CS 223 Project 1: Concurrent Transaction Processing

Ricky Cheng
University of California, Irvine
`shengrc`

Isaac Shvartsman
University of California, Irvine
`shvartsm`

1 Introduction

The purpose of this project is to build a concurrent transaction simulator that mimics a real-life transaction workload of an IoT based system on a database system. Later sections in this report will discuss the design of the simulator, the conducted experiments, and the obtained results.

2 Design and Methodology

2.1 Data

The data used for transactions are from TIPPERS. The dataset contains two workloads, a high concurrency workload and a low concurrency workload, used for our experiments. The high concurrency workload contains 15018400 queries and the low concurrency workload contains 7998798 queries.

Before conducting the experiment, we extracted all the data of a workload from its respective files. Since each files contain blocks of consecutive queries to the same table, we then shuffled the data to try to increase conflict dependencies between transactions. The shuffled data is then re-sorted by timestamp to form the final query set.

2.2 Simulator Design

Simulating concurrent transactions requires a database along with clients capable of sending transactions in parallel. Normally, an experiment of this nature would involve a database server connected to a network of hosts that send transactions; however, given the lack of resources, this simulation was done on a single computer using multiple processes/threads to act as 'hosts' that interact with the database on the same machine. Given enough computational resources, this should feasible since the bottleneck depends on the latency of the database and not the scheduling of host processes on the CPU.

We designed two models of the concurrent simulator based on conventional methods of parallel processing. The following subsections will go into details on the model and the challenges we faced.

2.2.1 Design 1

This initial design takes inspiration from the work-queue model in which all workers maintain one single shared input queue and each worker extracts one task at a time from the queue and work on it.

However, we found a few issues with this approach. The first issue is that we observed resource contention on the queue as the number of workers increases. Since only one worker may access the queue at anytime, this may block other workers, leading to delays. The second issue is that we are unable to monitor progress of experiments effectively.

2.2.2 Design 2

The issue with blocking in Design 1 led to the second design in which each worker receives its own list of transactions, thus eliminating the need for a shared input queue. All the workers are also given a shared output queue to output its intermediate results at certain intervals, thus allowing an easy method for monitoring progress. The main disadvantage of this method is that some workers may finish earlier and thus reducing the multiprogramming level towards the end of the experiment.

We did face a significant challenge with this design. Since we used a multiprocessing framework, each worker is a entirely new process and inherits all data from its parent. This led to significant memory bloat and, coupled with Linux's copy-on-write, repeatedly led to a system restart. To get around this, we loaded the inputs after the worker processes have started and sent each worker's inputs through a shared pipe.

2.3 Software & Hardware

We used PostgreSQL as the database to experiment on. In particular, we used a Dockerized version of PostgreSQL to easily guarantee the same initial state of the DB at the start of each experiment. The simulator was written in Python and the `Multiprocessing` module was used extensively to build our concurrent framework.

The machine used to conduct our experiments is a desktop workstation running Ubuntu 22.04 with 32GB of RAM, 2TB of M.2 NVMe SSD, and an Intel i9-9900K CPU with 8 cores and 16 threads.

3 Experiment Setup

We performed our experiments by varying three parameters: multiprogramming level (MPL), isolation models, and transaction sizes. For each concurrency workload, we tested five MPL, three isolation levels, with five different transaction sizes for a total of 75 experiments per workload. The experiments are expected to complete in at least 21 minutes and has a capped run time of 30 minutes due to time constraints.

3.1 Multiprogramming Level

Multiprogramming level refers to the number of active transactions allowed concurrently. We varied our MPL across these values: 16, 32, 64, 128, 256. We expect to see an increase in throughput as MPL increases. However, if there are many conflicts (i.e. high concurrency), then we will see degraded performance in higher MPL.

3.2 Isolation Level

PostgreSQL allows users to specify four different isolation levels. In actuality, only three levels are implemented internally as PostgreSQL does not allow read uncommitted isolation. The consistency levels tested are, in order of least strict to most strict: read committed, repeatable read, and serializable. We expect that as the strictness of isolation increases we will observe a decrease in throughput and an increase in latency.

3.3 Transaction Sizes

We also experimented with varying the sizes of transactions, ranging from 10 to 150 records. Of course, we expect to see the throughput to decrease by at least a factor of size and, likewise, the latency to increase by at least a factor of size. However, increasing transaction sizes should lead to more conflicts so we should see degrading performance on higher transaction levels.

4 Results/Analysis

This section presents the results obtained from our experiments and offers an analysis of the findings. The data in this section can be found in the results folder submitted as part of the assignment.

4.1 Throughput Analysis

Since transaction throughput depends on the size of transactions, we will be looking at query throughput, which is defined as

$$\text{Query Throughput} = \text{Txn Throughput} \times \text{Txn Size}$$

This allows us to directly compare performance between experiments.

4.1.1 Low Concurrency

Given the low concurrency workload size of 7998798 queries, the maximum achievable throughput is 6348 queries/s.

The throughput results for the low concurrency workload is given in 1. As observed, the database is able to process at the maximum throughput for isolation mode of `READ COMMITTED` and `REPEATABLE READ` on all transactions of size

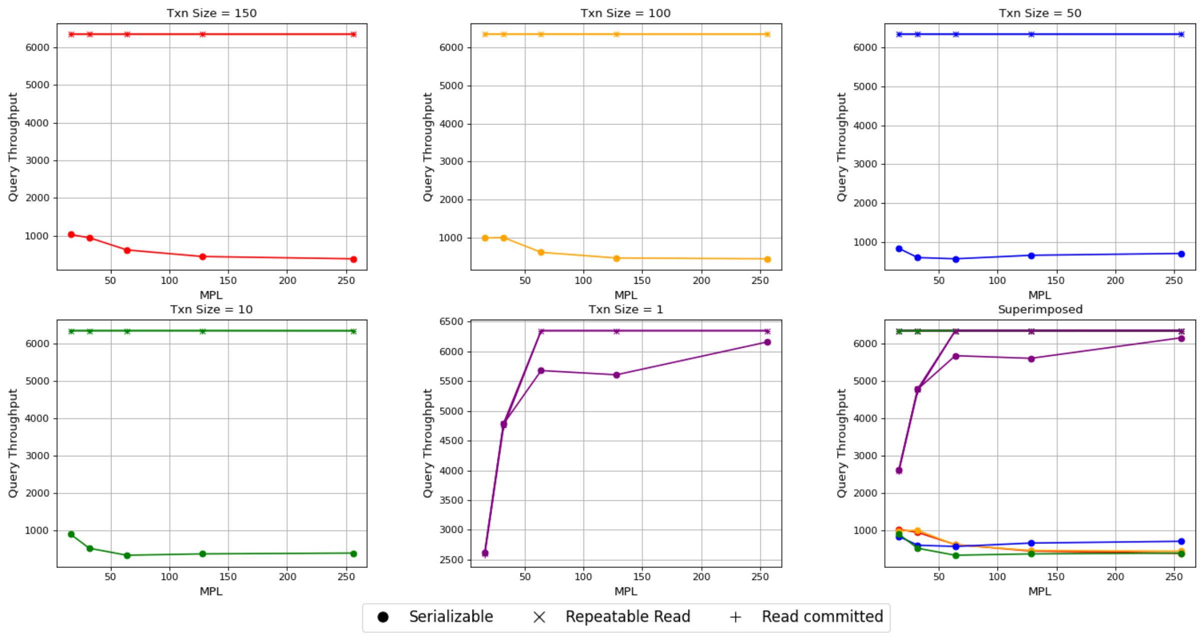

Figure 1: Query throughput of different transaction sizes on low concurrency workload.

10 and above. As expected, serializable reduces throughput considerably due to its stricter isolation guarantees. An interesting result is that in serializable mode, the throughput when transaction size is 50 is largest with higher MPL.

For transactions of size 1, we see that performance is identical across the isolation levels for MPL of 16 and 32. It is theorized that the speed at which transactions are sent is too slow, so the database essentially has capacity to do more work. Also, since this has only one query per transaction, we can see that performance of `SERIALIZABLE` is much better since there are likely fewer conflicts.

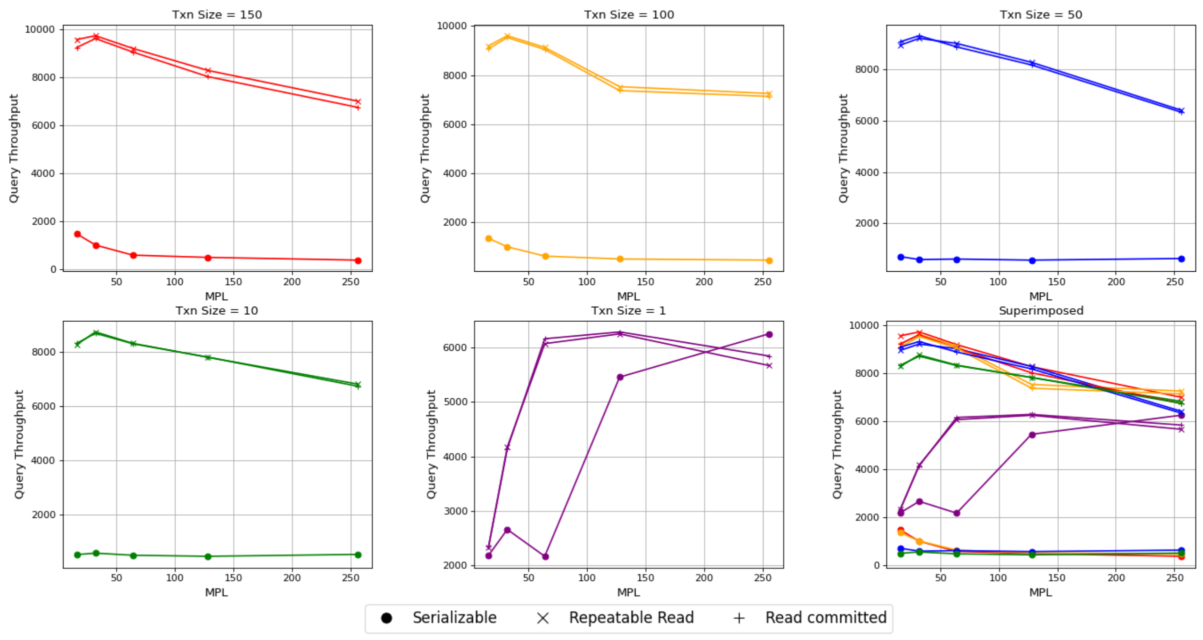

Figure 2: Throughput of different transaction sizes on high concurrency workload.

4.1.2 High Concurrency

Given the high concurrency workload size of 15018400 queries, the maximum achievable throughput is 11919 queries/s.

The throughput results of the high concurrency workload is given in 2. We can see that, unlike the low concurrency experiment, no configuration was able to achieve the maximum query throughput, with the maximum throughput having the value of 9714 queries/second with the parameters MPL of 32, isolation level of `REPEATABLE READ`, and transaction size of 150.

For transaction sizes 10 through 150, we see a general trend of slight increase in performance as MPL increases from 16 to 32 followed by a gradual decline in performance as MPL increases further. We believe this to be an issue of data contention and blocking in the database, which will cause blocking and transaction failures beyond a certain MPL.

We also observed that `REPEATABLE READ` generally has a higher throughput than `READ COMMITTED`. We believe this is because for `REPEATABLE READ`, PostgreSQL uses the same snapshot taken at the start of each transaction throughout the transaction's execution. This means that it is computationally cheaper, but does come at a cost of serialization error since the data cannot be modified or deleted if any transaction reads it. Since our queries consists of only `INSERT` and `SELECT` queries, it makes sense why `REPEATABLE READ` has a faster throughput.

For transaction size of 1, the behaviour is largely the same as what is observed in low concurrency. One interesting observation is that larger transactions attain greater throughput. We believe that this may be due to the fact that larger transactions means that the database can spend less overall time maintaining a conflict table.

4.2 Average Response Time Analysis

For this metric, we define the average response time to only be the average time of a successful transaction. We do not count time spent on failures, rollbacks, and retries.

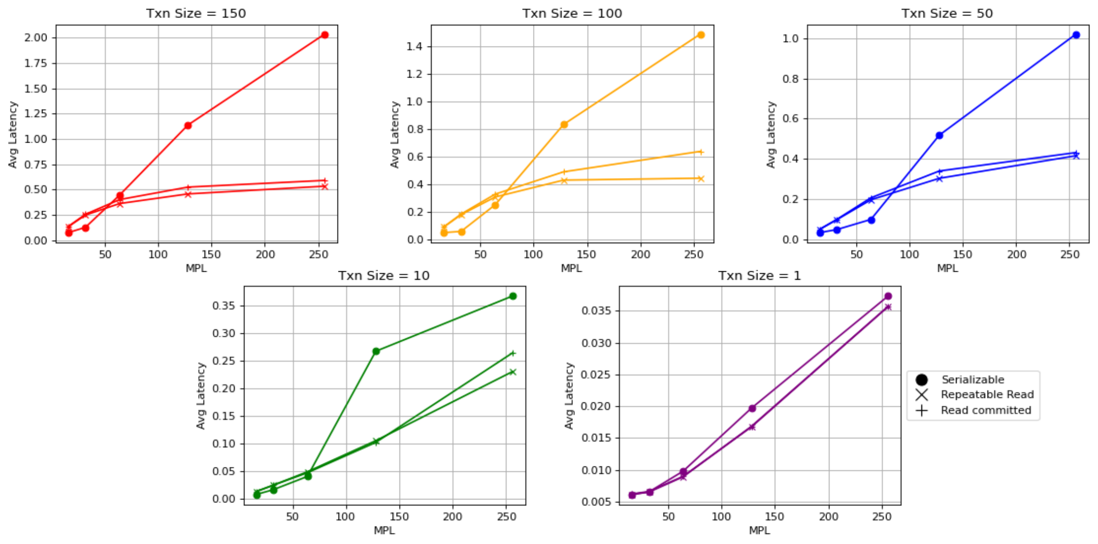

Figure 3: Average latency of low concurrency workload.

4.2.1 Low Concurrency

We see in 3 that the response time for transaction sizes 10 through 150 increases as MPL increases; though the rate of increase decreases. This is likely due to the overhead incurred with concurrency.

We can also see here that `REPEATABLE READ` has a faster response on average than `READ COMMITTED`. However, the reason why this is not reflected in the throughput is likely due to the fact that our transactions are schedule, so a faster response time just means a worker process will wait for a bit longer before sending its next transaction.

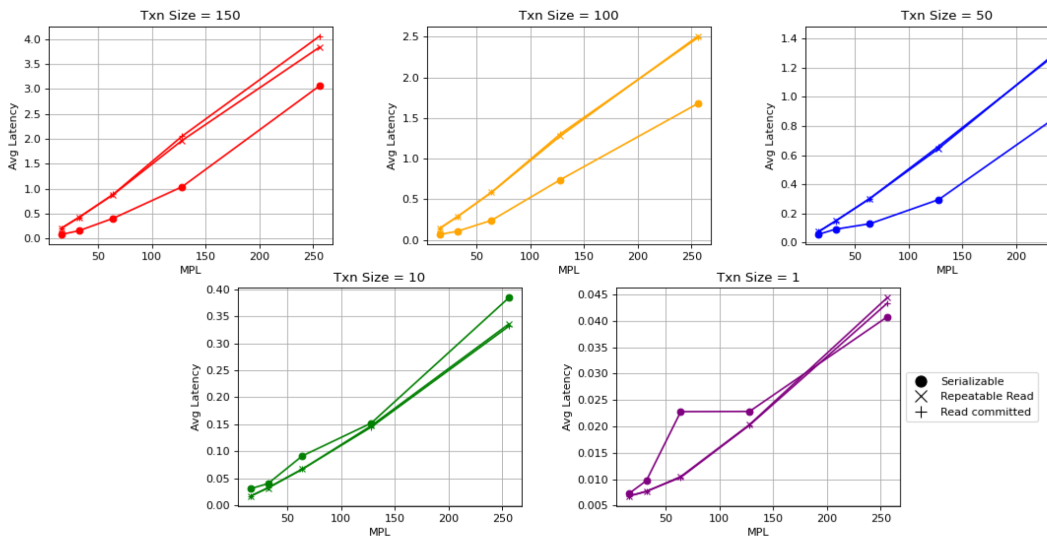

Figure 4: Average latency of high concurrency workload.

4.2.2 High Concurrency

The results of latency for the high concurrency workload are given in 4. As expected, latency increases as MPL increase. However, it seems that latency increases linearly as a function of MPL, which is different from the observations for low concurrency. This would explain why the database sees a decline in throughput as MPL, since each process takes a longer time to perform the same work.

We also see that for transaction sizes 50 through 150, `SERIALIZABLE` has the fastest response time. This is likely due to the way we measure response, since we only measure successful responses. In `SERIALIZABLE` mode, the database is much likelier to throw a serializability error, causing a failure needing a retry. Thus, transactions that successfully commit are much likelier to encounter fewer blocking, leading to a faster response time.

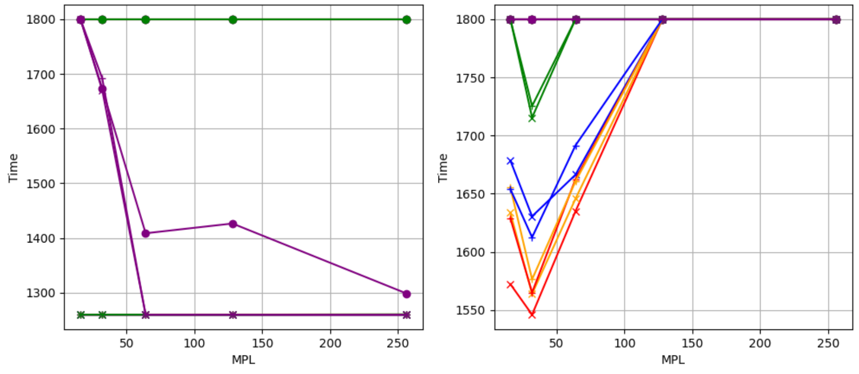

Figure 5: Total workload time of each experiment. The graph on the left is the low concurrency workload, and the graph on the right is the high concurrency workload.

4.3 Response Time of the Whole Workload

The response time of the workload is given in 5. We can see that in all with the exception of transaction size of 1, all worksloads in `READ COMMITTED` and `REPEATABLE READ` finished in the time of 1260 seconds, while no `SERIALIZABLE` workload finished within our cap of 1800 seconds. For the high concurrency workload, the graph reflects the throughput graph. Most of the experiments with lower MPL did finish within 1800 seconds, but none finished in 1260 seconds.

5 Conclusion

Building this simulator was very insightful into how concurrency, consistency, and transaction sizes all work towards determining overall system performance. To recap our report, there are several key insights that should summarized. We observed that increasing the MPL led to decreased throughput, which could be viewed as a result of increased resource contention and conflict dependencies. The repeatable read isolation level achieved higher throughput, whereas serializable level showed longer response times. With respect to transaction sizes, we notice that larger transactions led to increased query throughput and increased latency Also, larger transactions could potentially result in better overall performance, since the database is spending less time maintaining conflict tables.