Disk Storage in a Deterministic Database

Philip Shao

December 9, 2011

Abstract

In database systems where data resides in both RAM and disk, it is imperative that as few data requests as possible are made to disk, since the speed of disk access is orders of magnitudes slower than that for RAM. In typical mixed storage database systems, since the final state of the database after a set of concurrent transactions is guaranteed only to be equivalent to the state that would have been reached if the transactions had executed in some serial order, it is often profitable to reschedule transactions touching data not pegged in RAM until the data is available after it has been read from disk. This work considers the issue of data availability in the context of a deterministic database system where the final state of the database is equivalent to the state that would have been reached if the transactions had executed in a specific, predetermined serial order. In particular, this paper discusses the constraints imposed by the inability to arbitrarily reorder concurrent transactions, techniques to minimize the number of requests sent to disk, and the performance of the deterministic database utilizing such techniques in the context of various data access patterns.

1 Background and Introduction

In recent years, with the decrease in price of RAM and the improving techniques for ensuring the durability of data in volatile storage, much interest in the database research community has been directed towards systems whose data reside entirely RAM. But despite this trend, the amount of hardware necessary for a purely in-memory approach to certain data-intensive applications, such as data warehousing, remains prohibitively expensive. Thus for any database targeted at such applications, the ability to perform well when not all data can feasibly fit in main memory is crucial. The performance of these systems depends on how effectively they can avoid cache misses, when data is requested that is not available in main memory and must first be fetched from disk.

For deterministic database systems, such as Calvin, the consequences of a cache miss are somewhat more costly than for traditional database systems. Unlike traditional systems, deterministic systems do not have the liberty of rescheduling transactions incurring cache misses until later when the data has
been read from disk and made available to memory. Instead, since the deterministic database must execute transactions in a manner consistent with a predetermined ordering, transactions must hold locks on data until the data is read from disk and database throughput grinds to a halt as subsequent transactions idly wait for the first transaction to release its locks, which it cannot do until its expensive disk reads complete successfully.

To illustrate the potential difference the order of execution can make on overall execution times in a deterministic database, we consider a situation where the cost of disk access $C_D$ is one hundred times as long as the cost of main memory access $C_M$, such that $C_D = 100 \times C_M$. Data item $D_0$ resides in disk while remaining items $D_1...D_{100}$ reside in main memory. Now we consider a batch of 100 transactions $T_0...T_{99}$ such that each transaction $T_i$ touches two data items $\{D_i, D_{100}\}$, so that no two transactions can execute concurrently.

Thus, executing transactions in order of increasing $i$ would have the all the transactions $T_1..T_{99}$ waiting on $T_0$ to complete and has cost $99 \times C_M + C_D = 199 \times C_M$. On the other hand, if we had, instead, asynchronously first performed a “prefetch” on item $D_0$, and then scheduled $T_0$ last, the overall cost would be $\max\{99 \times C_M, C_D\} + C_M = 101 \times C_M$, since $T_0$ begins executing only when all previous transactions have completed and the data it touches has been placed in main memory.

From this example, we can make the observation that the cost of executing a given batch of transactions can be reduced if the deterministic system delays the execution of “cold” transactions, transactions touching data not available in main memory at the time of the transaction ordering, until after the originally unavailable data has been acquired from disk and placed in main memory. Thus, the deterministic system can mitigate the costs of offloading data to disk if the a priori state of system memory is such that vast majority of incoming transactions will not be cold. Additionally, we show that if the number of cold transactions is relatively small and the time, until the data necessary for these cold transactions is available in main memory, can be predicted with reasonable accuracy, then deleterious effects to throughput and average latency per transaction can be significantly reduced.

2 Calvin and Disk Storage

Calvin addresses the issue of transaction processing getting backlogged behind a cold transaction by keeping track of a list of keys pointing to data items that are pegged in memory. In much the same way we demonstrated in our example above, any time the transaction sequencer encounters a request for a cold transaction, it sends an asynchronous Prefetch call to the disk layer that immediately returns an estimate for how long the system should wait before scheduling the transaction, which Calvin’s scheduler then uses to delay the execution of the transaction by an appropriate amount of time. If the estimate is accurate, the cold transaction will execute immediately after all its data has finished loading into memory. With this approach, the overall time taken to
execute the given transaction has not increased, but delaying acquisition of
locks until the required data is pegged significantly reduces the time that the
transaction must hold the locks. This property, again, was illustrated in our
example where $T_0$, if executed first, holds locks for $C_D = 100 \cdot C_M$, but when
$T_0$ is prefetched, the amount of time $T_0$ holds its locks is dramatically reduced.

To better observe Calvin’s performance in mixed storage systems, a disk-
based storage layer, with an enhanced CRUD interface, was built that facili-
tated both fetching prediction and the tracking of records that were pegged in
memory. This storage layer enforced a allocation/deallocation pattern, similar
to the usage in operating systems, requiring that incoming reads and writes to
particular data items be preceded by an explicit Prefetch command from the
scheduler. Prefetch, as mentioned above, asynchronously queues up a disk fetch
if the data item is not currently pegged in memory. In addition, Prefetch re-
turns 0 if the data item is already pegged in memory or an estimate in seconds
(represented as a double-precision floating point) of time until availability in
memory. Thus, at least for single-node transactions, Calvin need only run a
series of Prefetch commands on required data items and automatically gets a
list of which ones are cold, since cold data items correspond precisely to those.
Also, when a transaction completes successfully, Calvin sends an Unfetch sig-
nal to the storage, which then proceeds to page the data to disk if no other
transactions require the data.

In order to maintain speed and safety in a multithreaded execution environ-
ment, storage keeps track of a mutex-protected counter of the number of active
requests for any given data item. In addition to reducing extraneous reads from
and writes to disk, this design allows for elegant methods to maintain a reasona-
ably accurate a priori state of system memory and to estimate which data items
are hot. For example, commonly accessed data items can be kept permanently
pegged in memory via a Prefetch signal without an eventual corresponding Un-
fetch. Computing which data items should be pegged in memory long-term is a
matter of periodically querying the storage for the number of active requests and
using a threshold operation whereby permanently pegged keys (if unaccessed for
too long) can be released and unpegged keys (if accessed frequently enough) can
become permanently pegged.

For delay prediction, the internal state of the storage layer maintained for
each data item allows for straightforward case analysis. If the data is not pegged
in memory, Prefetch will return the $\hat{C} = \max \{ \frac{1}{\text{throughput}}, \text{latency}_\text{read} \}$. Fur-
thermore, if a data item is particularly popular, and has received multiple
requests from disk before the read completes, Prefetch can presume a tem-
poral distribution of requests and make an educated guess as to when the
data might be ready. A simple linear decay function, for example, results
in an estimate of $\hat{C}_{\text{linear}} = \hat{C} / \text{requests}_\text{active}$, where each subsequent request
increments $\text{requests}_\text{active}$. Exponential decay might result in an estimate of
$\hat{C}_\text{exp} = \hat{C} \cdot (\frac{1}{2})^{\text{requests}_\text{active}}$. The linear decay function was used in the experi-
ments presented below.
3 Simulating Disk Latency

During the course of implementation and experimentation, it became apparent that differing hardware setups made calibration of the storage constants of disk throughput and latency quite difficult. Preliminary results of using the networked filesystem in Yale’s Zoo computing cluster yielded extreme and potentially biased variation in throughput and latency that made it very hard to measure the much smaller effects of more interesting independent variables, such as the number of cold transactions or the estimate of time until data availability, on these observable performance parameters. To address this problem, a “fake disk” was constructed to test the effects of an ideal disk with known latency and throughput, without having to worry about physical layout, network usage biases, or excessive variance.

This fake disk was implemented on top of a main-memory data structure with asynchronous calls to Prefetch and Unfetch registering events, which were pairs of a data item and a scheduled time, to respective synchronized queues with a delay corresponding to pre-specified read and write latencies. Upon initialization of the database, two additional threads, called Prefetcher and Unfetcher, synchronously read items off their respective queues and execute either a read or a write to the fake disk as soon as the scheduled timestamp is reached. After performing the reads and writes, the Prefetcher and Unfetcher threads sleep for \( \frac{1}{\text{throughput}_{\text{read}} \text{(reads/second)}} \) and \( \frac{1}{\text{throughput}_{\text{write}} \text{(writes/second)}} \) seconds, respectively. Thus, the baseline estimate of \( \hat{C} = \max \{ \frac{1}{\text{throughput}_{\text{read}} \text{ latency}_{\text{read}}}, \frac{1}{\text{throughput}_{\text{write}} \text{ latency}_{\text{write}}} \} \) holds as a reasonable upper bound for how long a fresh request to a cold data item should be delayed until it is available in main memory. For simplicity in experiments, we generally presume that \( \text{throughput}_{\text{read}} = \text{throughput}_{\text{write}} \) and \( \text{latency}_{\text{read}} = \text{latency}_{\text{write}} \). Because of the latching mechanisms in place in the storage layer, however, writes to disk happen asynchronously and only result in an observable effect on throughput in extreme cases that are not considered in this paper.

4 Microbenchmark

Much as in the disk section of the Calvin paper, the experiments in this paper use a TPC-C inspired Microbenchmark. This benchmark is essentially TPC-C’s New Order transaction generator with reduced overhead (key names for data items being integers instead of strings, for example) and finer controls on workload. Much as in the example in Section 1, Microbenchmark generates transactions that touch a set of, in this case, 10 data items. In the single node case we are interested in, Microbenchmark selects one of these ten items randomly from a set of “hot” data items and the rest from a much larger pre-generated set. This benchmark, in particular, allows for fine-grained control of contention. If the size of the set of “hot” data items is small, for example of size one, the situation is analogous to the example presented in section one where no two transactions can execute simultaneously. On the other hand, if the set
of “hot” data is large, it is quite unlikely that any two concurrent transactions will touch the same set of data. The experiments presented in this paper utilize a “hot” set of size 1000, which is a relatively low level of contention.

5 Performance Data

This section provides a description of various experimental setups and the resulting data from those experiments.

5.1 Proportion of Cold Data

In these experiments, we examine how the proportion of cold reads in the a priori memory set affects throughput, average latency. These experiments were run with truthful estimations of wait times from Prefetch on one million records. The disk latency of the simulated disk was 0.05 seconds, while the throughput was set to 1000 reads/writes per second. The data here can be used to infer the relative importance of keeping track of an accurate “hot” set of data items.

Figure 1: We see that latency grows rapidly with the number of cold records in the original memory set. The actual growth pattern is somewhat strangely shaped since latency between subsequent transactions are correlated between threads. A cold transaction, for example, might block progress on any of the other threads that require a lock that the cold transaction has yet to release.
Figure 2: Throughput exhibits an initial rapid decline, but the throughput levels off once the majority of transactions are touching cold data items. Since the Microbenchmark transactions touch 10 data items, when 10% of the data items are cold, simplifying to the case where the 10 data items are chosen uniformly at random, the proportion of cold transactions is \( P = 1 - .9^{10} = 0.65 \). Since a transaction must hold locks until all of its data has been written, the additional cost for a transaction with more than one cold data item is not nearly as high as the cost incurred by the first cold data item.

5.2 Accurate Prediction of Delay

In these experiments, we examine how the accuracy of the estimate of time to availability in main memory returned by Prefetch affects the throughput. These experiments were run with similar input parameters to the previous subsection, except with less-than-truthful responses from Prefetch, on one million records. The disk latency of the simulated disk was set to 0.1 seconds, while the throughput was reduced to 1000 reads/writes per second. We will be using the default case where the number of cold data items is pegged at 1%. In the Calvin paper it was hypothesized that the lack of obvious relationship between the accuracy of the estimate and the throughput was due to the disk layer which interacted with a networked file system. Using the tunable disk simulator, we are able to validate this hypothesis, since here a modest but apparent relationship can be observed.
Figure 3: The horizontal axis reflects the actual delay returned by Prefetch as a percentage of the truthful estimate. The vertical axis represents the number of transactions per second attained with the level of delay reported. As might be expected, higher delays modestly increase throughput since it is more likely that originally cold data items will have made their way into main memory prior to the transactions acquiring locks on those data items.

6 Conclusions and Research Experience

The storage layer presented in this paper differs noticeably from that presented in the initial project proposal. The proposal was made under certain false presumptions of deterministic guarantees in Calvin, so the initial storage system that acted as an interface for various cache algorithms, such as LRU and Clock, was not viable in the context of multithreaded requests. Synchronization of data structures proved to be too costly, so a fundamental redesign was undertaken prior to the November 1 submission of the Calvin follow up paper to Sigmod. Instead of a cache algorithm that intended to maximize main memory utilization, what was found to be more useful was a storage layer that facilitated the identification of hot and cold data items so that Calvin’s sequencer might deterministically order transactions so that cold transactions were scheduled only when their required data was made available in main memory. Thus, this work focuses on the effects of the two obvious inputs into such a system: keeping track of commonly accessed items and accurately predicting the time costs of disk access. While only rudimentary methods for these two concerns have been implemented in Calvin so far, this work offers a solid set of experimental results.
toward which future implementations might aim.

7 Acknowledgments

In addition to the advisorship of Professor Abadi, my experience with this project would not have been so educational were it not for the hard work of Alex Thomson, Thaddeus Diamond, Kun Ren, Shu-chun Weng on the Calvin Deterministic Database Project.