Concurrency Control In Distributed Main Memory Database Systems

Justin A. DeBrabant
debbrabant@cs.brown.edu
Concurrency control

• Goal:
  – maintain consistent state of data
  – ensure query results are correct

• The Gold Standard: ACID Properties
  – atomicity – “all or nothing”
  – consistency – no constraints violated
  – isolation – transactions don’t interfere
  – durability – persist through crashes
Why?

• Let’s just keep it simple...
  – serial execution of all transactions
  – e.g. T1, T2, T3
  – simple, but boring and slow

• The Real World:
  – interleave transactions to improve throughput
  • …crazy stuff starts to happen
Traditional Techniques

• Locking
  – lock data before reads/writes
  – provides isolation and consistency
  – 2-phase locking
    • phase 1: acquire all necessary locks
    • phase 2: release locks (no new locks acquired)
    • locks: shared and exclusive

• Logging
  – used for recovery
  – provides atomicity and durability
  – write-ahead logging
    • all modifications are written to a log before they are applied
How about in parallel?

- many of the same concerns, but must also worry about committing multi-node transactions
- distributed locking and deadlock detection can be expensive (network costs are high)
- 2-phase commit
  - single coordinator, several workers
  - phase 1: voting
    - each worker votes “yes” or “no”
  - phase 2: commit or abort
    - consider all votes, notify workers of result
The Issue

• these techniques are very general purpose
  – “one size fits all”
  – databases are moving away from this

• By making assumptions about the system/workload, can we do better?
  – YES!
  – keeps things interesting (and us employed)
Paper 1

- *Low Overhead Concurrency Control for Partitioned Main Memory Databases*
  - Evan Jones, Daniel Abadi, Sam Madden
  - SIGMOD ‘10
Overview

• Contribution:
  – several concurrency control schemes for distributed main-memory databases

• Strategy
  – Take advantage of network stalls resulting from multi-partition transaction coordination
  – don’t want to (significantly) hurt performance of single-partition transactions
    • probably the majority
System Model

• based on H-Store
• partition data to multiple machines
  – all data is kept in memory
  – single execution thread per partition
• central coordinator that coordinates
  – assumed to be single coordinator in this paper
    • multi-coordinator version more difficult
System Model (cont’d)

3.2 Single Partition Transactions

A single partition transaction for example is composed of one fragment containing the entire transaction. A multi-transaction is composed of multiple fragments with data dependencies between them. For each transaction, a fragment is a self-contained unit of computation. A fragment is equivalent to forcing the participant's tPC vote to disk. For example, by using loosely synchronized clocks [t], we can globally order transactions with multiple coordinators. To globally order transactions with multiple coordinators, we forward them through the central coordinator, which assigns them a global order. We do not keep an undo log. Otherwisen we forward them through partition data dependencies between transaction fragments. So a fragment is equivalent to forcing the participant’s tPC vote to disk. For example, by using loosely synchronized clocks [t], we can globally order transactions with multiple coordinators.

Network stalls can occur while waiting for data from other partition servers. This can introduce a performance bottleneck, even if the application isn’t execution a multi-transaction sequentially. The coordinator piggybacks the tPC “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. After receiving the acknowledgments, the primary executes the transaction and sends the results to the coordinator. The coordinator forwards the results to the client. When a client determines that a request is a single partition transaction, it forwards it to the primary partition coordinator limits the rate of multi-transaction to the central coordinator, which assigns them a global order. We do not keep an undo log. Otherwise, we forward them through partition data dependencies between transaction fragments. So a fragment is equivalent to forcing the participant’s tPC vote to disk. For example, by using loosely synchronized clocks [t], we can globally order transactions with multiple coordinators.

4. CONCURRENCY CONTROL SCHEMES

The simplest scheme for handling multi-transaction is to block until they complete. When the partition transactions are processed, they are processed in order. After the transaction is executed and the results are returned, all other transactions are processed.

When all acknowledgments from the backups are received, the transaction is sent to the client. This process is responsible for the data in the database. The primary uses a typical priority queue to ensure durability. In the failure-free case, the primary reads the request from the backup replication protocol to ensure durability. In the fail case, the primary reads the request from the backup replication protocol to ensure durability. In the fail case, the primary reads the request from the backup replication protocol to ensure durability. In the fail case, the primary reads the request from the backup replication protocol to ensure durability.
Transaction Types

- **Single Partition Transactions**
  - client forwards request directly to primary partition
  - primary partition forwards request to backups

- **Multi-Partition Transactions**
  - client forwards request to coordinator
  - transaction divided into fragments and forwards them to the appropriate transactions
  - coordinator uses undo buffer and 2PC
  - network stalls can occur as a partition waits for other partitions for data
  - network stalls twice as long as average transaction length
Concurrency Control Schemes

• **Blocking**
  – queue all incoming transactions during network stalls
  – simple, safe, slow

• **Speculative Execution**
  – speculatively execute queued transactions during network stalls

• **Locking**
  – acquire read/write locks on all data
Blocking

• for each multi-partitioned transaction, block until it completes
• other fragments in the blocking transaction are processed in order
• all other transactions are queued
  – executed after the blocking transaction has completed all fragments
Speculative Execution

- speculatively execute queued transactions during network stalls
- must keep undo logs to roll back speculatively executed transaction if transaction causing stall aborts
- if transaction causing stall commits, speculatively executed transaction immediately commit
- two cases:
  - single partition transactions
  - multi-partition transactions
Speculating Single Partitions

- wait for last fragment of multi-partition transaction to execute
- begin executing transactions from unexecuted queue and add to uncommitted queue
- results must be buffered and cannot be exposed until they are known to be correct
Speculating Multi-Partitions

• assumes that 2 speculative transactions share the same coordinator
  – simple in the single coordinator case
• single coordinator tracks dependencies and manages all commits/aborts
  – must cascade aborts if transaction failure
• best for simple, single-fraction per partition transactions
  – e.g. distributed reads
Locking

- locks allow individual partitions to execute and commit non-conflicting transactions during network stalls
- problem: overhead of obtaining locks
- optimization: only require locks when a multi-partition transaction is active
- must do local/distributed deadlock
  - local: cycle detection
  - distributed: timeouts
Microbenchmark Evaluation

• Simple key/value store
  – keys/values arbitrary strings
• simply for analysis of techniques, not representative of real-world workload
Microbenchmark Evaluation

Concurrency Control
Microbenchmark Evaluation

Figure 6: Microbenchmark With Aborts

Figure 7: General Transaction Microbenchmark

Transactions/second

0 5000 10000 15000 20000 25000 30000
0% 20% 40% 60% 80% 100%

Speculation 0% aborts
Speculation 3% aborts
Speculation 5% aborts
Speculation 10% aborts
Blocking 10% aborts
Locking 10% aborts

Multi-Partition Transactions

Concurrency Control
TPC-C Evaluation

• TPC-C
  – common OLTP benchmark
  – simulates creating/placing orders at warehouses
• This benchmark is a modified version of TPC-C
Figure 8: TPC-C Throughput Varying Warehouses

Figure 9: TPC-C 100u 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 deadlock, 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 one-multipartition probability shows that over 80% of the execution time is spent in the lock implementation. Approximately 25% of the time is spent managing the lock table, 15% is spent acquiring locks, and 20% is spent releasing locks. While our locking implementation certainly has room for optimization, this is similar to what was previously measured for Shore, where over 20% of the CPU instructions could be attributed to locking.

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. Table 1 summarizes 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.

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, since each partition runs single-threaded file I/O, 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 multipartition scaling experiment from Section 5. 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, P1 and P2. The workload consists of two transactions. The first is a single partition transaction that accesses only P1 or P2, chosen uniformly at random. The second is a multipartition 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...
TPC-C Evaluation (100% New Order)

![Graph showing transactions per second against multi-partition transactions with three curves: Speculation (solid), Blocking (dashed), and Locking (dotted). The x-axis represents the percentage of multi-partition transactions, ranging from 0% to 100%, and the y-axis represents transactions per second, ranging from 0 to 35000. The graph indicates a decrease in transactions per second as the percentage of multi-partition transactions increases.]
### Evaluation Summary

<table>
<thead>
<tr>
<th>Few multi-round xactions</th>
<th>Many multi-partition xactions</th>
<th>Few Conflicts</th>
<th>Many Conflicts</th>
<th>Few Conflicts</th>
<th>Many Conflicts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Few multi-round xactions</td>
<td>Many multi-partition xactions</td>
<td>Speculation</td>
<td>Speculation</td>
<td>Locking</td>
<td>Locking or Speculation</td>
</tr>
<tr>
<td>Few multi-round xactions</td>
<td>Few multi-partition xactions</td>
<td>Speculation</td>
<td>Speculation</td>
<td>Blocking or Locking</td>
<td>Blocking</td>
</tr>
<tr>
<td>Many multi-round xactions</td>
<td>Many multi-round xactions</td>
<td>Locking</td>
<td>Locking</td>
<td>Locking</td>
<td>Locking</td>
</tr>
</tbody>
</table>
Paper 2

- *The Case for Determinism in Database Systems*
  - Alexander Thompson, Daniel Abadi
  - VLDB 2010
Overview

• Presents a deterministic database prototype
  – argues that in the age of memory-based OLTP systems (think H-Store), clogging due to disk waits will be a minimum (or nonexistant)
  – allows for easier maintenance of database replicas
Nondeterminism in DBMSs

- transactions are executed in parallel
- most databases guarantee consistency for some serial order of transaction execution
  - which?...depends on a lot of factors
  - key is that it is not necessarily the order in which transactions arrive in the system
Drawbacks to Nondeterminism

• Replication
  – 2 systems with same state and given same queries could have different final states
  • defeats the idea of “replica”

• Horizontal Scalability
  – partitions have to perform costly distributed commit protocols (2PC)
Why Determinism?

- nondeterminism is particularly useful for systems with long delays (disk, network, deadlocks, …)
  - less likely in main memory OLTP systems
  - at some point, the drawbacks of nondeterminism outweigh the potential benefits
How to make it deterministic?

• all incoming queries are passed to a preprocessor
  – non-deterministic work is done in advance
    • results are passed as transaction arguments
  – all transactions are ordered
  – transaction requests are written to disk
  – requests are sent to all database replicas
A small issue…

• What about transactions with operations that depend on results from a previous operation?
  – $y \leftarrow \text{read}(x), \text{write}(y)$
    • $x$ is the records primary key

• This transaction cannot request all of its locks until it knows the value of $y$
  – …probably a bad idea to lock $y$’s entire table
Dealing with “difficult” transactions

• Decompose the transaction into multiple transactions
  – all but the last are simply to discover the full read/write set of the original transaction
  – each transaction is dependent on the previous ones
• Execute the decomposed transactions 1 at a time, waiting for results of previous
System Architecture

Upon the failure of a replica, recovery in our system is performed by copying database state from a non-faulty replica. Recovery in our system is performed by copying database state from a non-faulty replica. Deterministic work (such as calls to sys.random() or time.now()) appears much less frequently in real-world workloads, but dependent transactions such as the second-order transaction index lookups followed by record accesses. Higher-order dependent transactions are often seen in OLTP workloads in the form of second-order dependent transactions. First-order dependent transactions can be decomposed into the transactions:

\[ U \leftarrow \text{read}(y) \]
\[ U \leftarrow \text{write}(z) \]
\[ U \leftarrow \text{write}(y) \]
\[ U \leftarrow \text{read}(x) \]
\[ V \leftarrow \text{read}(y) \]
\[ V \leftarrow \text{write}(z) \]
\[ V \leftarrow \text{write}(y) \]
\[ V \leftarrow \text{read}(x) \]

Immediate upon entering the system, it is clearly impossible to lock every record it accesses immediately upon entering. It then checks if it locked the record whose primary key is recorded transactions and after which all transactions that ran between the abort and includes all abort-and-retry actions are deterministic (the transaction's determinism invariant). Our scheme addresses the problem of dependent transactions where \( x \) is a record's primary key, \( y \) can proceed; however, if it fails, then \( z \) is returned to the preprocessor (any number of transactions can be run in the meantime). Since \( x \) can be decomposed into the transactions:

\[ U \leftarrow \text{newtxnrequest}(y) \]
\[ U \leftarrow \text{newtxnrequest}(y) \]
\[ U \leftarrow \text{newtxnrequest}(y) \]
\[ U \leftarrow \text{newtxnrequest}(y) \]

Concurrent Failover Mechanisms in Actively Replicated Systems Can Also Help Hide Performance Dips that a Replicated Database Systems Can Drastically Reduce the Effect Only a Subset of Replicas.
Evaluation

Figure 3: Deterministic vs. traditional throughput of TPC-C (100% New Order) workload, varying frequency of multipartition transactions.
Evaluation Summary

• In systems/workloads where stalls are sparse, determinism can be desirable
• Determinism has huge performance costs in systems with large stalls
• bottom line: good in some systems, but not everywhere
Paper 3

• *An Almost-Serial Protocol for Transaction Execution in Main-Memory Database Systems*
  – Stephen Blott, Henry Korth
  – VLDB 2002
Overview

• In main memory databases, there is a lot of overhead in locking

• naïve approaches that lock the entire database suffer during stalls when logs are written to disk

• main idea: maintain timestamps and allow non-conflicting transaction to execute during disk stalls
Timestamp Protocol

• Let transaction $T_1$ be a write on $x$
• Before $T_1$ writes anything, issue new timestamp $TS(T_1)$ s.t. $TS(T_1)$ is greater than any other timestamp
• When $x$ is written, $WTS(d)$ is set to $TS(T_1)$
• When any transaction $T_2$ reads $d$, $TS(T_2)$ is set to $\max(TS(T_2), WTS(d))$
Transaction Result

- If $T$ is an update transaction:
  - $TS(T)$ is a new timestamp, higher than any other

- If $T$ is a read-only transaction:
  - $TS(T)$ is the timestamp of the most recent transaction from which $T$ reads

- For data item $x$:
  - $WTS(x)$ is the timestamp of the most recent transaction that wrote into $x$
The Mutex Array

- an “infinite” array of mutexes, 1 per timestamp
- Commit Protocol:
  - Update
    - $T$ acquires database mutex, executes
    - When $T$ wants to commit, acquire $A[TS(T)]$, prior to releasing database mutex
    - $T$ releases $A[TS(T)]$ after receiving ACK that its commit record has been written to disk
  - Read-Only
    - release database mutex and acquire $A[TS(T)]$
    - immediately release $A[TS(T)]$, commit
Evaluation

Throughput

Percentage of transactions which are update transactions

Multi-programming level = 1 [ SP ]
Multi-programming level = 1 [ 2PL ]
General Conclusions

• As we make assumptions about query workload and/or database architecture, old techniques need to be revisited

• No silver bullet for concurrency/determinism questions
  – tradeoffs will depend largely on what is important to the user of the system
Questions?