CS165 - Milestone 1

Lev Kruglyak

October 2022

1 Introduction

In this milestone, we implement the basic functionality of a column store database server with the ability to run single-table queries. Starting with a server program that starts a UNIX socket and a simple client program which connects to the server in order to send queries, we must develop a database catalog, parsing system, and some simple single-column primitive operators.

To keep the project simple, we can assume several things about the usage of our database system. Firstly, we can ignore physical issues of hardware failure or interrupted power supply / internet connection. This gives us a good amount of lee-way in error handling. Another assumption we can make is that at the peak capacity of the database loads, all of the data fits comfortably in RAM, (running on a grading server which has $\approx$ 128Gb of RAM) so we don't have to deal with streaming data from disk. Rather, we can load all of the persistent data into memory at the database startup.

2 Problems Tackled

Now let's examine some of the core problems faced in the implementation of this milestone.

- **Error Reporting:** In order to build a usable data system, we need a good way of reporting errors to the client so they can deal with them in a consistent manner. Errors are quite diverse, and detailed error messages are critical to properly debug the system.
- **Database Catalog:** We need to keep track of what tables and columns are currently loaded in the database system, and also be able to efficiently query this catalog by a system of unique identifiers. (column/table names)
- **Bit-Vectors:** To efficiently express selection results, especially large ones, we can pack selection indices in the form of bit-vectors, which represent a set of indices as bits in a large block of memory. However this approach is very finnicky based on specific implementation in C, so we must carefully do it in a way that actually improves performance.

3 Technical Description

Here we'll explain how we solved these various problems.

3.1 Error Reporting:

We found the given starter code lacking in the error reporting department; there the server returns a `message` struct which contains a status enum and a string payload. This status enum was quite large, containing all of the possible error states. We didn't like this approach, so we opted instead for a simple struct which contains a variable length pointer, and a single `SUCCESS/ERROR` enum state.

To provide valuable error messages, we also provide functions to `printf` append to a message struct, and this contains a macro to add the line number and file of the place where the error was thrown. This makes debugging a lot easier. Also, every relevant function in the execution path takes in this "global" message struct as a parameter, enabling throwing an error from any stage in the execution path; e.g. parsing, loading, executing, etc.

3.2 Database Catalog:

In order to keep track of what name corresponds to what database/table/column, we use series of nested string-pointer hashtable; one for the databases, in each database a hashtable for the tables, and in each table a hashtable for columns.

We implemented a robinhood hash table for this [1], with a simple polynomial string hashing function. A robinhood hash table uses an open addressing scheme with a clever mechanism for reordering nodes to provide optimal access time. Robinhood hashing has superior cache performance and lower memory allocation costs than a traditional chained hash table. This provided an order of magnitude improvement over a standard chained hash table optimization, especially when this table grows dynamically.

3.3 Bit-Vectors:

To optimally implement bit-vectors, we must decide an appropriate size for the entries in the array. The smallest solution would be to use `char`s and the largest solution would use `long long` or some even bigger 128 bit primitive, however there is a tradeoff. To determine the optimal size, we ran some experiments in GodBolt Compiler Explorer to see at what point the standard gcc compliler stops unrolling loops. We found that the optimal size was a `short`.

4 Challenges

The main challenge faced in this milestone was architecting the overall system and getting used to debugging/fixing code in C.

References

[1] Pedro Celis and Per-Åke Larson and J. Ian Munro (1985) *Robin hood hashing*, 26th Annual Symposium on Foundations of Computer Science, 281-288.