `IN` portion of the query is "un-nested" by rewriting it as a join.

Q4. The `chartevents` table records measurements and observations that are part of a patient's (electronic) chart. Compute a histogram of the number of patients with a certain number of chart events, bucketed in increments of 1000. You may find the `%` operator useful to calculate the buckets. The format should be as follows:

0	number of patients with between 0 (inclusive) and 1000 (exclusive) chart events
1000	number of patients with between 1000 and 2000 chart events
...	...
n	number of patients with between n and n + 1000 chart events

You do not need to worry about printing buckets that have 0 patients.

Q5. Look at the `icustayevents` and `medevents` tables. Find the `subject_id` of patients who have stayed in the ICU but do not have any medications associated with those stays.

We will now ask a few more involved questions. To answer these, there are a few more features of SQL you should be familiar with.

Temporary Tables: In addition to allowing you to nest queries, SQL supports saving the results of a query as temporary table. This table can be queried just like a normal table, providing similar functionality to nested queries, with the ability to reuse the results of a query. The command to create a temporary table is:

```
CREATE TEMP TABLE name AS SELECT ...
```

where "..." are the typical `SELECT` arguments. This creates a table called `name`. `TEMP` causes the table to automatically be deleted (or "dropped" in SQL nomenclature) when the session is over, such as when you quit `sqlite3`.

Temporary tables can be useful when interactively developing a SQL query, since you can explore or build on previous results. It is usually possible to use nesting in place of temporary tables and vice versa; nesting will (generally) lead to better performance as query optimizers include special optimizations to "de-nest" queries that cannot be easily applied on temporary tables.

In addition to nesting (or temporary tables), to answer the next few questions, you must learn one more concept: *self-joins*.

Self joins: A self-join is a join of a table with itself. This is often useful when exploring a transitive relationship. For example, the following query:

```
SELECT le2.itemid
FROM labevents AS le1, labevents AS le2
WHERE le1.itemid = 50225
AND le1.subject_id = le2.subject_id;
```

returns the list of the ids of labs that have been performed on a patient who also received lab 50225 (potassium test).

You will need to use a combination of the above features in answering the following queries.

Q6. Look at the table `icustayevents`. Find the pairs of patients who were in the ICU at the same time. List the pairs in order of the total number of days of overlap, descending, and be sure to list each pair only once. Report the two `subject_ids` and the total number of days of overlap for patients overlapping by a total of at least one day. Also list the number of overlapping stays. You may find the SQLite function `julianday` useful.

Q7. We say the *busiest month* of a given year in the ICU is the the month with the most records in the `medevents` table that have corresponding ICU stays. Write a query that reports the busiest month of each year and the number of ICU stays that span that busiest month (i.e., that start before and end after the month). Your output should consist of 3 columns: year, month, and the number of the ICU stays, ordered in ascending order by year. You may find the SQLite function `strftime` useful.

Q8. Consider the table `a_meddurations`. For each patient who received at least 2 medications, find the longest period of time during which at least 2 medications were taken. Output 2 columns, each row containing the subject_id and the length of the time period. Limit your output to the top 20 rows, in descending order of duration.