Embarrassingly parallelizeable preprocessing for deterministic multiversion concurrency control

Sylvan Zheng
Yale University
sylvan.zheng@yale.edu

Advisor Daniel J. Abadi
Yale University
dna@cs.yale.edu

Advisor Jose M. Faleiro
Yale University
jose.faleiro@yale.edu

ABSTRACT

Multiversion concurrency algorithms offer many advantages in database management systems, most notably the ability to improve throughput because read-write conflicts can be avoided. Traditionally the improvement in performace must be paid by sacrificing serializability in the database system. Fully serializable multiversion systems must significantly constrain the permitted read and write operations, often leading to performance significantly worse than single-version systems. However, recent advances in the literature propose novel, highly performant algorithms that are both multiversioned and serializable.

This paper proposes an improvement to the previously proposed BOHM system, removing an important scalability bottleneck in the previous design that limited the ability of the system to concurrently preprocess transactions before passing them to the next stage of the BOHM pipeline. The improvement adds a simple preprocessing layer to the architecture that is highly parallelizable and significantly improves the throughput potential of the BOHM system.

1. INTRODUCTION

Databases that use multiversion concurrency control algorithms maintain multiple copies, or versions, of all database records (as opposed to other systems which only maintain a single copy of each data record and overwrite it as necessary). The multiversion system can thus allow parallel reads and writes to take place, as long as the proper bookkeeping is done to ensure that the writer and reader are accessing their relevant versions. As memory (especially main memory) becomes cheaper, multiversioned systems have seen a corresponding increase in use.

However, the additional bookkeeping required by multiversioned systems causes a severe drop in concurrent performance. Many modern multiversioned systems impose a less strict demand on serializability (known as snapshot isolation) in order to compensate for this performance cost. Unfortunately, snapshot isolation can suffer from certain serializability violations, such as the write-skew anamoly which results in a database state that could not have been reached by any serial execution order. However, recently the BOHM algorithm has been introduced, a multiversion concurrency system that is both fully serializable and performant [1].

The Bohm system introduces a wholly new system based on intra-transaction parallelism and separation of versioning and execution concerns. This allows each Bohm thread to operate almost entirely with no coordination required with other threads, greatly increasing parallelism. The key issue that this paper attempts to address is the fact that the lack of inter thread communication means that a nontrivial amount of identical work is performed across multiple threads, effectively being executed in serial order and creating a scalability bottleneck. The Bohm architecture as relevant to this paper is summarized in greater detail in section 2. The curious reader is referred to the full paper for more [1].

2. BOHM SYSTEM DESIGN

Bohm is separated into two main layers - the scheduling or concurrency control layer, and the execution layer. The threads of each layer partition the database records among themselves, so that the work done by one thread will be guaranteed to not affect the others. Transactions are processed in each layer in batches, thus amortizing the cost of synchronization between the different layers. This also removes the need for expensive, synchronized central timestamp allocation.

The BOHM algorithm guarantees serializability by requiring the full writeset of the transaction to be declared prior to processing - in this way the system can pre-emptively process and schedule each transaction deterministically according to the contents of its writesets and readsets. The scheduling layer can thus create empty versions before the transaction is actually executed. When the execution layer receives the same transaction, it is guaranteed that the proper version numbers have already been allocated and assigned.

However, the issue lies in the fact that every thread in the scheduling layer must nevertheless examine each item in every transaction in every batch that passes through the system in order to detremine whether or not that item falls in the domain of the given partition. Since this identical work is performed by every scheduling thread, in effect all par-

allelism is lost and a scalability bottleneck appears. This paper introduces a third layer in addition to the existing scheduling and execution layers, a preprocessing layer that concurrently analyzes transactions and routes them to the relevant scheduler threads in order to eliminate this bottleneck.

3. SYSTEM DESIGN

The new preprocessing layer is placed first in the transaction processing pipeline, ahead of the scheduling and execution layers. Each preprocessor worker thread works on its own batch independently, ensuring only minimal coordination is needed between threads and maximizing potential parallel performance.

3.1 The Process of Preprocessing

In order for the execution layer to successfully synchronize with the scheduling layer, each scheduling worker thread needs to receive the same batch object. In order to avoid unnecessary iteration in addition to memory management headaches, we attach to each batch and transaction a single data structure indicating exactly which items and transactions are relevant to any given partition.

Each transaction is assigned an array with size of the number of partitions, *readstarts* and *writestarts*. $*starts[i]$ then indicates the index within *readset* or *writeset* of the first read or write item that is relevant to the ith partition. Since each item can only belong to one partition, each transaction item is assigned an integer field *next* that simply holds the value of the next index of *readset* or *writeset* that is relevant to the partition. The *next* value of the last item is -1. It is then relatively easy for the scheduler layer to follow this linked list and only examine keys that are relevant to it, completely skipping over any that are not.

The problem of knowing which transactions to look at within a batch are solved in a similar way. Each transaction contains a linking array *next* also sized to the number of partitions. Since each batch is totally ordered, a transaction can be uniquely identified within a batch by its array index within the batch object. The entry $next[i]$ then indicates the index of the next transaction within the batch that contains keys that are relevant to partition i. Each scheduling thread can thus examine precisely those transactions that are relevant to it.

One concern is that the shared memory accesses (since each of these intial array lists are allocated in the same block) create potential for cache coherency problems, since multiple threads are competing to access the same cache line that holds the *readstarts* or *writestarts*. Further consideration and experimentatino is needed to determine the full extent to which cache problems as outlined here can negatively affect the system's performance.

3.2 Organizing the Preprocessing Threads

A nontrivial concern is that of how to organize the threads within the preprocessing layer. It is critically important to maintain the deterministic nature of the system; batches must be output to the scheduling layer in the same order that they arrive in. If preprocessor threads are to be able to process batches in parallel then some kind of coordination is needed to preserve this total ordering.

This issue is solved with the use of a leader-worker architecture. The leader thread's primary job is to simply hand off incoming batches to the worker that is next in line in a round robin fashion. The leader also maintains a reference to the worker whose batch is required next for output, periodically checking if it is ready to be passed along to the scheduling layer. In this way a slow preprocessor thread only slightly hinders the flow of data; all other worker threads can continue working and simply buffer their batches in an output queue while they wait for the leader to pick them up. Meanwhile the leader can still listen for incoming batches and distribute them to worker threads as necessary, ensuring progress continues even in the face of one slow worker thread.

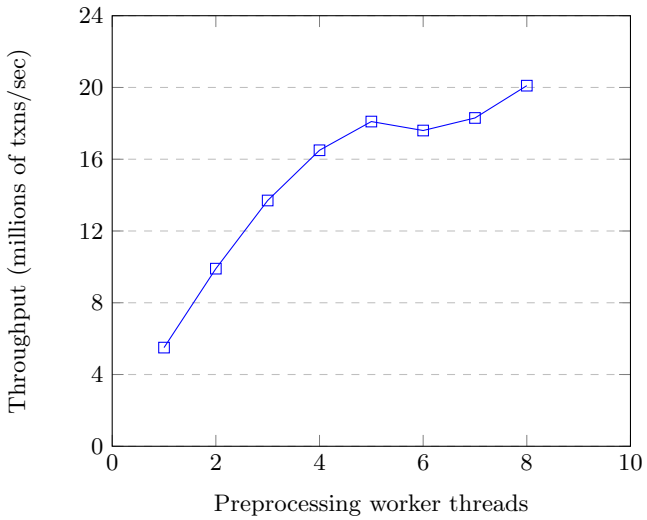
The main disadvantage of this approach is that the leader thread cannot actually perform any work, so a minimum of two preprocessor threads are needed to do complete the task. However, with the advent of machines with extremely high core counts we expect this to have a relatively negligible effect on overall performance.

4. EXPERIMENTAL RESULTS

All experiments were run on an x86_64 Intel(R) Xeon(R) CPU E5-2650 v3 2.30GHz machine with 10 cores.

4.1 Parallelizable Preprocessing

The first set of experiments aim to evaluate the parallelizability and performance of the preprocessing layer in isolation. This paper's main contribution is the introduction of a preprocessing layer that shows minimal overhead costs for additional threads and continuing performance gains as threads are added to the system.

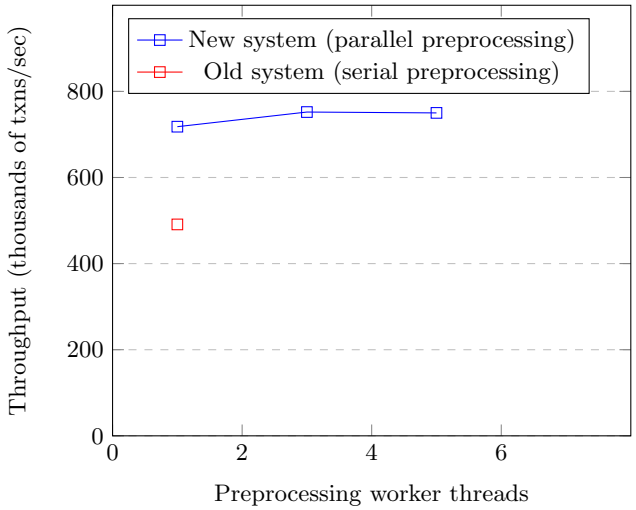


While not quite linear in nature (a slight taper off can be observed starting at approximately five worker threads), the data shows consistent gains in performance as threads are added to the system. Further exploration of multi-tiered distribution systems on machines with higher core counts may offer further insight into the layer performance.

4.2 End-to-End testing

For the second round of experiments we compare the performance of the new three layer system to that of the old two layer system across all transaction processing stages. Initial results show substantial improvement in the new system with parallelized preprocessing abilities.

Throughput comparison - 2 worker threads, 2 scheduler threads



Here we see that the new system even with a single worker thread (effectively performing preprocessing work in serial) offers a near 50% throughput advantage over the old system. Also interesting is the observation that the performance does not significantly improve as more preprocessor worker threads are added to the system.

5. CONCLUSIONS AND FUTURE WORK

Introducing a dedicated preprocessing layer significantly improves the overall performance of the Bohm system. The preprocessing layer itself is simple, relatively thread-local, and highly parallelizable. The work done by the preprocessing thread significantly reduces the workload of the scheduling layer, which can be seen by the fact that throughput is almost 50% higher even with a single, serial preprocessing thread. It is interesting to note however that the overall impact of increasing preprocessing parallelism is more or less negligible, especially after 2-3 worker threads. We hypothesize that it is because of the preprocessing layer's relatively small workload compared to the work undertaken by the scheduling and execution layers that the capacity of the scheduling and execution threads become the main bottleneck of the system. Adding additional preprocessing threads thus does not further improve performance.

Because our testing machine only has 10 cores it may be illuminating to repeat these tests with higher counts of both scheduler and execution threads. In this way a more direct measurement can be taken of the impact of the preprocessing layer under the high levels of concurrency for which it was designed.

6. REFERENCES

[1] D. Abadi and J. Faleiro. Rethinking serializable multiversion concurrency control. *VLDB*, 2015.