Batch Processing Transactions for a Scalable Banking Application

Daniel P. Ryan
Department of Computer Science
Stanford University
dryan2@stanford.edu

Abstract

When replicating a database, processing transactions in batches provides significant opportunity for scalability. Since the transactions within a batch can be processed in any order, we can utilize multiple cores for processing transactions. In this way, adding more cores to the system can produce a linear increase in throughput. This work defines a set of commutativity rules for a simple banking application to demonstrate scalability. We will show that these rules maintain the consistency of the transactions while providing the opportunity for parallel processing, and will measure the performance improvements achieved by utilizing multiple CPU cores.

1 Introduction

Designing software and systems that are easily scalable is an important facet of modern applications. In an ideal scenario, one would like the capability to scale an application simply by adding more compute (ie. more CPU cores) to the system. However, this is not always achievable if the operations within that application are not commutative. In other words, the operations have to be performed in a specific order, and adding more CPU cores would require coordination between the cores. Conversely, commutative operations can be performed in any order, and there is no way to distinguish the execution order simply by examining the final result. Therefore, we can say that when operations commute, they can be implemented in a scalable fashion. [1]

2 Related Work

This work is similar to that performed in the development of SPEEDEX [2]. There, the authors create a decentralized exchange where participants can trade assets. They create blocks of transactions wherein all of the transactions within that block commute and can be processed in parallel. Their commutativity rules are structured to create a fixed valuation for each asset within each block, such that exchange rates between different assets are constant and the amount of each asset sold within the block equals the amount bought. While their work focuses on a distributed exchange, our focus will be on a traditional banking application where money is transferred between accounts. Similarly, however, we will seek to group transactions into blocks of commutative operations, such that they could be easily scaled.

3 System Architecture

In this work, we simulate a simple banking application with Accounts and Transactions, that perform actions against those accounts. The following transactions are implemented: Deposit (add some specified amount of money to an account), Withdrawal (remove some specified amount of money from an account), and Transfer (send some specified amount of money from one account to another). We implemented a Transaction Generator that can generate an arbitrary number of Transactions for an arbitrary number of Accounts with an arbitrary starting balance. Our commutativity rules are applied by our Pre-Processor component, that takes a series of transactions and splits them into batches that can be processed in parallel. Our basic commutativity rules are as follows:

1. Deposit transactions can be applied at any time
2. Each batch will consist of transactions that credit (ie. remove money from) only one account
3. We can continue adding transactions to the batch as long as the credit does not cause the account to have a negative balance
4. Transactions that credit different accounts will be processed in subsequent batches

Since these rules guarantee that money is only being removed from one account within each batch, and the sum total of all withdrawals will not cause the account to have a negative balance, they can be processed in any order and the state of our system will remain consistent with the serial processed transactions.

The results of pre-processing the transactions into batches is visualized in Figure 1. First, we apply the commutativity rules to determine which transactions can be processed in parallel. Those transactions are assembled into batches, and then the batches are serialized for processing. As long as the batches are processed in order, the transactions within each batch can be processed in any order.

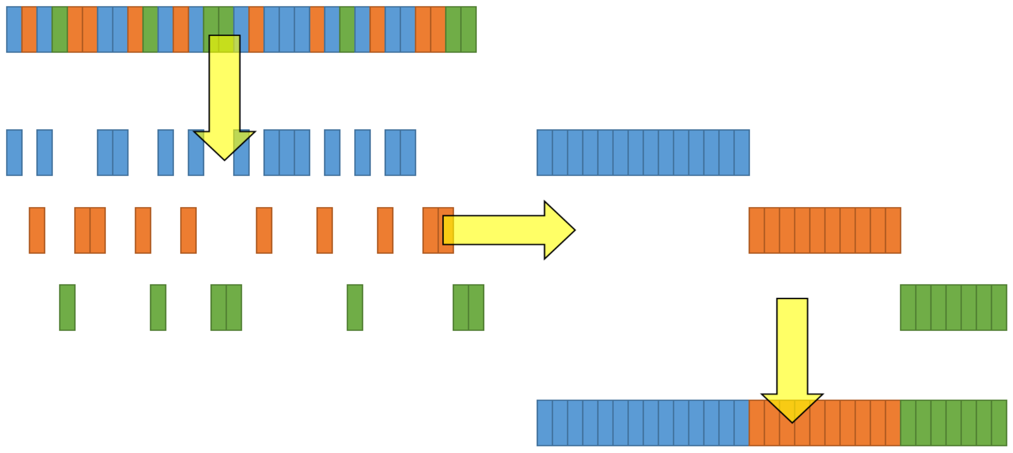


Figure 1: Batch Formation

4 Tests

We run several tests to exercise the functionality of our system and to verify that our components are performing as expected. We verify that we are able to create accounts and debit/credit their balance appropriately. We verify that each of our transactions has the desired effect on each account that it interacts with. Additionally, each transaction can be tested for validity prior to its execution to ensure that we never enter an undesired state (ie. an account with a negative balance). Finally, each transaction can be rolled back, where each account it interacted with is restored to its previous state.

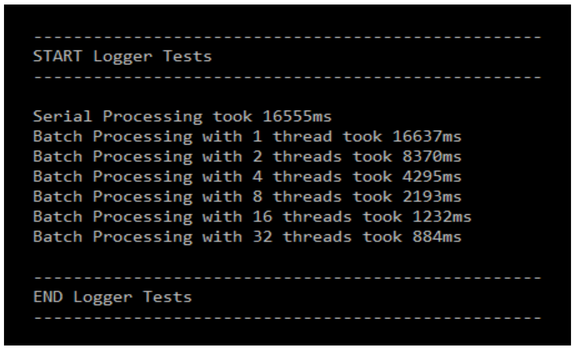
When exercising our Pre-Processor component, the intent is to verify the consistency of our batch processing. Here, we want to confirm that batch processing our transactions produces the same result as if they were processed serially. To do so, we verify that the final state of the accounts is the same in both scenarios and at no point does any account achieve a negative balance.

For our performance tests, we initialize 10 Accounts with a starting balance of $5000, and randomly generate 100,000 transactions against those accounts. For simplicity, we stipulate that no account can be overdrawn (have a negative balance) at any point in time. During the random generation of transactions, if a transaction would result in a negative balance for any account, that transaction is discarded and another is randomly generated. After pre-processing the transactions into batches, the batches are passed to our Logging component where each transaction is logged. Here, we are only simulating what would occur in a real-world database replication system. In our simplified application, we just assume that logging each transaction requires a constant time and so we add a small time delay for each transaction. To simulate parallel processing on multiple cores, we vary the number of threads that are used to process the transactions within our batches. We launch an arbitrary number of threads and stagger the transactions that each thread processes such that each of our n threads processes every nth transaction. In our performance testing we used 1, 2, 4, 8, 16, and 32 threads.

In order to achieve the full benefits of our threading approach, all tests were executed on an Amazon AWS EC2 instance (c5a.8xlarge) with 32 vCPUs, 64GB of memory, and an Ubuntu 22.04 LTS Linux platform.

5 Results

After pre-processing our set of 100,000 transactions, we have generated 357 batches with an average batch size of 280 transactions. After verifying the basic functionality of our system, we observe the results for serial and parallel processing with multiple threads, as shown in Figure 2. We first note that, as expected, the serial processing and single threaded batch processing perform virtually identically. Furthermore, we observe a nearly linear decrease in processing time as we linearly increase the number of threads.



```
--------------------------------------------------
START Logger Tests
--------------------------------------------------

Serial Processing took 16555ms
Batch Processing with 1 thread took 16637ms
Batch Processing with 2 threads took 8370ms
Batch Processing with 4 threads took 4295ms
Batch Processing with 8 threads took 2193ms
Batch Processing with 16 threads took 1232ms
Batch Processing with 32 threads took 884ms

--------------------------------------------------
END Logger Tests
--------------------------------------------------
```

Figure 2: Performance Results

Examining the batch processing results, we calculate the throughput achieved in each test case. Here, the throughput is defined as the number of transactions processed per second and we normalized the data such that the throughput is equal to 1 for single thread case. In this way, we are able to easily compare our measured results to what would be expected for a purely linearly scaling system. The results are shown in Figure 3.

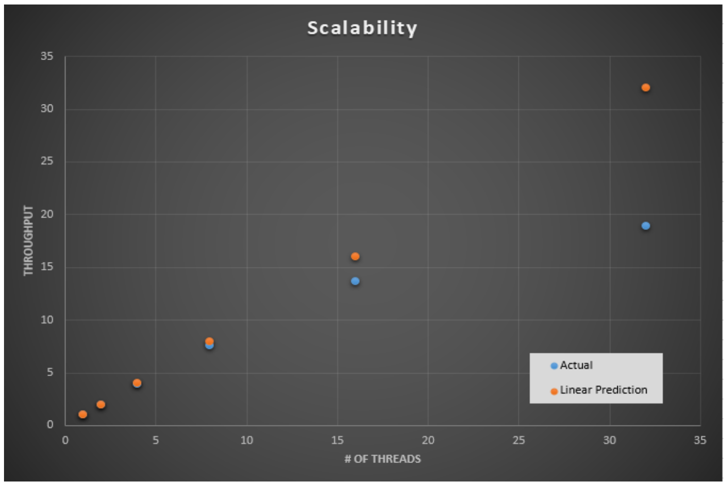


Figure 3: Scalability Results

We see that our measured results perform as predicted up to 8 threads, but then start to deviate, with our measured results lagging the predicted performance. We believe that this is likely due to the overhead required to initiate the multiple threads and the fact that the average batch size in our relatively simple test scenario is small (several hundred). In practice, we would anticipate the batch size to be significantly larger (likely, several thousand to hundreds of thousands) and this overhead cost would be less significant.

6 Conclusions

The work presented here simulates a simple banking application and demonstrates a set of commutativity rules that can be applied to create batches of transactions to be processed in parallel. We then simulate parallel processing of these batches and compare the performance to what we might expect to see by scaling up the system. We see that our system performs delivers close to the predicted throughput when utilizing up to 8 threads, then starts to diverge beyond that. We offer some hypothesis with regards to the deviation from the predicted value and note that this may be a limitation of our simplified approach. Most importantly, this work demonstrates the ability to create a banking application where the transactions can be replicated in a scalable fashion. In order to increase the throughput of the system, one would only need to increase the number of CPU cores servicing the application.

7 Git Repository

`https://github.com/dan-ryan21/cs244b_final_project`

References

[1] Clements, A. T., Kaashoek, M. F., Zeldovich, N., Morris, R. T., & Kohler, E. (2015). The scalable commutativity rule: Designing scalable software for multicore processors. ACM Transactions on Computer Systems (TOCS), 32(4), 1-47.

[2] Ramseyer, G., Goel, A., & Mazières, D. (2021). SPEEDEX: A Scalable, Parallelizable, and Economically Efficient Digital EXchange. arXiv preprint arXiv:2111.02719.