Low Overhead Concurrency Control for Partitioned Main Memory Databases

Evan P. C. Jones
MIT CSAIL
Cambridge, MA, USA
evanj@csail.mit.edu

Daniel J. Abadi
Yale University
New Haven, CT, USA
dna@cs.yale.edu

Samuel Madden
MIT CSAIL
Cambridge, MA, USA
madden@csail.mit.edu

ABSTRACT

Database partitioning is a technique for improving the performance of distributed OLTP databases, since "single partition" transactions that access data on one partition do not need coordination with other partitions. For workloads that are amenable to partitioning, some argue that transactions should be executed serially on each partition without any concurrency at all. This strategy makes sense for a main memory database where there are no disk or user stalls, since the CPU can be fully utilized and the overhead of traditional concurrency control, such as two-phase locking, can be avoided. Unfortunately, many OLTP applications have some transactions which access multiple partitions. This introduces network stalls in order to coordinate distributed transactions, which will limit the performance of a database that does not allow concurrency.

In this paper, we compare two low overhead concurrency control schemes that allow partitions to work on other transactions during network stalls, yet have little cost in the common case when concurrency is not needed. The first is a light-weight locking scheme, and the second is an even lighter-weight type of speculative concurrency control that avoids the overhead of tracking reads and writes, but sometimes performs work that eventually must be undone. We quantify the range of workloads over which each technique is beneficial, showing that speculative concurrency control generally outperforms locking as long as there are few aborts or few distributed transactions that involve multiple rounds of communication. On a modified TPC-C benchmark, speculative concurrency control can improve throughput relative to the other schemes by up to a factor of two.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems

General Terms

Experimentation, Performance

SIGMOD'10, June 6–11, 2010, Indianapolis, Indiana, USA.

1. INTRODUCTION

Databases rely on concurrency control to provide the illusion of sequential execution of transactions, while actually executing multiple transactions simultaneously. However, several research papers suggest that for some specialized databases, concurrency control may not be necessary [11, 12, 27, 26]. In particular, if the data fits in main memory and the workload consists of transactions that can be executed without user stalls, then there is no need to execute transactions concurrently to fully utilize a single CPU. Instead, each transaction can be completely executed before servicing the next. Previous work studying the Shore database system [14] measured the overhead imposed by locks, latches, and undo buffer maintenance required by multi-threaded concurrency control to be 42% of the CPU instructions executed on part of the TPC-C benchmark [1]. This suggests that removing concurrency control could lead to a significant performance improvement.

Databases also use concurrency to utilize multiple CPUs, by assigning transactions to different threads. However, recent work by Pandis et al. [20] shows that this approach does not scale to large numbers of processing cores; instead they propose a data-oriented approach, where each thread "owns" a *partition* of the data and transactions are passed to different threads depending on what data they access. Similarly, H-Store [26] also has one thread per data partition. In these systems, there is only one thread that can access a data item, and traditional concurrency control is not necessary.

Data partitioning is also used for shared-nothing systems. Data is divided across n database servers, and transactions are routed to the partitions that contain the data they need to access. This approach is often used to improve database performance. Some applications are "perfectly partitionable," such that every transaction can be executed in its entirety at a single partition. In such a case, if the data is stored in main memory, each transaction can run without concurrency control, running to completion at the corresponding partition. However, many applications have some transactions that span multiple partitions. For these transactions, some form of concurrency control is needed. since Network stalls are necessary to coordinate the execution of these multi-partition transactions, and each processor must wait if there is no concurrency control.

This paper focuses on these "imperfectly partitionable" applications. For these workloads, there are some multi-partition transactions that impose network stalls, and some single-partition transactions that can run to completion without disk, user, or network stalls. The goal is to use a concur-

rency control scheme that allows a processor to do something useful during a network stall, while not significantly hurting the performance of single-partition transactions.

We study two such schemes. The first is a speculative approach which works as follows: when a multi-partition transaction t has completed on one partition, but is waiting for other participating partitions to complete, additional *speculative* transactions are executed. However, they are not committed until t commits. Speculation does not hold locks or keep track of read/write sets—instead, it assumes that a speculative transaction conflicts with any transaction t with which it ran concurrently. For this reason, if t aborts, any speculative transactions must be re-executed.

The second scheme is based on two-phase locking. When there are no active multi-partition transactions, single partition transactions are efficiently executed without acquiring locks. However, any active multi-partition transactions cause all transactions to lock and unlock data upon access, as with standard two-phase locking.

We compare the strengths and limitations of these two concurrency control schemes for main-memory partitioned databases, as well as a simple blocking scheme, where only one transaction runs at a time. Our results show that the simple blocking technique only works well if there are very few multi-partition transactions. Speculative execution performs best for workloads composed of single partition transactions and multi-partition transactions that perform one unit of work at each participant. In particular, our experimental results show that speculation can double throughput on a modified TPC-C workload. However, for abort-heavy workloads, speculation performs poorly, because it cascades aborts of concurrent transactions. Locking performs best on workloads where there are many multi-partition transactions, especially if participants must perform multiple rounds of work, with network communication between them. However, locking performs worse with increasing data conflicts, and especially suffers from distributed deadlock.

2. ASSUMPTIONS ON SYSTEM DESIGN

Our concurrency control schemes are designed for a partitioned main-memory database system similar to H-Store [26]. This section gives an overview of the relevant aspects of the H-Store design.

Traditional concurrency control comprises nearly half the CPU instructions executed by a database engine [14]. This suggests that avoiding concurrency control can improve throughput substantially. H-Store was therefore designed explicitly to avoid this overhead.

2.1 Transactions as Stored Procedures

H-Store only supports executing transactions that have been pre-declared as stored procedures. Each stored procedure invocation is a single transaction that must either abort or commit before returning results to the client. Eliminating ad-hoc transactions removes client stalls, reducing the need for concurrency.

2.2 No Disk

Today, relatively low-end 1U servers can have up to 128 GB of RAM, which gives a data center rack of 40 servers an aggregate RAM capacity of over 5 TB. Thus, a modest amount of hardware can hold all but the largest OLTP databases in memory, eliminating the need for disk access during normal operation. This eliminates disk stalls, which further reduces the need for concurrent transactions.

Traditional databases rely on disk to provide durability. However, mission critical OLTP applications need high availability which means they use replicated systems. H-Store takes advantage of replication for durability, as well as high availability. A transaction is committed when it has been received by $k > 1$ replicas, where k is a tuning parameter.

2.3 Partitioning

Without client and disk stalls, H-Store simply executes transactions from beginning to completion in a single thread. To take advantage of multiple physical machines and multiple CPUs, the data must be divided into separate partitions. Each partition executes transactions independently. The challenge becomes dividing the application's data so that each transaction only accesses one partition. For many OLTP applications, partitioning the application manually is straightforward. For example, the TPC-C OLTP benchmark can be partitioned by warehouse so an average of 89% of the transactions access a single partition [26]. There is evidence that developers already do this to scale their applications [22, 23, 25], and academic research provides some approaches for automatically selecting a good partitioning key [4, 21, 28]. However, unless the partitioning scheme is 100% effective in making all transactions only access a single partition, then coordination across multiple partitions for multi-partition transactions cause network stalls and executing a transaction to completion without stalls is not possible. In this paper, we focus on what the system should do in this case.

3. EXECUTING TRANSACTIONS

In this section, we describe how our prototype executes transactions. We begin by describing the components of our system and the execution model. We then discuss how single partition and multi-partition transactions are executed.

3.1 System Components

The system is composed of three types of processes, shown in Figure 1. First, the data is stored in *partitions*, a single process that stores a portion of the data in memory, and executes stored procedures using a single thread. For each partition, there is a primary process and $k-1$ backup processes, which ensures that data survives $k-1$ failures. Second, a single process called the *central coordinator* is used to coordinate all distributed transactions. This ensures that distributed transactions are globally ordered, and is described in Section 3.3. Finally, the *client* processes are end-user applications that issue transactions to the system in the form of stored procedure invocations. When the client library connects to the database, it downloads the part of the system catalog describing the available stored procedures, network addresses for the partitions, and how data is distributed. This permits the client library to direct transactions to the appropriate processes.

Transactions are written as stored procedures, composed of deterministic code interleaved database operations. The client invokes transactions by sending a stored procedure invocation to the system. The system distributes the work across the partitions. In our prototype, the mapping of work to partitions is done manually, but we are working on a query planner to do this automatically.

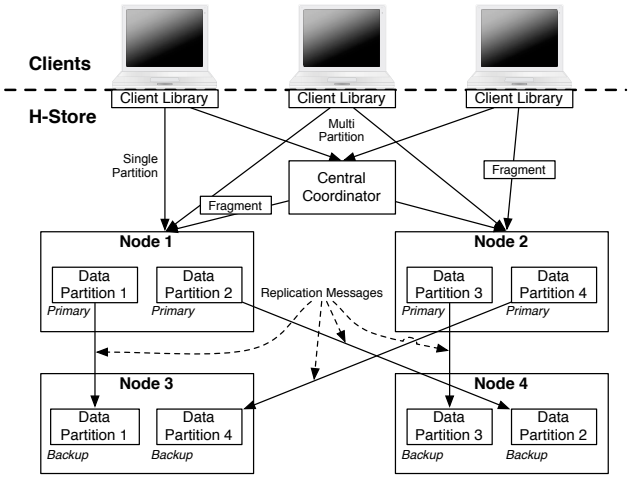

Figure 1: System Architecture

Each transaction is divided into *fragments*. A fragment is a unit of work that can be executed at exactly one partition. It can be some mixture of user code and database operations. A single partition transaction, for example, is composed of one fragment containing the entire transaction. A multi-partition transaction is composed of multiple fragments with data dependencies between them.

3.2 Single Partition Transactions

When a client determines that a request is a single partition transaction, it forwards it to the primary partition responsible for the data. The primary uses a typical primary/backup replication protocol to ensure durability. In the failure free case, the primary reads the request from the network and sends a copy to the backups. While waiting for acknowledgments, the primary executes the transaction. Since it is a single partition transaction, it does not block. When all acknowledgments from the backups are received, the result of the transaction is sent to the client. This protocol ensures the transaction is durable, as long as at least one replica survives a failure.

No concurrency control is needed to execute single partition transactions. In most cases, the system executes these transactions without recording undo information, resulting in very low overhead. This is possible because transactions are annotated to indicate if a user abort may occur. For transactions that have no possibility of a user abort, concurrency control schemes that guarantee that deadlock will not occur (see below) do not keep an undo log. Otherwise, the system maintains an in-memory undo buffer that is discarded when the transaction commits.

3.3 Multi-Partition Transactions

In general, multi-partition transaction can have arbitrary data dependencies between transaction fragments. For example, a transaction may need to read a value stored at partition P_1, in order to update a value at partition P_2.

To ensure multi-partition transactions execute in a serializable order without deadlocks, we forward them through the central coordinator, which assigns them a global order. Although this is a straightforward approach, the central coordinator limits the rate of multi-partition transactions. To handle more multi-partition transactions, multiple coordinators must be used. Previous work has investigated how to globally order transactions with multiple coordinators, for example by using loosely synchronized clocks [2]. We leave selecting the best alternative to future work, and only evaluate a single coordinator system in this paper.

The central coordinator divides the transaction into fragments and sends them to the partitions. When responses are received, the coordinator executes application code to determine how to continue the transaction, which may require sending more fragments. Each partition executes fragments for a given transaction sequentially.

Multi-partition transactions are executed using an undo buffer, and use two-phase commit (2PC) to decide the outcome. This allows each partition of the database to fail independently. If the transaction causes one partition to crash or the network splits during execution, other participants are able to recover and continue processing transactions that do not depend on the failed partition. Without undo information, the system would need to block until the failure is repaired.

The coordinator piggybacks the 2PC "prepare" message with the last fragment of a transaction. When the primary receives the final fragment, it sends all the fragments of the transaction to the backups and waits for acknowledgments before sending the final results to the coordinator. This is equivalent to forcing the participant's 2PC vote to disk. Finally, when the coordinator has all the votes from the participants, it completes the transaction by sending a "commit" message to the partitions and returning the final result to the application.

When executing multi-partition transactions, network stalls can occur while waiting for data from other partitions. This idle time can introduce a performance bottleneck, even if multi-partition transactions only comprise a small fraction of the workload. On our experimental systems, described in Section 5, the minimum network round-trip time between two machines connected to the same gigabit Ethernet switch was measured using `ping` to be approximately 40 μs. The average CPU time for a TPC-C transaction in our system is 26 μs. Thus, while waiting for a network acknowledgment, the partition could execute at least two single-partition transactions. Some form of concurrency control is needed to permit the engine to do useful work while otherwise idle. The challenge is to not reduce the efficiency of simple single partition transactions. The next section describes two concurrency control schemes we have developed to address this issue.

4. CONCURRENCY CONTROL SCHEMES

4.1 Blocking

The simplest scheme for handling multi-partition transactions is to block until they complete. When the partition receives the first fragment of a multi-partition transaction, it is executed and the results are returned. All other transactions are queued. When subsequent fragments of the active transaction are received, they are processed in order. After the transaction is committed or aborted, the queued transactions are processed. In effect, this system assumes that all transactions conflict, and thus can only execute one at a time. Pseudocode describing this approach is shown in Figure 2.

```
Transaction Fragment Arrives
if no active transactions:
    if single partition:
        execute fragment without undo buffer
        commit
    else:
        execute fragment with undo buffer
else if fragment continues active multi-partition transaction:
    continue transaction with fragment
else:
    queue fragment

Commit/Abort Decision Arrives
if abort:
    undo aborted transaction
execute queued transactions
```

Figure 2: Blocking Pseudocode

4.2 Speculative Execution

This concurrency control scheme speculatively executes queued transactions when the blocking scheme would otherwise be idle. More precisely, when the last fragment of a multi-partition transaction has been executed, the partition must wait to learn if the transaction commits or aborts. In the majority of cases, it will commit. Thus, we can execute queued transactions while waiting for 2PC to finish. The results of these speculatively executed transactions cannot be sent outside the database, since if the first transaction aborts, the results of the speculatively executed transactions may be incorrect. Undo information must be recorded for speculative transactions, so they can be rolled back if needed. If the first transaction commits, all speculatively executed transactions are immediately committed. Thus, speculation hides the latency of 2PC by performing useful work instead of blocking.

Speculation produces serializable execution schedules because once a transaction t has finished all of its reads and writes and is simply waiting to learn if t commits or aborts, we can be guaranteed that any transaction t^* which makes use of t's results on a partition p will be serialized after t on p. However, t^* may have read data that t wrote, such that we will have to abort t^* to avoid exposing t's dirty (rolled-back) data in the event that t aborts.

The simplest form of speculation is when the speculatively executed transactions are single partition transactions, so we consider that case first.

4.2.1 Speculating Single Partition Transactions

Each partition maintains a queue of unexecuted transactions and a queue of uncommitted transactions. The head of the uncommitted transaction queue is always a non-speculative transaction. After a partition has executed the last fragment of a multi-partition transaction, it executes additional transactions speculatively from the unexecuted queue, adding them to the end of the uncommitted transaction queue. An undo buffer is recorded for each transaction. If the non-speculative transaction aborts, each transaction is removed from the tail of the uncommitted transaction queue, undone, then pushed onto the head of the unexecuted transaction queue to be re-executed. If it commits, transactions are dequeued from the head of the queue and results are sent. When the uncommitted queue is empty, the system resumes executing transactions non-speculatively (for transactions that cannot abort, execution can proceed without the extra overhead of recording undo information).

As an example of how this works, consider a two-partition database (P_1 and P_2) with two integer records, x on P_1 and y on P_2. Initially, $x = 5$ and $y = 17$. Suppose the system executes three transactions, A, B_1, and B_2, in order. Transaction A is a multi-partition transaction that swaps the value of x and y. Transactions B_1 and B_2 are single partition transactions on P_1 that increment x and return its value.

Both partitions begin by executing the first fragments of transaction A, which read x and y. P_1 and P_2 execute the fragments and return the values to the coordinator. Since A is not finished, speculation of B_1 and B_2 cannot begin. If it did, the result for transaction B_1 would be $x = 6$, which is incorrect. The coordinator sends the final fragments, which write $x = 17$ on partition P_1, and $y = 5$ on partition P_2. After finishing these fragments, the partitions send their "ready to commit" acknowledgment to the coordinator, and wait for the decision. Because A is finished, speculation can begin. Transactions B_1 and B_2 are executed with undo buffers and the results are queued. If transaction A were to abort at this point, both B_2 and B_1 would be undone and re-executed. When P_1 is informed that transaction A has committed, it sends the results for B_1 and B_2 to the clients and discards their undo buffers.

This describes purely *local speculation*, where speculative results are buffered inside a partition and not exposed until they are known to be correct. Only the first fragment of a multi-partition transaction can be speculated in this way, since the results cannot be exposed in case they must be undone. However, we can speculate many multi-partition transactions if the coordinator is aware of the speculation, as described in the next section.

4.2.2 Speculating Multi-Partition Transactions

Consider the same example, except now the system executes A, B_1, then a new multi-partition transaction C, which increments both x and y, then finally B_2. The system executes transaction A as before, and partition P_1 speculates B_1. Partition P_2 can speculate its fragment of C, computing $y = 6$. With local speculation described above, it must wait for A to commit before returning this result to the coordinator, since if A aborts, the result will be incorrect. However, because A and C have the same coordinator, partition P_2 can return the result for its fragment of C, with an indication that it depends on transaction A. Similarly, partition P_1 can speculate its fragment of C, computing and returning $x = 17$, along with an indication that this depends on A. It also speculates B_2, computing $x = 18$. However, it cannot send this result, as it would go directly to a client that is not aware of the previous transactions, because single partition transactions do not go through the central coordinator. Once the multi-partition transactions have committed and the uncommitted transaction queue is empty, the partitions can resume non-speculative execution.

After the coordinator commits A, it examines the results for C. Since C depends on A, and A committed, the speculative results are correct and C can commit. If A had aborted, the coordinator would send an abort message for A to P_1 and P_2, then discard the incorrect results it received

for C. As before, the abort message would cause the partitions to undo the transactions on the uncommitted transaction queues. Transaction A would be aborted, but the other transactions would be placed back on the unexecuted queue and re-executed in the same order. The partitions would then resend results for C. The resent results would not depend on previous transactions, so the coordinator could handle it normally. The pseudocode for this scheme is shown in Figure 3.

This scheme allows a sequence of multi-partition transactions, each composed of a single fragment at each partition to be executed without blocking, assuming they all commit. We call such transactions *simple multi-partition transactions*. Transactions of this form are quite common. For example, if there is a table that is mostly used for reads, it may be beneficial to replicate it across all partitions. Reads can then be performed locally, as part of a single partition transaction. Occasional updates of this table execute as a simple multi- partition transaction across all partitions. Another example is if a table is partitioned on column x, and records are accessed based on the value of column y. This can be executed by attempting the access on all partitions of the table, which is also a simple multi-partition transaction. As a third example, all distributed transactions in TPC-C are simple multi-partition transactions [26]. Thus, this optimization extends the types of workloads where speculation is beneficial.

There are a few limitations to speculation. First, since speculation can only be used after executing the last fragment of a transaction, it is less effective for transactions that require multiple fragments at one partition.

Second, multi-partition speculation can only be used when the multi-partition transactions come from the same coordinator. This is necessary so the coordinator is aware of the outcome of the earlier transactions and can cascade aborts as required. However, a single coordinator can become a bottleneck, as discussed in Section 3.3. Thus, when using multiple coordinators, each coordinator must dispatch transactions in batches to take advantage of this optimization. This requires delaying transactions, and tuning the number of coordinators to match the workload.

Speculation has the advantage that it does not require locks or read/write set tracking, and thus requires less overhead than traditional concurrency control. A disadvantage is that it assumes that all transactions conflict, and therefore occasionally unnecessarily aborts transactions.

Transaction Fragment Arrives

```
if no active transaction:
  if single partition:
    execute fragment without undo buffer
    commit
  else:
    execute fragment with undo buffer
else if fragment continues active multi-partition transaction:
  continue transaction by executing fragment
  if transaction is finished locally:
    speculate queued transactions
else if tail transaction in uncommitted queue is finished locally:
  execute fragment with undo buffer
  same_coordinator ← false
  if all txns in uncommitted queue have same coordinator:
    same_coordinator ← true
  if transaction is multi-partition and same_coordinator:
    record dependency on previous multi-partition transaction
    send speculative results
else:
  queue fragment
```

Commit/Abort Decision Arrives

```
if abort:
  undo and re-queue all speculative transactions
  undo aborted transaction
else:
  while next speculative transaction is not multi-partition:
    commit speculative transaction
    send results
execute/speculate queued transactions
```

Figure 3: Speculation Pseudocode

4.3 Locking

In the locking scheme, transactions acquire read and write locks on data items while executing, and are suspended if they make a conflicting lock request. Transactions must record undo information in order to rollback in case of deadlock. Locking permits a single partition to execute and commit non-conflicting transactions during network stalls for multi-partition transactions. The locks ensure that the results are equivalent to transaction execution in some serial order. The disadvantage is that transactions are executed with the additional overhead of acquiring locks and detecting deadlock.

We avoid this overhead where possible. When our locking system has no active transactions and receives a single partition transaction, the transaction can be executed without locks and undo information, in the same way as the blocking and speculation schemes. This works because there are no active transactions that could cause conflicts, and the transaction is guaranteed to execute to completion before the partition executes any other transaction. Thus, locks are only acquired when there are active multi-partition transactions.

Our locking scheme follows the strict two phase locking protocol. Since this is guaranteed to produce a serializable transaction order, clients send multi-partition transactions directly to the partitions, without going through the central coordinator. This is more efficient when there are no lock conflicts, as it reduces network latency and eliminates an extra process from the system. However, it introduces the possibility of distributed deadlock. Our implementation uses cycle detection to handle local deadlocks, and timeout to handle distributed deadlock. If a cycle is found, it will prefer to kill single partition transactions to break the cycle, as that will result in less wasted work.

Since our system is motivated by systems like H-Store [26] and DORA [20], each partition runs in single-threaded mode. Therefore, our locking scheme has much lower overhead than traditional locking schemes that have to latch a centralized lock manager before manipulating the lock data structures. Our system can simply lock a data item without having to worry about another thread trying to concurrently lock the same item. The only type of concurrency we are trying to enable is *logical* concurrency where a new transaction can make progress only when the previous transaction

is blocked waiting for a network stall — *physical* concurrency cannot occur.

When a transaction is finished and ready to commit, the fragments of the transaction are sent to the backups. This includes any data received from other partitions, so the backups do not participate in distributed transactions. The backups execute the transactions in the sequential order received from the primary. This will produce the same result, as we assume transactions are deterministic. Locks are not acquired while executing the fragments at the backups, since they are not needed. Unlike typical statement-based replication, applying transactions sequentially is not a performance bottleneck because the primary is also single threaded. As with the previous schemes, once the primary has received acknowledgments from all backups, it considers the transaction to be durable and can return results to the client.

5. EXPERIMENTAL EVALUATION

In this section we explore the trade-offs between the above concurrency control schemes by comparing their performance on some microbenchmarks and a benchmark based on TPC-C. The microbenchmarks are designed to discover the important differences between the approaches, and are not necessarily presented as being representative of any particular application. Our TPC-C benchmark is intended to represent the performance of a more complete and realistic OLTP application.

Our prototype is written in C++ and runs on Linux. We used six servers with two Xeon 3.20 GHz CPUs and 2 GB of RAM. The machines are connected to a single gigabit Ethernet switch. The clients run as multiple threads on a single machine. For each test, we use 15 seconds of warm-up, followed by 60 seconds of measurement (longer tests did not yield different results). We measure the number of transactions that are completed by all clients within the measurement period. Each measurement is repeated three times. We show only the averages, as the confidence intervals are within a few percent and needlessly clutter the figures.

For the microbenchmarks, the execution engine is a simple key/value store, where keys and values are arbitrary byte strings. One transaction is supported, which reads a set of values then updates them. We use small 3 byte keys and 4 byte values to avoid complications caused by data transfer time.

For the TPC-C benchmarks, we use a custom written execution engine that executes transactions directly on data in memory. Each table is represented as either a B-Tree, a binary tree, or hash table, as appropriate.

On each benchmark we compare the three concurrency control schemes described in Section 4.

5.1 Microbenchmark

Our microbenchmark implements a simple mix of single partition and multi-partition transactions, in order to understand the impact of distributed transactions on throughput. We create a database composed of two partitions, each of which resides on a separate machine. The partitions each store half the keys. Each client issues a read/write transaction which reads and writes the value associated with 12 keys. For this test, there is no sharing (i.e., potential conflict across clients): each client writes its own set of keys. We experiment with shared data in the next section. To create a single partition transaction, a clients selects a partition at random, then accesses 12 keys on that partition. To create a multi-partition transaction, the keys are divided evenly by accessing 6 keys on each partition. To fully utilize the CPU on both partitions, we use 40 simultaneous clients. Each client issues one request, waits for the response, then issues another request.

We vary the fraction of multi-partition transactions and measure the transaction throughput. The results are shown in Figure 4. From the application's perspective, the multi-partition and single partition transactions do the same amount of work, so ideally the throughput should stay constant. However, concurrency control overhead means this is not the case. The performance for locking is linear in the range between 16% and 100% multi-partition transactions. The reason is that none of these transactions conflict, so they can all be executed simultaneously. The slight downward slope is due to the fact that multi-partition transactions have additional communication overhead, and thus the performance degrades slightly as their fraction of the workload increases.

The most interesting part of the locking results is between 0% and 16% multi-partition transactions. As expected, the performance of locking is very close to the other schemes at 0% multi-partition transactions, due to our optimization where we do not set locks when there are no multi-partition transactions running. The throughput matches speculation and blocking at 0%, then decreases rapidly until 16% multi-partition transactions, when there is usually at least one multi-partition transaction in progress, and therefore nearly all transactions are executed with locks.

The performance of blocking in this microbenchmark degrades steeply, so that it never outperforms locking on this low-contention workload. The reason is that the advantage of executing transactions without acquiring locks is outweighed by the idle time caused by waiting for two-phase commit to complete. Our locking implementation executes many transactions without locks when there are few multi-partition transactions, so there is no advantage to blocking. If we force locks to always be acquired, blocking does outperform locking from 0% to 6% multi-partition transactions.

With fewer than 50% multi-partition transactions, the throughput of speculation parallels locking, except with approximately 10% higher throughput. Since this workload is composed of single-partition and simple multi-partition transactions, the single coordinator can speculate all of them. This results in concurrent execution, like locking, but without the overhead of tracking locks. Past 50%, speculation's performance begins to drop. This is the point where the central coordinator uses 100% of the CPU and cannot handle more messages. To scale past this point, we would need to implement distributed transaction ordering, as described in Section 4.2.2.

For this particular experiment, blocking is always worse than speculation and locking. Speculation outperforms locking by up to 13%, before the central coordinator becomes a bottleneck. With the bottleneck of the central coordinator, locking outperforms speculation by 45%.

5.2 Conflicts

The performance of locking depends on conflicts between transactions. When there are no conflicts, transactions execute concurrently. However, when there are conflicts, there is additional overhead to suspend and resume execution. To

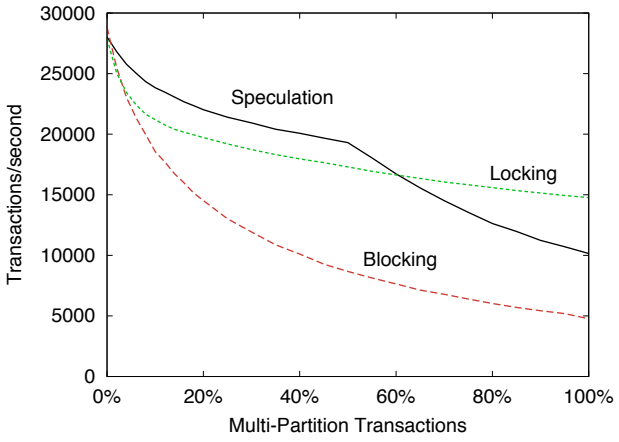

Figure 4: Microbenchmark Without Conflicts

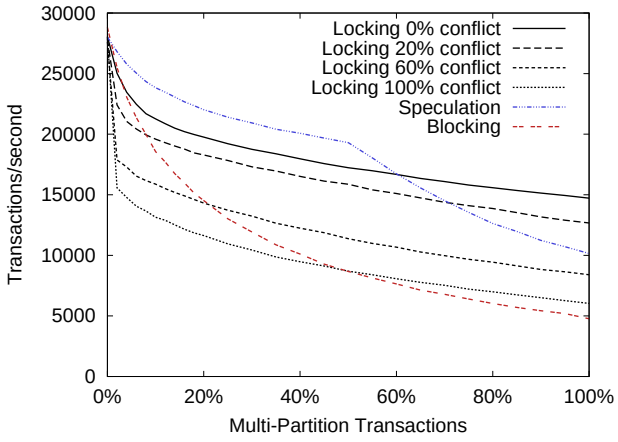

Figure 5: Microbenchmark With Conflicts

investigate the impact of conflicts, we change the pattern of keys that clients access. When issuing single partition transactions, the first client only issues transactions to the first partition, and the second client only issues transactions to the second partition, rather than selecting the destination partition at random. This means the first two clients' keys on their respective partitions are nearly always being written. To cause conflicts, the other clients write one of these "conflict" keys with probability p, or write their own private keys with probability $1 - p$. Such transactions will have a very high probability of attempting to update the key at a same time as the first two clients. Increasing p results in more conflicts. Deadlocks are not possible in this workload, allowing us to avoid the performance impact of implementation dependent deadlock resolution policies.

The results in Figure 5 show a single line for speculation and blocking, as their throughput does not change with the conflict probability. This is because they assume that all transactions conflict. The performance of locking, on the other hand, degrades as conflict rate increases. Rather than the nearly straight line as before, with conflicts the throughput falls off steeply as the percentage of multi-partition transactions increases. This is because as the conflict rate increases, locking behaves more like blocking. Locking still outperforms blocking when there are many multi-partition transactions because in this workload, each transaction only conflicts at one of the partitions, so it still performs some work concurrently. However, these results do suggest that if conflicts between transactions are common, the advantage of avoiding concurrency control is larger. In this experiment, speculation is up to 2.5 times faster than locking.

5.3 Aborts

Speculation assumes that transactions will commit. When a transaction is aborted, the speculatively executed transactions must be undone and re-executed, wasting CPU time. To understand the effects of re-execution, we select transactions to be aborted at random with probability p. When a multi-partition transaction is selected, only one partition will abort locally. The other partition will be aborted during two-phase commit. Aborted transactions are somewhat cheaper to execute than normal transactions, since the abort happens at the beginning of execution. They are identical in all other respects, (e.g., network message length).

The results for this experiment are shown in Figure 6. The cost of an abort is variable, depending on how many speculatively executed transactions need to be re-executed. Thus, the 95% confidence intervals are wider for this experiment, but they are still within 5%, so we omit them for clarity. Since blocking and locking do not have cascading aborts, the abort rate does not have a significant impact, so we only show the 10% abort probability results. This has slightly higher throughput than the 0% case, since abort transactions require less CPU time.

As expected, aborts decrease the throughput of speculative execution, due to the cost of re-executing transactions. They also increase the number of messages that the central coordinator handles, causing it to saturate sooner. However, speculation still outperforms locking for up to 5% aborts, ignoring the limits of the central coordinator. With 10% aborts, speculation is nearly as bad as blocking, since some transactions are executed many times. These results suggest that if a transaction has a very high abort probability, it may be better to limit to the amount of speculation to avoid wasted work.

5.4 General Multi-Partition Transactions

When executing a multi-partition transaction that involves multiple rounds of communication, speculation can only begin when the transaction is known to have completed all its work at a given partition. This means that there must be a stall between the individual fragments of transaction. To examine the performance impact of these multi-round transactions, we changed our microbenchmark to issue a multi-partition transaction that requires two rounds of communication, instead of the simple multi-partition transaction in the original benchmark. The first round of each transaction performs the reads and returns the results to the coordinator, which then issues the writes as a second round. This performs the same amount of work as the original benchmark, but has twice as many messages.

The results are shown in Figure 7. The blocking throughput follows the same trend as before, only lower because the two round transactions take nearly twice as much time as

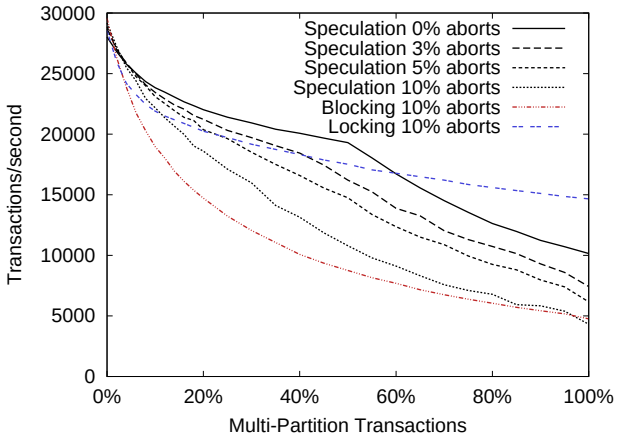

Figure 6: Microbenchmark With Aborts

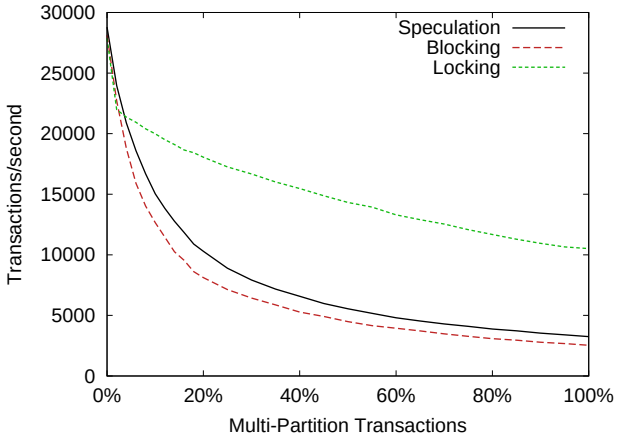

Figure 7: General Transaction Microbenchmark

the multi-partition transactions in the original benchmark. Speculation performs only slightly better, since it can only speculate the first fragment of the next multi-partition transaction once the previous one has finished. Locking is relatively unaffected by the additional round of network communication. Even though locking is generally superior for this workload, speculation does still outperform locking as long as fewer than 4% of the workload is composed of general multi-partition transactions.

5.5 TPC-C

The TPC-C benchmark models the OLTP workload of an order processing system. It is comprised of a mix of five transactions with different properties. The data size is scaled by adding warehouses, which adds a set of related records to the other tables. We partition the TPC-C database by warehouse, as described by Stonebraker et al. [26]. We replicate the items table, which is read-only, to all partitions. We vertically partition the stock table, and replicate the read-only columns across all partitions, leaving the columns that are updated in a single partition. This partitioning means 89% of the transactions access a single partition, and the others are simple multi-partition transactions.

Our implementation tries to be faithful to the specification, but there are three differences. First, we reorder the operations in the new order transaction to avoid needing an undo buffer to handle user aborts. Second, our clients have no pause time. Instead, they generate another transaction immediately after receiving the result from the previous one. This permits us to generate a high transaction rate with a small number of warehouses. Finally, we change how clients generate requests. The TPC-C specification assigns clients to a specific (warehouse, district) pair. Thus, as you add more warehouses, you add more clients. We use a fixed number of clients while changing the number of warehouses, in order to change only one variable at a time. To accommodate this, our clients generate requests for an assigned warehouse but a random district.

We ran TPC-C with the warehouses divided evenly across two partitions. In this workload, the fraction of multi-partition transactions ranges from 5.7% with 20 warehouses to 10.7% with 2 warehouses. The throughput for varying numbers of warehouses are shown in Figure 8. With this workload, blocking and speculation have relatively constant performance as the number of warehouses is increased. The performance is lowest with 2 partitions because the probability of a multi-partition transaction is highest (10.7%, versus 7.2% for 4 warehouses, and 5.7% for 20 warehouses), due to the way TPC-C new order transaction requests are generated. After 4 warehouses, the performance for blocking and speculation decrease slightly. This is due to the larger working set size and the corresponding increase in CPU cache and TLB misses. The performance for locking increases as the number of warehouses is increased because the number of conflicting transactions decreases. This is because there are fewer clients per TPC-C warehouse, and nearly every transaction modifies the warehouse and district records. This workload also has deadlocks, which leads to overhead due to deadlock detection and distributed deadlock timeouts, decreasing the performance for locking. Speculation performs the best of the three schemes because the workload's fraction of multi-partition transactions is within the region where it is the best choice. With 20 warehouses, speculation provides 9.7% higher throughput than blocking, and 63% higher throughput than locking.

5.6 TPC-C Multi-Partition Scaling

In order to examine the impact that multi-partition transactions have on a more complex workload, we scale the fraction of TPC-C transactions that span multiple partitions. We execute a workload that is composed of 100% new order transactions on 6 warehouses. We then adjust the probability that an item in the order comes from a "remote" warehouse, which is a multi-partition transaction. With TPC-C's default parameters, this probability is 0.01 (1%), which produces a multi-partition transaction 9.5% of the time. We adjust this parameter and compute the probability that a transaction is a multi-partition transaction. The throughput with this workload is shown in Figure 9.

The results for blocking and speculation are very similar to the results for the microbenchmark in Figure 4. In this experiment, the performance for locking degrades very rapidly. At 0% multi-partition transactions, it runs efficiently without acquiring locks, but with multi-partition transactions it

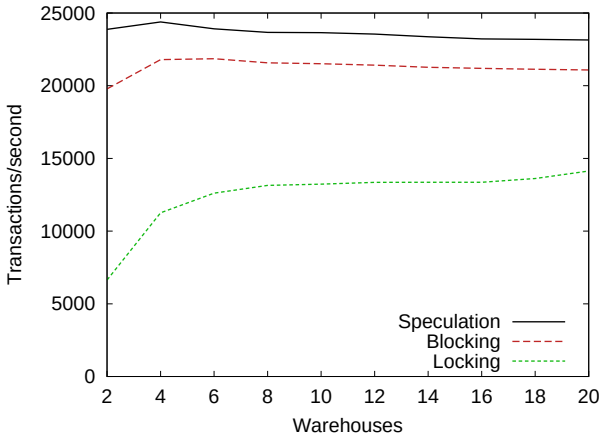

Figure 8: TPC-C Throughput Varying Warehouses

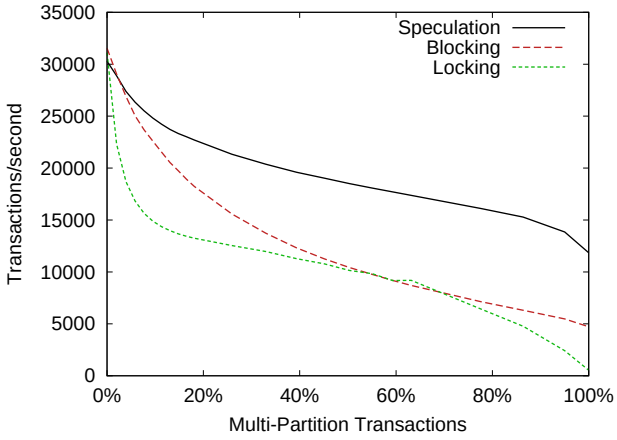

Figure 9: TPC-C 100% New Order

must acquire locks. The locking overhead is higher for TPC-C than our microbenchmark for three reasons: more locks are acquired for each transaction, the lock manager is more complex, and there are many conflicts. In particular, this workload exhibits local and distributed deadlocks, hurting throughput significantly. Again, this shows that conflicts make traditional concurrency control more expensive, increasing the benefits of simpler schemes.

Examining the output of a sampling profiler while running with a 10% multi-partition probability shows that 34% of the execution time is spent in the lock implementation. Approximately 12% of the time is spent managing the lock table, 14% is spent acquiring locks, and 6% is spent releasing locks. While our locking implementation certainly has room for optimization, this is similar to what was previously measured for Shore, where 16% of the CPU instructions could be attributed to locking [14].

5.7 Summary

Our results show that the properties of the workload determine the best concurrency control mechanism. Speculation performs substantially better than locking or blocking for multi-partition transactions that require only a single round of communication and when a low percentage of transactions abort. Our low overhead locking technique is best when there are many transactions with multiple rounds of communication. Table 1 shows which scheme is best, depending on the workload; we imagine that a query executor might record statistics at runtime and use a model like that presented in Section 6 below to make the best choice.

		Few Aborts		*Many Aborts*	
		Few Conflicts	*Many Conflicts*	*Few Conflicts*	*Many Conflicts*
Few multi-round xactions	*Many multi-partition xactions*	Speculation	Speculation	Locking	Locking or Speculation
	Few multi-partition xactions	Speculation	Speculation	Blocking or Locking	Blocking
Many multi-round xactions		Locking	Locking	Locking	Locking

Table 1: Summary of best concurrency control scheme for different situations. Speculation is preferred when there are few multi-round (general) transactions and few aborts.

Optimistic concurrency control (OCC) is another "standard" concurrency control algorithm. It requires tracking each item that is read and written, and aborts transactions during a validation phase if there were conflicts. Intuitively, we expect the performance for OCC to be similar to that of locking. This is because, unlike traditional locking implementations that need complex lock managers and careful latching to avoid problems inherent in physical concurrency, our locking scheme can be much lighter-weight, since each partition runs single-threaded (i.e., we only have to worry about the logical concurrency). Hence, our locking implementation involves little more than keeping track of the read/write sets of a transaction — which OCC also must do. Consequently, OCC's primary advantage over locking is eliminated. We have run some initial results that verify this hypothesis, and plan to explore the trade-offs between OCC and other concurrency control methods and our speculation schemes as future work.

6. ANALYTICAL MODEL

To improve our understanding of the concurrency control schemes, we analyze the expected performance for the multi-partition scaling experiment from Section 5.1. This model predicts the performance of the three schemes in terms of just a few parameters (which would be useful in a query planner, for example), and allows us to explore the sensitivity to workload characteristics (such as the CPU cost per transaction or the network latency). To simplify the analysis, we ignore replication.

Consider a database divided into two partitions, P_1 and P_2. The workload consists of two transactions. The first is a single partition transaction that accesses only P_1 or P_2, chosen uniformly at random. The second is a multi-partition transaction that accesses both partitions. There are no data dependencies, and therefore only a single round of communication is required. In other words, the coordinator simply sends two fragments out, one to each partition, waits

for the response, then sends the commit or abort decision. Each multi-partition transaction accesses half as much data in each partition, but the total amount of data accessed is equal to one single partition transaction. To provide the best case performance for locking, none of the transactions conflict. We are interested in the throughput as the fraction of multi-partition transactions, f, in the workload is increased.

6.1 Blocking

We begin by analyzing the blocking scheme. Here, a single partition transaction executes for t_{sp} seconds on one partition. A multi-partition transaction executes for t_{mp} on both partitions, including the time to complete the two-phase commit. If there are N transactions to execute, then there are Nf multi-partition transactions and $\frac{1}{2}N(1-f)$ single partition transactions to execute at each partition. The factor of $\frac{1}{2}$ arises because the single partition transactions are distributed evenly between the two partitions. Therefore the time it takes to execute the transactions and the system throughput are given by the following equations:

$$\text{time} = Nft_{mp} + \frac{N(1-f)}{2}t_{sp}$$
$$\text{throughput} = \frac{N}{\text{time}} = \frac{2}{2ft_{mp} + (1-f)t_{sp}}$$

Effectively, the time to execute N transactions is a weighted average between the times for a pure single partition workload and a pure multi-partition workload. As f increases, the throughput will decrease from $\frac{2}{t_{sp}}$ to $\frac{1}{t_{mp}}$. Since $t_{mp} > t_{sp}$, the throughput will decrease rapidly with even a small fraction of multi-partition transactions.

6.2 Speculation

We first consider the local speculation scheme described in Section 4.2.1. For speculation, we need to know the amount of time that each partition is idle during a multi-partition transaction. If the CPU time consumed by a multi-partition transaction at one partition is t_{mpC}, then the network stall time is $t_{mpN} = t_{mp} - t_{mpC}$. Since we can overlap the execution of the next multi-partition transaction with the stall, the limiting time when executing a pure multi-partition transaction workload is $t_{mpL} = \max(t_{mpN}, t_{mpC})$, and the time that the CPU is idle is $t_{mpI} = \max(t_{mpN}, t_{mpC}) - t_{mpC}$.

Assume that the time to speculatively execute a single partition transaction is t_{spS}. During a multi-partition transaction's idle time, each partition can execute a maximum of $\frac{t_{mpI}}{t_{spS}}$ single partition transactions. When the system executes Nf multi-partition transactions, each partition executes $\frac{N(1-f)}{2}$ single partition transactions. Thus, on average each multi-partition transaction is separated by $\frac{(1-f)}{2f}$ single partition transactions. Therefore, for each multi-partition transaction, the number of single partition transactions that each partition can speculate is given by:

$$N_{hidden} = \min\left(\frac{1-f}{2f}, \frac{t_{mpI}}{t_{spS}}\right)$$

Therefore, the time to execute N transactions and the resulting throughput are:

$$\text{time} = Nft_{mpL} + (N(1-f) - 2NfN_{hidden})\frac{t_{sp}}{2}$$
$$\text{throughput} = \frac{2}{2ft_{mpL} + ((1-f) - 2fN_{hidden})t_{sp}}$$

In our specific scenario and system, $t_{mpN} > t_{mpC}$, so we can simplify the equations:

$$N_{hidden} = \min\left(\frac{1-f}{2f}, \frac{t_{mp} - 2t_{mpC}}{t_{spS}}\right)$$
$$\text{throughput} = \frac{2}{2f(t_{mp} - t_{mpC}) + ((1-f) - 2fN_{hidden})t_{sp}}$$

The minimum function produces two different behaviors. If $\frac{1-f}{2f} \geq \frac{t_{mpI}}{t_{spS}}$, then the idle time can be completely utilized. Otherwise, the system does not have enough single partition transactions to completely hide the network stall, and the throughput will drop rapidly as f increases past $\frac{t_{spS}}{2t_{mpI}+t_{spS}}$.

6.2.1 Speculating Multi-Partition Transactions

This model can be extended to include speculating multi-partition transactions, as described in Section 4.2.2. The previous derivation for execution time assumes that one multi-partition transaction completes every t_{mpL} seconds, which includes the network stall time. When multi-partition transactions can be speculated, this restriction is removed. Instead, we must compute the CPU time to execute multi-partition transactions, and speculative and non-speculative single partition transactions. The previous model computed the number of speculative single partition transactions per multi-partition transaction, N_{hidden}. We can compute the time for multi-partition transactions and speculative single partition transactions as $t_{period} = t_{mpC} + N_{hidden}t_{spS}$. This time replaces t_{mpL} in the previous model, and thus the throughput becomes:

$$\text{throughput} = \frac{2}{2ft_{period} + ((1-f) - 2fN_{hidden})t_{sp}}$$

6.3 Locking

Since the workload is composed of non-conflicting transactions, locking has no stall time. However, we must account for the overhead of tracking locks. We define l to be the fraction of additional time that a transaction takes to execute when locking is enabled. Since locking always requires undo buffers, we use t_{spS} to account for the single partition execution time. Furthermore, the overhead of two-phase commit must be added, so for multi-partition transactions we use t_{mpC}. The time to execute N transactions, and the resulting throughput are given by:

$$\text{time} = Nflt_{mpC} + \frac{N(1-f)}{2}lt_{spS}$$
$$\text{throughput} = \frac{N}{\text{time}} = \frac{2}{2flt_{mpC} + (1-f)lt_{spS}}$$

6.4 Experimental Validation

We measured the model parameters for our implementation. The values are shown in Table 2. Figure 10 shows the analytical model using these parameters, along with the

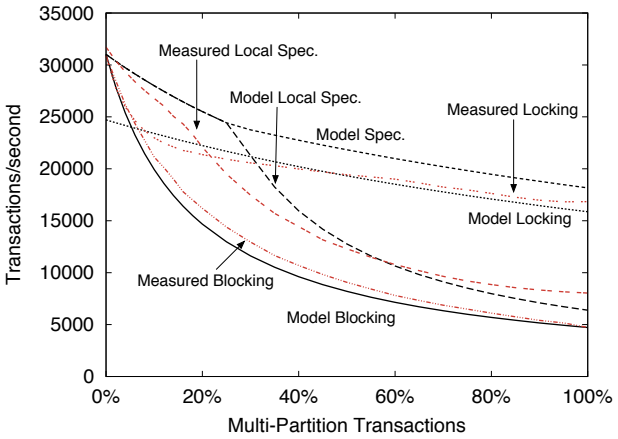

Figure 10: Model Throughput

measured throughput for our system without replication. As can be observed, the two are a relatively close match. This suggests that the model is a reasonable approximation for the behavior of the real system. These results also show that speculating multi-partition transactions leads to a substantial improvement when they comprise a large fraction of the workload.

Variable	Measured	Description
t_{sp}	64 μs	Time to execute a single partition transaction non-speculatively.
t_{spS}	73 μs	Time to execute a single partition transaction speculatively.
t_{mp}	211 μs	Time to execute a multi-partition transaction, including resolving the two-phase commit.
t_{mpC}	55 μs	CPU time used to execute a multi-partition transaction.
t_{mpN}	40 μs	Network stall time while executing a multi-partition transaction.
l	13.2%	Locking overhead. Fraction of additional execution time.

Table 2: Analytical Model Variables

7. RELATED WORK

Most distributed database systems (e.g., [9, 6, 19]) use some form of two-phase locking for processing concurrent queries, which our results show is best for low contention workloads. Other schemes, such as timestamp ordering [5], can avoid deadlocks but still allow multiple transactions to execute concurrently, and so they require read/write sets, latching, and must support rollback.

The first proposal that large main memory capacity could be used to eliminate concurrency control appears to have been made by Garcia-Molina, Lipton and Valdes [11, 12]. Some early main memory databases used this technique [16]. Shasha et al. [27] presented a database execution engine with a similar design as ours. They also observe that a scheme similar to our blocking approach can offer significant performance gains on transaction processing workloads. However, their work was done in the context of disk-based systems that still involved logging, and they did not investigate speculation and single-threaded locking schemes as we do. Instead, they provide a system for the user to specify which transactions conflict. The Sprint middleware system partitions data across multiple commercial in-memory database instances [7]. It also relies on transactions being classified in advance as single-partition or multi-partition in advance. Unlike our scheme, it writes transaction logs to disk and uses the traditional concurrency control provided by the in-memory databases.

One variant of two-phase commit that is similar to our work is OPT [13], where transactions are permitted to "borrow" dirty data while a transaction is in the prepared phase. While their work assumes locks, it is very similar to our "local speculation" scheme. However, OPT only speculates one transaction, while speculative concurrency control will speculate many, and can overlap the two-phase commit for multi-partition transactions from the same coordinator. Reddy and Kitsuregawa proposed speculative locking, where a transaction processes both the "before" and "after" version for modified data items [15]. At commit, the correct execution is selected and applied by tracking data dependencies between transactions. This approach assumes that there are ample computational resources. In our environment, CPU cycles are limited, and thus our speculative concurrency control always acts on the "after" version, and does not track dependencies at a fine-granularity.

Some work has noted that locking does not work well in high-contention situations [3], and has recommended optimistic concurrency control for these cases. Our observation is similar, although our preliminary results for OCC suggests that it does not help in our setting because tracking read and write sets is expensive; instead, in high-contention settings, we find partitioning with speculation to be effective.

Data partitioning is a well-studied problem in database systems (e.g., [18, 24, 8, 17, 10], amongst others). Past work has noted that partitioning can effectively increase the scalability of database systems, by parallelizing I/O [17] or by assigning each partition to separate workers in a cluster [18]. Unlike this past work, our focus is on partitioning for eliminating the need for concurrency control.

H-Store [26] presents a framework for a system that uses data partitioning and single-threaded execution to simplify concurrency control. We extend this work by proposing several schemes for concurrency control in partitioned, main-memory databases, concluding that the combination of blocking and OCC proposed in the H-Store paper are often outperformed by speculation.

Previous work on measuring overheads of locking, latching, multi-threading, and concurrency control in Shore [14] showed that the overhead of all of these subsystems in significant. This paper extends this previous work by showing that some form of concurrency control can be beneficial in highly contended workloads, and that in some cases even locking and multi-threading can prove to be beneficial.

8. CONCLUSIONS

In this paper, we studied the effects of low overhead concurrency control schemes on the performance of main memory partitioned databases. We found that speculative con-

currency control, which overlaps the commit phase of an earlier transaction with the execution of later transactions, works well provided that the workload is composed of single partition transactions, abort rate is low, and only a few transactions involve multiple rounds of communication. Otherwise lightweight locking schemes which allow greater degrees of concurrency are preferable. On a (slightly-altered) TPC-C benchmark, speculative concurrency control was the clear winner, in some cases increasing throughput by a factor of two relative to locking. Our results are particularly relevant to systems that rely on partitioning to achieve parallelism because they show that a workload that does not partition perfectly can be executed without the overheads of locking-based concurrency control.

9. ACKNOWLEDGMENTS

This work was sponsored by the NSF under grants IIS-0845643 and IIS-0704424, and by the Natural Sciences and Engineering Research Council of Canada. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation (NSF).

10. REFERENCES

[1] TPC benchmark C. Technical report, Transaction Processing Performance Council, February 2009. Revision 5.10.1.

[2] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In *Proc. ACM SIGMOD*, 1995.

[3] R. Agrawal, M. J. Carey, and M. Livny. Concurrency control performance modeling: Alternatives and implications. *ACM Trans. Database Syst.*, 12(4):609–654, 1987.

[4] S. Agrawal, V. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *Proc. ACM SIGMOD*, 2004.

[5] P. A. Bernstein and N. Goodman. Timestamp-based algorithms for concurrency control in distributed database systems. In *Proc. VLDB*, 1980.

[6] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. Prototyping bubba, a highly parallel database system. *IEEE Trans. on Knowl. and Data Eng.*, 2(1):4–24, 1990.

[7] L. Camargos, F. Pedone, and M. Wieloch. Sprint: a middleware for high-performance transaction processing. *SIGOPS Oper. Syst. Rev.*, 41(3):385–398, June 2007.

[8] S. Ceri, S. Navathe, and G. Wiederhold. Distribution design of logical database schemas. *IEEE Trans. Softw. Eng.*, 9(4):487–504, 1983.

[9] D. J. Dewitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. I. Hsiao, and R. Rasmussen. The gamma database machine project. *IEEE Trans. on Knowl. and Data Eng.*, 2(1):44–62, 1990.

[10] G. Eadon, E. I. Chong, S. Shankar, A. Raghavan, J. Srinivasan, and S. Das. Supporting table partitioning by reference in oracle. In *Proc. ACM SIGMOD*, 2008.

[11] H. Garcia-Molina, R. J. Lipton, and J. Valdes. A massive memory machine. *Computers, IEEE Transactions on*, C-33(5):391–399, 1984.

[12] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Trans. on Knowl. and Data Eng.*, 4(6):509–516, 1992.

[13] R. Gupta, J. Haritsa, and K. Ramamritham. Revisiting commit processing in distributed database systems. In *Proc. ACM SIGMOD*, pages 486–497, New York, NY, USA, 1997. ACM.

[14] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *Proc. ACM SIGMOD*, pages 981–992, 2008.

[15] P. Krishna Reddy and M. Kitsuregawa. Speculative locking protocols to improve performance for distributed database systems. *Knowledge and Data Engineering, IEEE Transactions on*, 16(2):154–169, March 2004.

[16] K. Li and J. F. Naughton. Multiprocessor main memory transaction processing. In *Proc. Databases in Parallel and Distributed Systems (DPDS)*, 1988.

[17] M. Livny, S. Khoshafian, and H. Boral. Multi-disk management algorithms. *SIGMETRICS Perform. Eval. Rev.*, 15(1):69–77, 1987.

[18] M. Mehta and D. J. DeWitt. Data placement in shared-nothing parallel database systems. *The VLDB Journal*, 6(1):53–72, 1997.

[19] C. Mohan, B. Lindsay, and R. Obermarck. Transaction management in the R* distributed database management system. *ACM Trans. Database Syst.*, 11(4):378–396, 1986.

[20] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. In *PVLDB*, 2010.

[21] S. Papadomanolakis and A. Ailamaki. Autopart: Automating schema design for large scientific databases using data partitioning. In *Proc. Scientific and Statistical Database Management*, 2004.

[22] D. V. Pattishall. Friendster: Scaling for 1 billion queries per day. In *MySQL Users Conference*, April 2005.

[23] D. V. Pattishall. Federation at Flickr (doing billions of queries per day). In *MySQL Conference*, April 2007.

[24] D. Sacca and G. Wiederhold. Database partitioning in a cluster of processors. *ACM Transactions on Database Systems*, 10(1):29–56, 1985.

[25] R. Shoup and D. Pritchett. The eBay architecture. In *SD Forum*, November 2006.

[26] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proc. VLDB*, 2007.

[27] A. Whitney, D. Shasha, and S. Apter. High volume transaction processing without concurrency control, two phase commit, SQL or C++. In *Int. Workshop on High Performance Transaction Systems*, 1997.

[28] D. C. Zilio. *Physical Database Design Decision Algorithms and Concurrent Reorganization for Parallel Database Systems*. PhD thesis, University of Toronto, 1998.