

Assignment 3: Embedded SQL

Due Thu March 30
at the beginning of class

No late assignments will be accepted.

In this assignment, you are required to write a program in the C programming language with embedded SQL for the DB2 database management system. (Please see the links to embedded SQL on the course web page.) Your program should be tested, and you are encouraged to do the testing on the DB2 system on CDF, but you may use any DB2 system available to you.¹

Your program should be well documented (*i.e.*, well commented), easy to understand, and easy to grade. The program should begin with comments describing what the program does. 20% of your grade will be for presentation and good programming style, including good documentation in *good English*. If your program is incomplete in any way, point this out prominently in a comment at the front of each incomplete program.

Hand in your embedded SQL code and a script file of a terminal session with your program. (Type `man script` if you do not know about unix script files). The terminal session should demonstrate that your program works correctly, but it should not be too long. Thus, your tests should be carefully chosen to be succinct and convincing. Points will be deducted if you just hand in a long, massive terminal session.

In general, you should present your program and tests so that the TA can quickly and easily understand and grade them, especially since he has only a limited amount of time to spend on each assignment.

Start this assignment early. It is a lot of work.

The US Department of Homeland Security is trying to keep track of suspected terrorist networks. Unfortunately, the Department's intelligence information is incomplete. Fortunately, due to generous funding by Congress, the Department has an extensive intelligence gathering network, so its information is constantly expanding. Each time the Department discovers a new terrorist suspect, it assigns him/her an id (a 10-character alphanumeric string). In addition, The Department is very concerned about communication between terrorist suspects, and it maintains an information system just for this purpose. The system assigns a unique identification number to each suspect, and it stores information in a database with the following scheme:

```
suspect [id,name,date]
direct [id1,id2,date]
```

In the `suspect` relation, a tuple `[id,name,date]` means that `id` is the identifier for a suspect, `name` is his/her name, and `date` is the date when the information was entered into the system. In the `direct` relation, a tuple `[id1,id2,date]` means that suspect `id1`

¹However, we can only provide you with help and advice for the DB2 system on CDF. If you use another facility (especially one that does not run on Unix) and you run into trouble, then you are on your own.

can communicate directly with suspect `id2`, where `date` is the date when this information was entered into the system. Here, `id1, id2` is a key for `direct`, and all communication is assumed to be two-way.

- (5 points) Build a database with this schema and populate each relation with at least 40 tuples. You should choose data that allows you to test the program you are about to write.

Your main task is to write a program in embedded SQL that executes transactions on this database. The idea is that the user doesn't know SQL. The program must therefore provide a menu of commands, where each menu item is a type of transaction. Your program should provide the transaction types listed below, but it should also be easy to add new transaction types to your program. You should produce a reasonable interface, suitable for a dumb terminal. Don't worry about fancy graphics. The program should simply prompt the user for input, and provide neat, legible output. The input will tell the program what to do. You do not have to provide a help facility for the user. Here are the required transaction types:

- (5 points) Enter a new suspect into the database. The user supplies the name and date, and your program creates an identifier for him/her.
- (5 points) Enter a new direct communication link. The user provides the identifiers of the two communicating suspects.
- (5 points) Given a date and the identifier of a suspect, `S`, return the name and identifier of all other suspects that `S` was known to be in direct contact with as of the given date. (i.e., ignore all tuples with a later date.)
- (5 points) Print the entire database. i.e., print the `suspect` table, and print the `direct` table.

In addition to direct communication, the Department of Homeland Security is concerned about possible indirect communication between terrorist suspects, ie, communication that uses other terrorist suspects as intermediaries. Specifically, two suspects, `S1` and `Sn` can communicate indirectly if there is a sequence of other suspects, `S2, S3 . . . Sn-1` such that `Si` can communicate directly with `Si+1` for `i` from 1 to `n-1`. To this end, your embedded SQL program should also implement the following query transaction:

- (20 points) Given a date and the identifiers of two suspects, `S1` and `S2`, return `YES` if it was known that `S1` and `S2` could communicate either directly or indirectly as of the given date. (i.e., ignore all tuples with a later date.)

You should implement this transaction using a linear-time graph algorithm (such as breadth-first search, depth-first search or spanning tree). Your algorithm may make use of temporary tables in the database (as with the transitive closure example in Lecture 14). However, for full credit, you should make full use of SQL to simplify your algorithm, reduce the amount of C code, and minimize the number of temporary tables. In particular, you should use SQL joins whenever possible. (Hint: Breadth First Search can be implemented quite easily using SQL.) However, as described below, you should *not* use recursion, either in your C code or in your SQL queries.

In general, your program should prompt the user for a transaction type. Subsequent prompts will then depend on the type of transaction the user selected. For example, to enter a new suspect into the database, your program will have to prompt for different information than to enter a communication link. Your program should also do some simple error checking. For instance, to enter a communication link between two suspected terrorists, the two suspects should be in the database already. To simplify your task, you may assume that if the program expects the user to type in a value of a given type (e.g., a string), then he does indeed do this. Thus, the only error checking that needs to be done is checking for consistency with information already in the database.

Your program should also check for the proper execution of all SQL statements. If an SQL transaction does not execute properly, then your program should abort the current transaction and print an error message. If there are no errors, then your program should commit the transaction. After committing or aborting a transaction, the program should ask the user if he wants to start another transaction. Test your program on a reasonable set of data.

In the terminal session that you hand in, you should print out the database tables after every few transactions, so that the marker can check the effects of your program.

Your program should be general enough to work on *any* set of data, even if there are millions of tuples in each relation. (This is why we are using a database to store the data.) In particular, the amount of main memory used by your program should be *fixed*, *i.e.*, independent of database size. Thus, your program should *not* store a relation (or any significant portion of it) in a data structure in main memory (such as an array or a linked list). Only a fixed number of tuples may be stored in main memory at one time, regardless of how large the database is. Note that these requirements imply that your program should *not* be recursive, since each level of recursion requires more memory. In addition, your program should *not* use recursive queries, even if the DB2 system you are using supports them. (The whole point here is to write non-trivial program in embedded SQL.)

At the front of your program, include a comment that describes (in *good English*) how the program works. This comment should give a clear and succinct description of all your algorithms. Likewise, be sure to clearly identify the code for each transaction, and describe how it works. If you used any clever implementation tricks in your program, be sure to describe them, so that you can get credit for them. In general, the marker should be able to understand and appreciate what you have done without studying your source code. (She should only have to look at the source code to confirm that you have done what you say you have done.)

Have fun!