Low Overhead Concurrency Control for Partitioned Main Memory Databases

Evan Jones, Daniel Abadi, Samuel Madden, June 2010, SIGMOD

CS 848
May, 2016
Michael Abebe
Background
Motivations

• Database partitioning is important because it allows transactions to run on a single site in a distributed environment

• However, some transactions need to run across partitions, which results in distributed transactions

• What concurrency control techniques should be used to take advantage of the network stalls incurred during distributed transaction?
Two Phased Commit (2PC)

- **Purpose:** atomically commit a transaction with multiple coordinators
- **Algorithm**
  - **Voting phase**
    - Coordinator sends a “prepare to commit message” to participants
    - Participants reply with an “agreement” which is whether they locally have decided whether to commit or abort
  - **Commit Phase**
    - Once coordinator has received an agreement message from every participant they send a commit or abort message to every participant
    - Each participant acknowledges the decision and performs the required activity (commit or abort)
- **Blocking algorithm**
- **Fault tolerance**
  - By logging the results at each stage can handle failure of coordinator or failure of participants.
Low Overhead Concurrency Control

- If data is stored in main memory it may be more efficient to have only single thread executing a transaction at a time.
- Overhead of locking, latching and undo buffers represents up to 42% of CPU instructions on a TPC-C workload.
- If “perfectly partitionable” can run every transaction at one partition.
- Goal is to design a scheme that runs concurrently but only during network stalls.

Harizopoulos et. al, 2008

![Graph showing the breakdown of instructions with categories: buffer manager (34.6%), latching (16.3%), locking (16.2%), logging (11.9%), hand-coded optimizations (14.2%)]
Partitioning

- In shared nothing architecture partition data across $n$ servers
- Each server “owns” some of the data
- If “perfectly partitionable” can run every transaction at one partition
- In reality there will always be “imperfectly partitionable” workloads
Design and Architecture
System Design

- Data stored in partitions
  - Single process that stores data in memory
  - With \( k -1 \) backup copies
- Central coordinator, which coordinates distributed transactions to ensure a global order
- Client library submits the requests and gets the partitioning scheme for single partition transactions
- Transactions
  - Stored procedures
  - Broken down into fragments

Jones, Abadi, Madden, 2010
Executing Transactions

- Single Partition Transactions
  - Primary uses replication protocol to ensure durability
  - Then execute the transaction without blocking
  - No concurrency control is needed

- Multi-Partition Transaction
  - Go through central coordinator to assign global order
  - Divide transaction into fragments and send them to partition
  - After each fragment executes coordinator passes results to application logic to decide whether to continue executing
  - Execute using 2PC
Concurrency Control Schemes

• Blocking
  • Queue requests and do not process any other requests during network stalls

• Speculative Execution
  • Execute other transactions during network stalls as if the waiting transaction will commit
    • Can only begin speculative transactions once all work done at partition is done
  • Wait to return speculative transactions until commit or abort decision comes
  • Complicated when speculating multi-partition transactions

• Locking
  • Lock data items with read and write locks while executing
  • Only lock during multi-partition transactions as single thread executing
Experimental Analysis
Experiments

- Microbenchmarks which tests the features of the different concurrency control schemes
  - Simple key/value store
- TPC-C benchmark executing transactions directly on data as either a B-Tree, binary table or hash table, for more representative OLTP workload
- Measure transaction throughput
  - Confidence intervals are within few percent
cheaper to execute than normal transactions, since the abort will abort locally. The other partition will be aborted during a multi-partition transaction is selected, only one partition actions to be aborted at random with probability To understand the e a transaction is aborted, the speculatively executed transac-

5.3 Aborts speculation is up to 2.5 times faster than locking. avoiding concurrency control is larger. In this experiment, conflicts between transactions are common, the advantage of work concurrently. However, these results do suggest that if conflicts at one of the partitions, so it still performs some transactions because in this workload, each transaction only outperforms blocking when there are many multi-partition increases, locking behaves more like blocking. Locking still input falls o the nearly straight line as before, with conflicts the through-

put follows the same trend as before, only lower because the

steeply as the percentage of multi-partition trans-

ects of re-execution, we select trans-

Multi-Partition Transactions

Transactions/second

Speculation

Locking

Blocking

Jones, Abadi, Madden, 2010

Microbenchmark: Multi-Partition Transactions

transactions conflict. The performance of locking, on the conflict probability. This is because they assume that all and blocking, as their throughput does not change with the implementation dependent deadlock resolution policies.

results in more conflicts. Deadlocks are not possible in this key at a same time as the first two clients. Increasing have a very high probability of attempting to update the private keys with probability 1 written. To cause conflicts, the other clients write one of keys on their respective partitions are nearly always being written. To cause conflicts, the other clients write one of keys at random. This means the first two clients' transactions perform the reads and returns the results to the coordinator, which then issues the writes as a second transaction performs the reads and returns the results to the coordinator. The first round of each communication, instead of the simple multi-partition trans-
sue a multi-partition transaction that requires two rounds of action in the original benchmark. The first round of each communication, instead of the simple multi-partition trans-
sue a multi-partition transaction that requires two rounds of action in the original benchmark. The first round of each communication, instead of the simple multi-partition trans-
sue a multi-partition transaction that requires two rounds of action in the original benchmark. The first round of each communication, instead of the simple multi-partition trans-
sue a multi-partition transaction that requires two rounds of action in the original benchmark. The first round of each communication, instead of the simple multi-partition trans-
sue a multi-partition transaction that requires two rounds of action in the original benchmark. The first round of each communication, instead of the simple multi-partition trans-
sue a multi-partition transaction that requires two rounds of action in the original benchmark. The first round of each communication, instead of the simple multi-partition trans-
sue a multi-partition transaction that requires two rounds of action in the original benchmark. The first round of each communication, instead of the simple multi-partition trans-
sue a multi-partition transaction that requires two rounds of action in the original benchmark. The first round of each communication, instead of the simple multi-partition trans-
sue a multi-partition transaction that requires two rounds of action in the original benchmark. The first round of each communication, instead of the simple multi-partition trans-
sue a multi-partition transaction that requires two rounds of action in the original benchmark. The first round of each communication, instead of the simple multi-partition trans-
sue a multi-partition transaction that requires two rounds of action in the original benchmark. The first round of each communication, instead of the simple multi-partition trans-
sue a multi-partition transaction that requires two rounds of action in the original benchmark. The first round of each communication, instead of the simple multi-partition trans-
sue a multi-partition transaction that requires two rounds of action in the original benchmark.

Thus, the 95% confidence intervals are wider for this experi-

As a result, the abort rate does not have a significant impact, so 1% abort 5%, so we omit them for clarity. Since blocking and locking do not have cascading impact of these multi-

direct impact of conflicts, we change the pattern in all other respects, (e.g., network message length).

happens at the beginning of execution. They are identical in all other respects, (e.g., network message length).


cheaper to execute than normal transactions, since the aborting two-phase commit. Aborted transactions are somewhat will abort locally. The other partition will be aborted dur-
a multi-partition transaction is selected, only one partition actions to be aborted at random with probability
To understand the e
transactions must be undone and re-executed, wasting CPU time. A
transaction is aborted, the speculatively executed transac-
Speculation is up to 2.5 times faster than locking.
avoiding concurrency control is larger. In this experiment, conflicts between transactions are common, the advantage of work concurrently. However, these results do suggest that if transactions because in this workload, each transaction only outperforms blocking when there are many multi-partition 
creases, locking behaves more like blocking. Locking still inputs falls o
the nearly straight line as before, with conflicts the through-
other hand, degrades as conflict rate increases. Rather than transactions conflict. The performance of locking, on the conflict probability. This is because they assume that all and blocking, as their throughput does not change with the results in more conflicts. Deadlocks are not possible in this key at a same time as the first two clients. Increasing these "conflict" keys with probability
written. To cause conflicts, the other clients write one of keys on their respective partitions are nearly always being
partition at random. This means the first two clients' first partition, and the second client only issues transactions, the first client only issues transactions to the of keys that clients access. When issuing single partition
investigate the impact of conflicts, we change the pattern Multi-Partition Transactions
5.4 General Multi-Partition Transactions

The results in Figure 5 show a single line for speculation Transactions/second
10000
15000
20000
25000
30000

Blocking
Speculation
Locking 0% conflict
Locking 20% conflict
Locking 60% conflict
Locking 100% conflict
0%
20%
40%
60%
80%
100%

Transactions/second
Multi-Partition Transactions
Jones, Abadi, Madden, 2010
updated in a single partition. This partitioning means 89% columns across all partitions, leaving the columns that are 
tically partition the stock table, and replicate the read-only 
the items table, which is read-only, to all partitions. We ver-
house, as described by Stonebraker et al. [26]. We replicate 
the other tables. We partition the TPC-C database by ware-
by adding warehouses, which adds a set of related records to 
transactions with di 

5.5 TPC-C

This workload, speculation does still outperform locking as 
communication. Even though locking is generally superior for 
tively una 

5.6 TPC-C Multi-Partition Scaling

speculation performs only slightly better, since it can only 
speculate the first fragment of the next multi-partition trans-
the multi-partition transactions in the original benchmark.

Abortion Rate

Jones, Abadi, Madden, 2010
Microbenchmark: Multi-round Multi-partition transactions

![Graph showing transactions per second versus multi-partition transactions]
Optimistic concurrency control (OCC) is another "standard" concurrency control algorithm. It requires tracking read/write sets of a transaction — which OCC also must memoize. It eliminates all locking and other concurrency control methods and our speculation hypothesis, and plan to explore the trade-offs between OCC and 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 item that is read and written, and aborts transactions during a validation phase if there were conflicts. Intuitively, 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 $P_1$, and 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 simulates the global state, and only sends two fragments out, one to each partition, waits for each fragment to be returned, and then executes the update on the object. The coordinator collects the results from each update, and then sends them back to the client. The client then checks if there were any conflicts, and if there were, aborts the transaction.

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 many parameters (which would be useful in a query planner, for example), and allows us to explore the sensitivity of the performance to workload characteristics (such as the CPU cost per item that is read and written, and aborts transactions). We expect the performance for OCC to be similar to that of locking or blocking, while the performance of speculation will depend 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.

Figure 8: TPC-C Throughput Varying Warehouses

Figure 9: TPC-C 100% New Order

Table 1: Summary of best concurrency control schemes as different situations. Speculation is preferred when there are few multi-round transactions and few aborts. Blocking or locking is preferred when there are many transactions with multiple rounds of communication and when a low percentage of transactions abort. Our low overhead locking technique is best preferred when there are few multi-round transactions that require only a single round of communication and when a low percentage of transactions abort.

Jones, Abadi, Madden, 2010
Optimistic concurrency control (OCC) is another “standard” concurrency control algorithm. It requires tracking the read/write sets of a transaction — which OCC also must implement. OCC is implemented with a fine-grain locking protocol. 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 makes traditional concurrency control more expensive, incurring throughput significantly. Again, this shows that conflicts are eliminated. We have run some initial results that verify this hypothesis, and plan to explore the trade-off 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.

6. ANALYTICAL MODEL

We consider a database divided into two partitions, one for each warehouse. The workload consists of two transactions. The first is a single partition transaction that accesses only one partition. 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 replies, and then proceeds. The time to complete the second transaction is the time for the coordinator to send and receive two fragments.

To improve our understanding of the concurrency control communication, we developed a simple analytical model to predict the performance of the three schemes in terms of transactions/second. The model involves little more than keeping track of the latency to the partitions (either a single round of communication or a validation phase if there were conflicts). Intuitively, we expect the performance for OCC to be similar to that for locking, when there are many transactions with multiple rounds of read/write. Speculation is preferred when there are few multi-round (general) transactions and few aborts. Our low overhead locking technique is best preferred when there are few multi-round (general) transactions and many aborts. The analytical model is presented in Section 6.1 below to make the best choice.

We have run some initial results that verify this hypothesis, and plan to explore the trade-off 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.
Some early main memory databases used this technique [16]. Garcia-Molina, Lipton and Valdes [11, 12] have made improvements that do not help in our setting because tracking read and write sets is expensive; instead, in high-contention settings, write sets are eliminated. Concurrency control always acts on the "after" version, and does not track modified data items [15]. At commit, the correct execution is selected and applied by tracking data dependencies between unmodified data and modified data [15]. Abadi et al. [18] performed by speculation. Unlike this past work, our focus is on partitioning by assigning each partition to separate workers in a cluster for high scalability of database systems, by parallelizing I/O [17] or reducing the overhead of computing, multi-threading, and concurrency control in Shore [14].

Data partitioning is a well-studied problem in database systems that still involved logging, and they did not investigate performance gains on transaction processing workloads. How-ever, their work was done in the context of disk-based systems. Previous work on measuring overheads of locking, latching, and must support rollback. As their work assumes locks, it is very similar to our "locking overhead" dirty data while a transaction is in the prepared phase.

In this paper, we studied the effects of low overhead concurrency control schemes on the performance of main memory databases. We found that speculative concurrency control will speculative transactions from the same coordinator. Reddy et al. [19] use a "multi-speculation" scheme. However, OPT only speculates one row dirty data while a transaction is in the prepared phase. While their work assumes locks, it is very similar to our "locking overhead" dirty data while a transaction is in the prepared phase. OPT only speculates one row dirty data while a transaction is in the prepared phase. They also observe that a scheme 

Shasha et al. [27] presented a database execution engine with speculative multi-partition transactions leads to a substantial improvement when they comprise a large fraction of the workload. 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, which our results show is best for low contention workloads. Some work has noted that locking does not work well in highly contended workloads, and that in some cases even the simplest locking schemes on the performance of main memory databases. We found that speculative concurrency control can prove to be beneficial.

Table 2: Analytical Model Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>spS</td>
<td>Time to execute a single partition non-speculatively.</td>
</tr>
<tr>
<td>spC</td>
<td>Time to execute a single partition speculatively.</td>
</tr>
<tr>
<td>mpS</td>
<td>Time to execute a multi-partition transaction non-speculatively.</td>
</tr>
<tr>
<td>mpC</td>
<td>Time to execute a multi-partition transaction speculatively.</td>
</tr>
<tr>
<td>µs</td>
<td>CPU time used to execute a multi-partition transaction.</td>
</tr>
<tr>
<td>µn</td>
<td>Network stall time while executing a multi-partition transaction.</td>
</tr>
<tr>
<td>Models</td>
<td>Transaction speculatively.</td>
</tr>
<tr>
<td></td>
<td>Transaction non-speculatively.</td>
</tr>
<tr>
<td></td>
<td>Locking.</td>
</tr>
<tr>
<td></td>
<td>Blocking.</td>
</tr>
</tbody>
</table>

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]. Unlike our scheme, it writes transaction logs to disk and in advance as single-partition or multi-partition in advance. Instead, they provide a system for the user to specify which transactions data across multiple system databases.

With OCC proposed in the H-Store paper are often outperforming, multi-threading, and concurrency control in Shore [14]. Unlike our scheme, it writes transaction logs to disk and in advance as single-partition or multi-partition in advance. Instead, they provide a system for the user to specify which transactions data across multiple system databases.

Table 2: Analytical Model Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>spS</td>
<td>Time to execute a single partition non-speculatively.</td>
</tr>
<tr>
<td>spC</td>
<td>Time to execute a single partition speculatively.</td>
</tr>
<tr>
<td>mpS</td>
<td>Time to execute a multi-partition transaction non-speculatively.</td>
</tr>
<tr>
<td>mpC</td>
<td>Time to execute a multi-partition transaction speculatively.</td>
</tr>
<tr>
<td>µs</td>
<td>CPU time used to execute a multi-partition transaction.</td>
</tr>
<tr>
<td>µn</td>
<td>Network stall time while executing a multi-partition transaction.</td>
</tr>
<tr>
<td>Models</td>
<td>Transaction speculatively.</td>
</tr>
<tr>
<td></td>
<td>Transaction non-speculatively.</td>
</tr>
<tr>
<td></td>
<td>Locking.</td>
</tr>
<tr>
<td></td>
<td>Blocking.</td>
</tr>
</tbody>
</table>

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].

Figure 10: Model Throughput

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>spS</td>
<td>Time to execute a single partition non-speculatively.</td>
</tr>
<tr>
<td>spC</td>
<td>Time to execute a single partition speculatively.</td>
</tr>
<tr>
<td>mpS</td>
<td>Time to execute a multi-partition transaction non-speculatively.</td>
</tr>
<tr>
<td>mpC</td>
<td>Time to execute a multi-partition transaction speculatively.</td>
</tr>
<tr>
<td>µs</td>
<td>CPU time used to execute a multi-partition transaction.</td>
</tr>
<tr>
<td>µn</td>
<td>Network stall time while executing a multi-partition transaction.</td>
</tr>
<tr>
<td>Models</td>
<td>Transaction speculatively.</td>
</tr>
<tr>
<td></td>
<td>Transaction non-speculatively.</td>
</tr>
<tr>
<td></td>
<td>Locking.</td>
</tr>
<tr>
<td></td>
<td>Blocking.</td>
</tr>
</tbody>
</table>

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].
Conclusions
Summary of Concurrency Control

Multi-round transactions?

- Few
- Many ➔ Locking!

The TPC-C benchmark models the OLTP workload of transactions with different properties. The data size is scaled to accommodate this, our clients generate requests for an assigned warehouse but a random district. 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.

We executed a workload that is composed of 100% new order transactions. In this workload, the fraction of multi-partition transactions is within the results for the microbenchmark in Figure 4. In this experiment, we reorder the transactions with this workload is shown in Figure 9.

We partition the TPC-C database by 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 increase this probability to 0.1 (10%). With TPC-C's default parameters, this probability is 0.01 (1%), which produces a multi-partition transaction 9.5% of the time. We increase this probability to 0.1 (10%).

At 0% multi-partition transactions, it runs exactly as expected with this workload. The throughput for Speculation is 10% aborts, Speculation 5% aborts, Speculation 3% aborts, Speculation 0% aborts, Blocking 10% aborts, Locking 10% aborts, Locking 0% aborts. The performance is lowest with 2 partitions because the number of conflicting transactions decreases. This is 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 number of conflicts in the new order transaction to avoid needing an undo buffer. Operations in the new order transaction to avoid needing an undo buffer.

The results for blocking and speculation are very similar. The speculation provides 9.7% higher throughput than blocking, which is the best of the three schemes because the speculation performs the best of the three schemes because the communication overhead due to deadlock detection and distributed deadlock timeouts, decreasing the performance for locking. Speculation performs only slightly better, since it can only undo buffer.

The number of conflicting transactions decreases. This is due to nearly every transaction modifies the warehouse and district data. The performance is lowest with 2 partitions because the number of conflicting transactions decreases. This is 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 number of conflicts in the new order transaction to avoid needing an undo buffer. Operations in the new order transaction to avoid needing an undo buffer.

The performance is lowest with 2 partitions because the number of conflicting transactions decreases. This is 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 number of conflicts in the new order transaction to avoid needing an undo buffer. Operations in the new order transaction to avoid needing an undo buffer.

The performance is lowest with 2 partitions because the number of conflicting transactions decreases. This is 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 number of conflicts in the new order transaction to avoid needing an undo buffer. Operations in the new order transaction to avoid needing an undo buffer.

The performance is lowest with 2 partitions because the number of conflicting transactions decreases. This is 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 number of conflicts in the new order transaction to avoid needing an undo buffer. Operations in the new order transaction to avoid needing an undo buffer.
Summary of Concurrency Control

- **Multi-round transactions?**
  - Few ➔ Aborts?
    - Few ➔ Speculation!
  - Many ➔ Locking!

- **Transactions/second**
  - Speculation 0% aborts
  - Speculation 3% aborts
  - Speculation 5% aborts
  - Speculation 10% aborts
  - Blocking 10% aborts
  - Locking 10% aborts

- **Transactions/second**
  - 0% ➔ 100%

- **Multi-Partition Transactions**
  - 0% ➔ 100%

---

*Figure 7: General Transaction Microbenchmark*

*Figure 6: Microbenchmark With Aborts*
Summary of Concurrency Control

Multi-round transactions?  
  - Few → Aborts?  
    - Few → Speculation!  
    - Many → Multi-partition transactions?  
      - Few → Conflicts?  
        - Few → Blocking or Locking!  
        - Many → Blocking!  
      - Many → Conflicts  
        - Many → Locking or Speculation!
  - Many → Locking!

Transactions/second vs. Multi-Partition Transactions

- Locking 0% conflict  
- Locking 20% conflict  
- Locking 60% conflict  
- Locking 100% conflict  
- Speculation  
- Blocking
Critique and Thoughts
Critique of the Paper

• Strengths
  • Dealing with imperfectly partitionable applications network delays in a concurrent way is a real problem
  • Performing both microbenchmarks and a real workload

• Weaknesses
  • Ignored the network delays that come from durability guarantees for replication
  • No comparisons against systems which are already concurrent
Competition and Other approaches

- True concurrency
  - Multi-Version Concurrency Control
  - True Optimistic Concurrency Control
- Avoiding distributed commit protocols
  - Convert distributed transactions to local transactions by dynamically moving data around
- Avoid holding locks while deciding on an ordering of distributed transactions
  - Machines decide on an order of distributed transactions outside of their transactional boundaries

