ECE 590:
Scalability

Faris Sbahi, Andrew Bihl, Xinghao

April 6, 2018

1 Code Analysis

We designed our Exchange Engine using Golang. One of the primary motivators for doing so is Go's built-in concurrency system, centered around the use of goroutines. Goroutines are similar to threads, however instead of there existing a one-to-one mapping of threads to goroutines, each goroutine is mapped to a group of threads. Hence, once the swap context and thread setup overhead becomes too high, Go will start mapping parallelized executions to the same threads.

Hence, each time our server accepts a request, the request is handled as part of a separate goroutine. This is our first level of parallelization.

Furthermore, we parse the received XML as part of a stream which allows us to start performing a job in parallel (e.g. buy order) as we continue reading. This seems especially useful for requests that consist of an extremely large amount of data.

When dealing with "transaction" and "create" requests, idempotency and atomicity become increasingly important because there are several places that a given request can fail. Hence, it was clear that we needed to execute certain operations atomically and rollback if a single command in the operation failed. For example, cancel transactions must both deposit money to a user's account and remove the open shares from the market. The first step of atomicity is solved using a simple mutex. Furthermore, if the shares can't be removed, then the transaction to deposit funds into the user's account is rolled back. We ignore cancel requests for transaction IDs which we've already cancelled. This provides idempotency.

Still, query transactions don't actually modify data so it seemed like an unecessary overhead to use the same mutex that we use for the other transactions. Hence, we decided to measure the effect a Reader-Writer Lock (RWMutex in Go) would have on our scalability. One concern we had was whether this could lead to deadlock between a waiting writer and many readers. However, from the docs, if a goroutine holds a RWMutex for reading and another goroutine might call Lock, no goroutine should expect to be able to acquire a read lock until the initial read lock is released.

Another important consideration is the datastore. Our initial design simply used a hashmap, which provided great speed given that data would tend to reside in memory. We recognized that we needed some sort of database so that the server can go down without transaction history being permanently deleted. Our first choice was Postgres because of its robustness and support for redundancy. Later, we wondered if we could recover the benefits we saw when we used an in-memory dictonary. Hence, we explored the possibility of using Redis as an in-memory cache. In our experiments below, we explore the effect it had on our scalability.

2 Experiments

We use a Python script (included in the testing folder) that generates the time that each function takes to complete. Hence, this allows us to measure the overhead of various cat-

egories of delays e.g. Query, Send/Receive, Parse, etc.. Therefore, these measurements allowed us to turn our focus to where can best improve scalability.

2.1 Scalability

As mentioned in our code analysis, it was left to measure whether Reader-Writer Locks substantially improved our performance. One simple test was to send 1000 requests in parallel that are reader-intensive

```
time seq 1000 | parallel -n0 "cat transaction/query/1.txt | nc localhost 12345"
```

After running the above command at various request numbers, it is clear that the work scales linearly and does not suffer from increased lock contention.

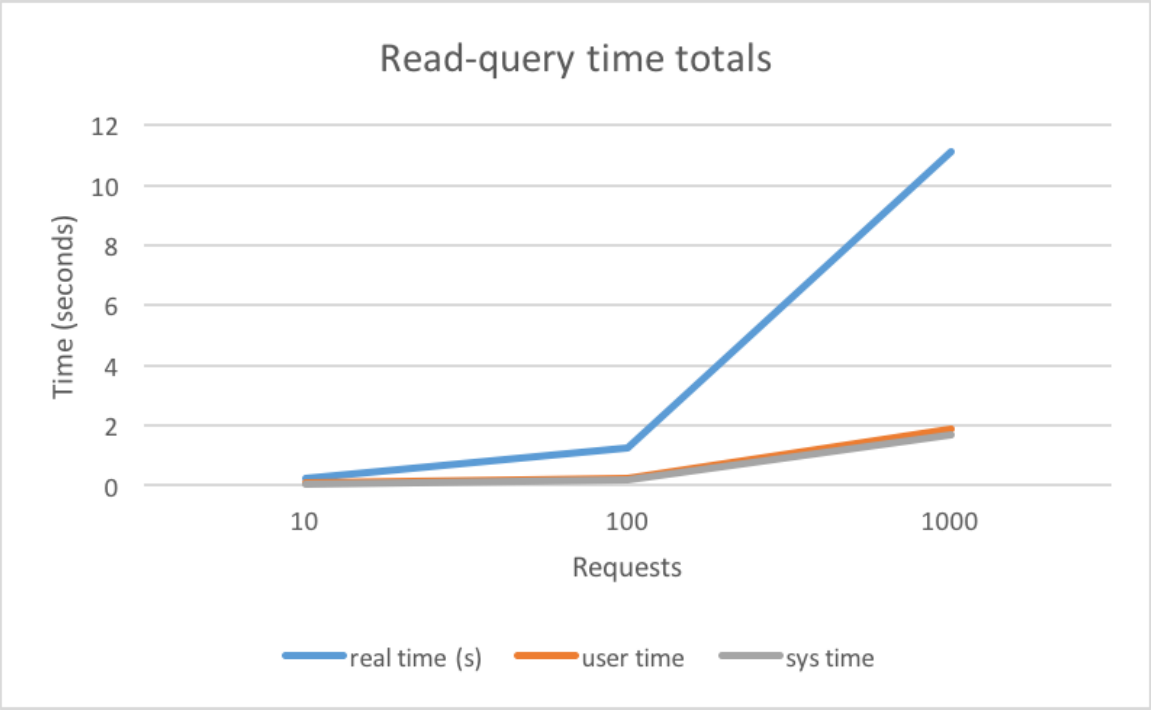

Figure 1: Read Query Times

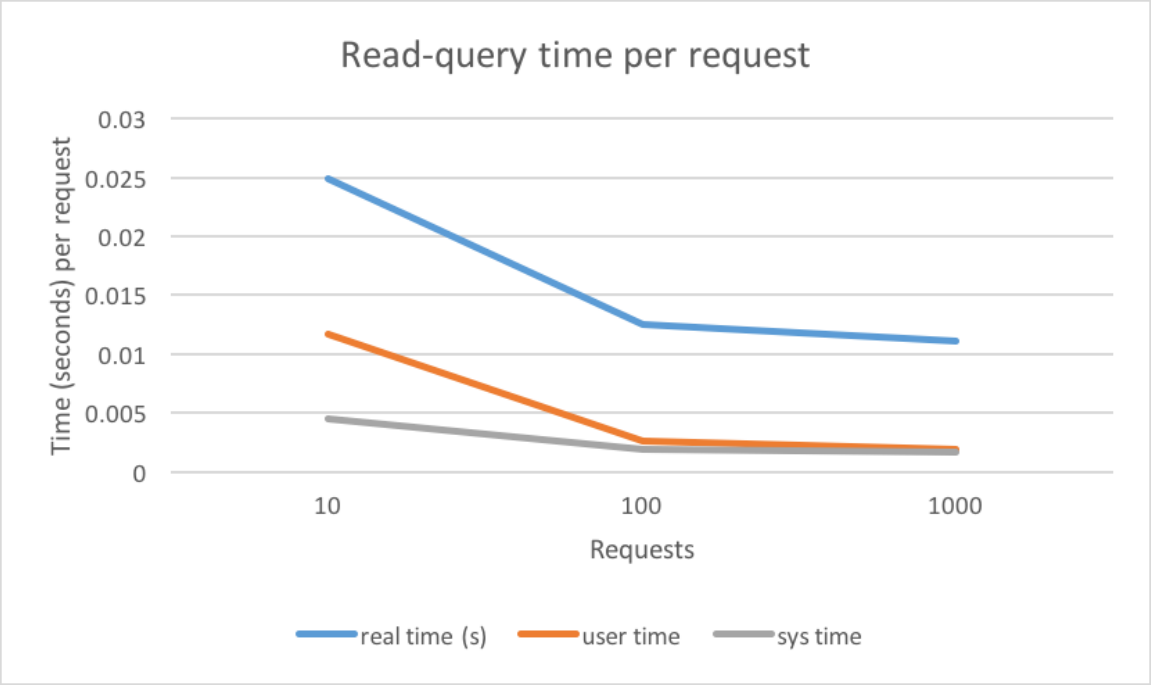

Figure 2: Read Query Times per Request

Time per request goes down as system time and communication overhead account for a smaller proportion of the overall runtime.

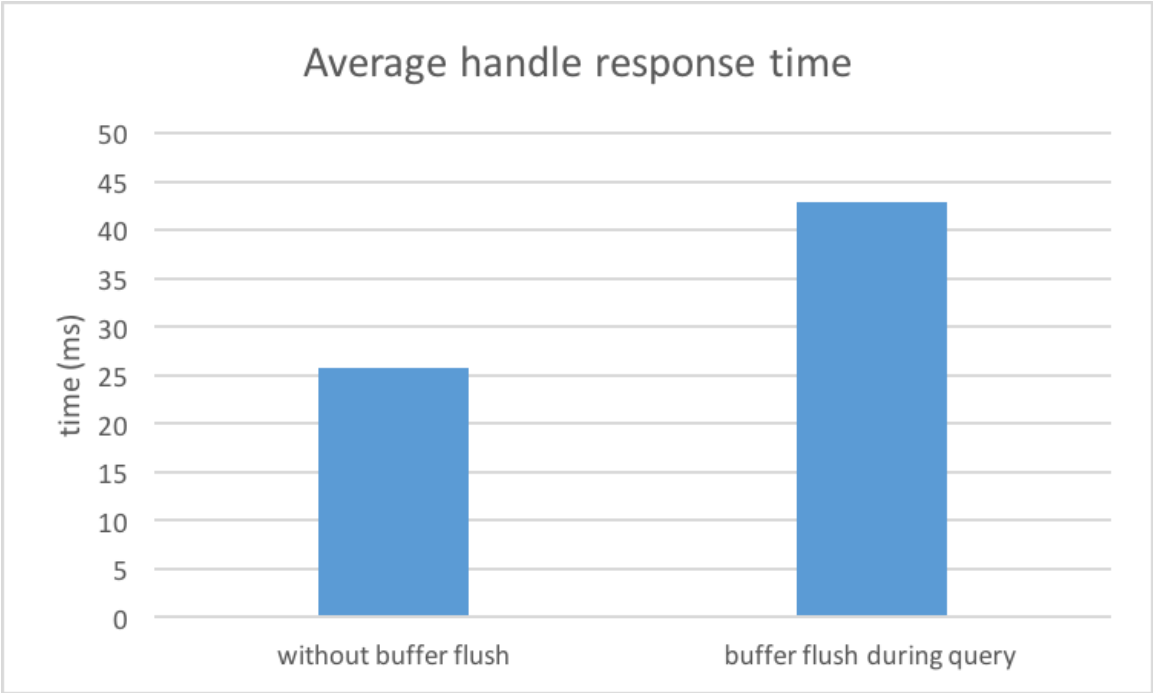

Figure 3: Scalability Breakdown

Another significant factor in runtime is handling persistence. While response times were improved with redis, it is clear there is improvement to be made in the buffer flush logic.

By writing to a local buffer and flushing separately, we avoid slowing all queries but the response time on queries. Golang buffered channels are thread-safe, allowing us to write without blocking to local memory. However, data must eventually be persisted; queries which trigger a buffer flush to the postgres database are delayed significantly.

Given more time, we would improve the logic for scheduling buffer flushing, ideally updating the database at times of low traffic if possible. We might also be able to find a more optimal buffer size.

2.2 Correctness

We developed specialized test cases that are executed as part of our `tests.sh` script. These ensure that we have correctness for the standard and edge cases that we came up with. Furthermore, we dump the database as part of our separate debug log each time we handle a request. Hence, we are able to see whether the correct values exist in the database at each step.

Furthermore, we tested for datastore failures my taking down the Redis and Postgres containers and random points of our testing and ensured that the same results were achieved.

3 Discussion

Evidently, we found that using Redis as a cache seriously improved our ability to scale.

Furthermore, we found that querying is our greatest bottleneck. One way we attempted to mitigate this, was using an in-memory cache and writing to a buffer in a queue on a separate thread. In the future, we'll look to improve the scheduling of flushing the buffer and also the eviction policy of the cache. For example, certain information, like open orders, should rarely be evicted. Hence, LRU may not be the perfect policy.

The project was a useful exercise in dealing with issues of concurrency and idempotency. We used several methods, mentioned above, to deal with this. Beyond multi-threading and mutexes, we used goroutines, Reader-Writer locks, atomic datastore operations, rollbacks, and buffered database writes.