Sam Dowell

CS 274

6/12/2017

Course Project: Concurrency Control

Introduction

In this project I implement two concurrency control protocols in a main-memory database – strict two phase locking and pessimistic Hekaton. Strict two phase locking requires a transaction to acquire a lock for every operation, whereas Hekaton utilizes the notion of multiple versions in order to allow finer grained temporal access to data. This means that Hekaton should have better performance for conflicting concurrent transactions. The initial experiments with large benchmarks did not yield very intuitive results, but after restricting the benchmark to having more conflicting transactions it was clear the Hekaton was able to have more efficient concurrency.

Results

For the initial benchmark I ran, I used a database with one million items. The benchmarks consists of two types of transactions: read-only and read-write transactions. Read-only transactions consist of 4 read operations, while read-write transactions consist of 2 read and 2 write operations. When generating transactions for the benchmark, keys are picked uniformly across the space of possible keys. The following results are for workloads with different ratios of read-read to read-write transactions, running with 100 clients and 100000 transactions.

Unfortunately, these results do not paint a very clear picture as to which protocol is better at handling different types of loads. This is most likely because since the key space is so large, there are not many conflicting transactions and thus not much differentiation in concurrency control implementation. In order to further investigate this, I ran another experiment in which the keys are chosen from a smaller space, thus leading to more conflicting transactions. The following plot is drawn from this experiment.

This experiment confirms our expectation that Hekaton scales better with a larger amount of conflicting transactions. The next experiment that I ran is for scaling up the number of clients. Using a consistent benchmark, I scale up the number of threads submitting transaction to the server while keeping the number of read-only transactions equal to the number of read-write transactions.

The intuition is that Hekaton should perform better on scaling out as there will be an increased number of concurrent transactions and thus increased need for efficient concurrency control. These results, however, did not paint the clearest picture as to how Hekaton scales in comparison to two phase locking. This is likely in part due to the nature of how the experiment was run. This experiment was run using multithreading in Python on windows, which is certainly a detriment to the performance of the clients.

Conclusion

The traditional two phase locking concurrency protocol utilizes a read-write lock on each database item using a single version. This means that any time a transaction does a write on some item, no other transaction can read from or write to that item. Hekaton, on the other hand, utilizes multi-versioning in order to increase the granularity of concurrency. By saving a history of versions, a transaction can read previous versions of a data item even if the most recent version is being updated by another transaction. This increases the amount of potential reading that can be done concurrently on a single data item in the presence of writes. This was confirmed by the second experiment which consisted of a large amount of conflicting transactions. As I increased the number of writes, Hekaton scaled much better than traditional two phase locking.