LKM #3: Databases and the Kevin Bacon Problem

Chloe Eghtebas and Samantha Kumarasena
Software Systems, Spring 2014

Our Github repository can be found here: https://github.com/NCEghtebas/BaconExperiment

In this LKM, we created a foundation for implementing our own version of the “Kevin Bacon” problem. We took data from IMDB, parsed it, loaded it into a database, and used a C API to query it. To do so, we used MySQL and its C API.

MySQL is free, and is under the GNU GPL (General Public License). This means that the software can be modified and distributed as long as these modifications also fall under GPL. Additionally, any applications can be distributed commercially, but the source code must also be provided. MySQL 5.1 is installed by default as of Ubuntu 12.04 and onward. We chose to update to the latest version anyway, version 5.5. The install was not at all complicated -- we had no difficulty updating.

The MySQL documentation can be found here: https://dev.mysql.com/doc/refman/5.1/en/index.html

The documentation is very thorough and intuitive. The tutorial is easy-to-follow and we both understood the basics after following the examples provided. Additionally, MySQL is a very popular database manager, so many of the problems we encountered could be solved by looking for answers on popular forums (eg. StackOverflow). We also found plenty of helpful examples to follow, and followed these examples when learning to make simple queries.

MySQL tends not to provide useful debugging information. Warnings and errors are generally limited to messages to the effect of “You have an error in your SQL syntax; check the manual” or “Syncing error,”. These errors are not very informative, and so it can be difficult to find the error in a complicated query. This was a notable (but nowhere near insurmountable) obstacle in learning MySQL. Other than a lack of helpful error messages, learning MySQL was simple.

The MySQL C API is also well-documented, although its documentation is not quite as thorough as that of MySQL itself. Many tutorials were available online, and plenty of examples could be found on popular software help forums. The code for this homework was built on a very thorough example we found in the documentation. This example was well-commented and included instantiation of connection objects, simple queries, and error-handling. We were able to set up a simple example fairly easily, enabling us to start experimenting with more complicated queries quickly.

Our system utilizes a database with two tables. One table maps actors to movies they have played in. The other maps movies to the actors that have starred in them. This system can handle queries in both directions efficiently (i.e. given an actor, finding the movies they have starred in is efficient; given a movie title, finding the actors that have starred in that movie is also efficient) because we have created two different tables with flipped key-value pairs.

There is a good match between these capabilities. If we were to implement a shortest-path algorithm, we would be able to do so fairly efficiently, given that we are capable of running queries in both directions in reasonable amounts of time (without searching the entire database). A shortest-path algorithm would

involve querying the movies given an actor has starred in, then searching those movies for their actors, and seeing if any of those actors are Kevin Bacon. If Kevin Bacon is not found among those actors, then the algorithm would conduct movie queries on each of *those* actors, and so on (until Kevin Bacon is found in the cast of one of those movies). The number of times needed to repeat this process is the Bacon number. Though preprocessing could make our queries more efficient, there is still a good match between the capabilities of this system and the needs of the algorithm.

We can process basic queries relatively quickly, but performance becomes poorer as the query increases in size and complexity. In order to implement a shortest-path algorithm, being able to conduct a large number of concurrent queries is important. We expect the performance of this system will not scale up to large numbers of concurrent queries.

Precomputing connections between actors and movies (and storing those results) would speed up such queries dramatically. However, we do not use precomputed results in our system, and as a result, the system will not perform as well (we would be conducting the same queries repeatedly).

Additionally, MySQL's claims of ACIDity (Atomicity, Consistency, Isolation, Durability) make a convincing argument in favor of this database management system. These claims can be found here: https://dev.mysql.com/doc/refman/5.6/en/mysql-acid.html

In terms of **atomicity**, MySQL has settings that allow autocommits, rollbacks, and commit statements like a code repository which display its capability of synchronization. In addition, MySQL provides informational schema tables which allow the user to view read-only information regarding to the database. These provided features and their atomicity are thanks to the InnoDB storage engine which is claimed to be unmatched by any other disk-based relational database engine.

In terms of **consistency** in the ACID model, the InnoDB storage engine contains features for its internal processing like the doublewrite buffer or the crash recovery which mean that there is always an extra copy of your data when transferring and safely reconstructs the data from the buffer if a crash were to happen.

The atomic transactions mentioned earlier, such as the autocommit which commits after each SQL statement, displays **isolation** in that they minimize the amount of locking. Locking is a mechanism which protects transactions from other querying or changing transactions.

Lastly, the **durability** from MySQL relays on with the computers hardware features like the buffer is written to array of storage devices, battery backed cache, uninterruptible power supply protecting power to MySQL servers. In addition, there are numerous software implementations to ensure synchronization, security and frequency of backups, and various other customizable settings.

MySQL's C API is very easy to get started with, particularly because it is very well documented. The C API can be found here: http://dev.mysql.com/doc/refman/5.0/en/c-api.html

We also found many helpful tutorials online. This tutorial gave us a particularly thorough example: http://zetcode.com/db/mysqlc/

The API was a nice blend of C and MySQL. The functions were all well-documented and easy-to-use. The query calls were simple. However, C does make dealing strings and all the characters involved in a query call difficult. The MySQL functions did not support format strings, which made it difficult to make queries involving user input. We spent some time working through these issues, and eventually decided on using C string library functions (strcpy() and strcat()) to cobble together a working query. This was not terribly

difficult, but it took us some time.

In conclusion, we found MySQL and its C API to be appropriate for this project. Both are very easy to use, and both MySQL and its C library are easy to install. If we were to continue with this project, this combination would be simple and fast to set up.