Evaluation of Concurrency Control Schemes on Distributed Systems

Hannah Gommerstadt  Wennie Tabib  Dana Van Aken
hgommers@cs.cmu.edu  wtabib@cs.cmu.edu  dvanaken@cs.cmu.edu

1. Abstract
Concurrency on a massive scale is required to meet the needs of today’s increasingly distributed systems. To better understand how current concurrency control schemes work on distributed systems, this paper reports an evaluation of concurrency control schemes for online transaction processing (OLTP) workloads on distributed database management systems (DBMS) with a variable number of nodes. Five concurrency control schemes were implemented and tested on a cluster of nodes. We find that all five concurrency schemes fail to scale on a multinode system though variants of two-phase locking perform better than variants of timestamp ordering.

2. Introduction
As distributed systems continue to increase in size, scalable concurrency control algorithms are becoming critical for providing high availability and reliability. This paper examines concurrency control schemes in the context of transaction processing in a distributed database. We define transaction processing to mean many individual clients reading and writing to a database that is spread out over many machines. Data is partitioned disjointly between a set of machines called “Data Managers” (DMs). Clients do not directly access DMs. Rather, each client sends transactions to a single centralized coordinator that then routes individual data requests to the appropriate DMs. Because many clients may access the database simultaneously, the challenge with a distributed database is achieving scalability while maintaining consistency. Therefore, the goal of this paper is to comprehensively study the scalability of online transaction processing (OLTP) database management system (DBMS) by evaluating the performance of a group of well-known concurrency control schemes.

This paper evaluates five concurrency control algorithms in a distributed DBMS. Testing was conducted on Amazon’s Elastic Cloud Computing (EC2) service. Our system is a modification of an in-memory DBMS used to evaluate the scalability of multi-core systems. We transformed this multi-core system into a multinode system by first decoupling the client, concurrency control, and data storage code and then introducing a Remote Message Passing (RPC) mechanism to enable communication. Our analysis shows that all five concurrency schemes fail to scale to a large number of nodes with two-phase locking variants significantly outperforming timestamp ordering based ones.

The main contribution of this paper is the comprehensive evaluation of the scalability of five concurrency control schemes on a distributed DBMS and the identification of bottlenecks in these concurrency control schemes. The remainder of the paper is organized as follows. We begin in Section 3 with an overview of the related work. Section 4 describes the concurrency control schemes that our experiments tested and Section 5 describes the workload used in our analysis. We discuss the design of our system and experiment in Section 6 and Section 7. We then present the results in Section 8 and discuss them in Section 9. We conclude in Section 10 and discuss future directions in Section 11.

3. Related Work
This is an extension of previous work that evaluated concurrency control schemes for a single-node system (DBx1000) [1]. DBx1000 system tested the following control schemes: two-phase locking (2PL) with deadlock detection, 2PL with wait-and-die deadlock prevention, 2PL with no-waiting deadlock prevention, timestamp ordering (T/O), multi-version concurrency control (MVCC), optimistic concurrency control (OCC), and timestamp ordering with partition-level locking.
Two-phase locking requires that transactions acquire a lock prior to reading or writing to an element in a database. In this scheme, read locks can be shared, but write locks must be exclusive. Once a transaction relinquishes control of a lock it is not allowed to acquire any additional locks. Timestamp ordering concurrency control schemes generate a serialization order of transactions and then the DBMS enforces this order. MVCC, OCC, and HSTORE are considered variants of T/O. A more detailed explanation of these concurrency control schemes may be found in Section 4. DBx1000 evaluated the efficacy of these schemes in a multi-core architecture, whereas our work evaluates the performance of these schemes in a multi-node system. Our system is an adaptation of the DBx1000 system to run on a multi-node system with a centralized coordinator acting as an intermediary to route transactions from clients to the proper database.

More recent work has evaluated a subset of these schemes. Carey and Livny [4, 5] evaluated four concurrency control schemes – two-phase locking (2PL), wound-wait, basic timestamp ordering (TIMESTAMP) and distributed optimistic concurrency control (OPCC) on a system of up to eight nodes. Our paper presents a more comprehensive evaluation of well-known concurrency control algorithms in a multi-node system.

Ren et al [6] evaluated deterministic and nondeterministic systems. They found that deterministic systems significantly outperformed nondeterministic systems by not being prone to deadlock. As a result, deterministic systems are recommended for workloads that require extreme transactional scalability. These results quantify the inadequacy of state-of-art nondeterministic database systems for applications that support distributed transactions. We provide more evidence that speaks to the lack of scalability of concurrency control schemes to meet the challenges of distributed systems.

4. Concurrency Control Schemes

This paper considers an in-memory OLTP engine which is queried by a series of transactions. Each transaction consists of a number of requests, where each request is a single memory-access in the DBMS. In order to guarantee that the execution of transactions preserve the ACID properties (i.e., atomicity, consistency, isolation and durability), it is necessary to use a concurrency control algorithm. What follows is a description of the concurrency control schemes evaluated in our multi-node system. These concurrency control schemes are partitioned into two groups – variants of Two-phase Locking and variants of Timestamp Ordering.

4.1 Two-phase Locking & Variations

The basic idea of Two-phase Locking is that transactions must acquire locks for a particular element of the database before they may perform a read or a write on that element. There are two types of locks that can be acquired – read locks and write locks – where a read lock is acquired before attempting to read an element and a write lock is acquired before being allowed to modify an element. We do not allow two different transactions to own conflicting write locks on the same element or a read and write lock on the same element. Multiple transactions can, however, own a shared read lock to the same element. Once a transaction has obtained a lock there is no guarantee that it will be able to obtain additional locks. Each transaction proceeds through two phases. In the growing phase, each transaction is allowed to acquire as many locks as it needs to complete its task. In the shrinking phase, the transaction begins releasing locks and it is no longer allowed to obtain additional locks. At this point, the transaction either commits or aborts and all remaining locks are released. The main issue with 2PL arises in the situation where a transaction is unable to acquire a lock for an element and is forced to wait until the lock is released by another transaction. This waiting is can be indefinite and cause a deadlock. The variations of 2PL described below handle deadlock prevention in two different ways.

4.1.1 2PL with Non-waiting Deadlock Prevention

In 2PL with Non-waiting Deadlock Prevention (NO_WAIT), the DBMS uses a simple condition to prevent every situation where a deadlock might possibly occur: if a transaction is unable to obtain a lock, it is immediately aborted. The requesting transaction is not allowed to wait to acquire the lock.

4.1.2 2PL with Waiting Deadlock Prevention

In 2PL with Waiting Deadlock Prevention (WAIT_DIE), which is a variant of the NO_WAIT scheme, a transaction is only allowed to wait for a transaction that holds the lock it needs if the current transaction is newer than the transaction that currently holds the lock. If the requesting transaction is older, it is aborted and forced to restart.
4.2 Timestamp Ordering & Variations

Timestamp ordering concurrency schemes are based on the idea of generating a serialization order of transactions for the DBMS to enforce. Each transaction is assigned a unique monotonically increasing timestamp before it is executed which is then used by the DBMS to process transactions in the appropriate order. The variations of T/O presented below differ in the granularity that the DBMS checks for conflicts and when the DBMS checks for conflicts.

4.2.1 Basic Timestamp Ordering

In Basic Timestamp Ordering (TIMESTAMP), a transaction attempts to read or write to a tuple in the database, the DBMS compares the timestamp of this transaction to the timestamp of the last transaction that accessed this tuple. For any read or write operation, the DBMS rejects the request if the timestamp of the current operation is less than the timestamp of the last write to that tuple. Similarly, the DBMS rejects a write operation if the transaction’s timestamp is less than the timestamp of the last read to that tuple. Each aborted transaction is assigned a new timestamp and restarted. Further, a separate read query makes a local copy of the record to enable repeated reads due to the absence of locks.

4.2.2 Multi-version Concurrency Control

In Multi-version Concurrency Control (MVCC), every write operation creates a new version of the tuple in the database tagged with the timestamp of the transaction that created it. The DBMS is responsible for maintaining an internal list of all the versions of every element. When a read operation occurs, the DBMS determines which version of the element needs to be accessed which ensures a serializable ordering of all operations. The key benefit of MVCC is that a read operation that tries to access an element that has already been overwritten is not rejected.

4.2.3 Optimistic Concurrency Control

In Optimistic Concurrency Control (OCC), the DBMS is responsible for tracking the read/write sets of each transaction and storing all of their write operations in a private workspace. Whenever a transaction commits, the system determines whether its read set overlaps with the write set of any concurrent transactions. If there is no conflict, the DBMS applies the write changes from the transactions’s private workspace to the database. If there is a conflict, then the transaction is aborted and restarted. The key benefit of this approach is that transactions access shared memory only briefly when they commit.

5. YCSB Benchmark

The Yahoo! Cloud Serving Benchmark (YCSB) is a benchmark that simulates workloads that are representative of large-scale database services used by companies. Each YCSB tuple consists of a primary key column and 10 additional columns. The DBMS creates a hash for the primary key. Each transaction in this workload accesses 16 records of the database, where each access is either a read or a write; the transactions do not perform computations on the data. Further, all of the queries are independent from one another – where the result of one query is not used as input to another. The records accessed in the YCSB workload follow a Zipfian distribution which is varied by a parameter $\theta$. This parameter varies the amount of contention in the database.

6. System Design

The starting point for our implementation is the DBx1000 code. The DBx1000 code uses a CPU simulator that could scale up to 1024 cores. The DBMS is a main memory OLTP engine with basic functionality. We extracted the client and server out of the DBx1000 code and implemented a centralized coordinator in order to deploy our system to many machines. As a result, there are three distinct roles in the system: the client, DM, and the centralized coordinator. In our system there may be many clients and data managers, but there may be only a single coordinator.

6.1 Roles

In our system a single node can either be a data manager a coordinator, or a client. We describe these three roles below.

6.1.1 Coordinator

The centralized coordinator is responsible for enforcing the serializability of all of the transactions. Its role is to act as the mediator between a client and a database node. When a client sends a transaction to the coordinator, the coordinator unpacks the individual data requests and treats the transaction as if it were to be executed locally. Once any constraints required by the par-
ticular concurrency scheme are satisfied, (e.g., a lock is required, the current timestamp is greater than the last recorded timestamp), the coordinator hashes the request key and forwards the request to the corresponding DM. After all the requests in the transaction are processed, the results are aggregated and returned to the client. A request that cannot be processed by the coordinator without violating the constraints of the particular concurrency control scheme will cause the transaction to be aborted and restarted.

6.1.2 Data Manager
The data manager stores a partition of the entire database. It serves requests sent to it from the coordinator. From its perspective, all requests originate from the same node. The data manager only handles read and write data accesses and is unaware of transaction processing.

6.1.3 Client
The client creates transactions from the YCSB workload. The client is in charge of repeatedly sending transactions to the coordinator and examining the result of the transaction.

6.2 Communication
Because many machines were involved in our system, we had to provide a mechanism for communicating among the machines. We chose Thrift [7] to act as the RPC mechanism in our system (see Figure 1).

7. Experimental Analysis
The system was deployed to twenty machines on Amazon’s Elastic Cloud Computing (EC2) platform. One machine was dedicated to the coordinator and the other machines became either clients or data managers. All of the machines were 64-bit M1 General Purpose Large machines with 7.5GB of RAM, and 840GB of disk space. For testing purposes the number of data managers varied from [1,2,4,8,15] and the number of clients remained fixed at four. That is, five iterations were run for each concurrency scheme with the YCSB workload in which one EC2 machine was the coordinator, four EC2 machines took on the role of clients, and \( i \in [1, 2, 4, 8, 15] \) machines became data managers. EC2 machines were started and stopped using Python scripts deployed from a master node. The master node was an external machine which established communications among the clients, coordinator, and data managers. The output of each run was logged to a file and a Python script copied the log files from the EC2 instances to the master machine (see Figure 2).

The YCSB benchmark has a number of parameters that can be varied to produce different workload variations. In our analysis, we set the \( \theta \) parameter (see Section 5) to 0.6. This value of \( \theta \) translates to a moderate level of contention in the database. Specifically, it means that a hotspot of all the tuples in the database are accessed by 40% of the transactions. We set the read-write mixture to be 50% reads and 50% writes. In our experiments, there are 16 record requests in each transaction and 504 transactions per client.

8. Results
The graph in Figure 3 compares the throughput of the five concurrency control schemes for increasing
numbers of DMs. The 2PL schemes outperform the T/O schemes in terms of both scalability and throughput. In 2PL, processing a request requires a direct access or modification of a record in the database. This means that if the transaction does not abort, the number of RPCs issued to the DMs is equal to the number of requests in the transaction. NO_WAIT performs better than WAIT_DIE until the scale exceeds 4 DMs. The WAIT_DIE throughput peaks at 8 DMs and then decreases after this point. We expect that NO_WAIT would have the best performance if we were to keep increasing the number of DMs past 15. Although NO_WAIT has a higher number of aborts than WAIT_DIE, the abort cost relatively low since the set of transactions that need to be undone can be sent to each DM in a single RPC.

Out of the three T/O variants, OCC and TIMES-TAMP are the least scalable. Both TIMESTAMP and OCC keep local copies for read operations of records that requested from the DM. OCC also maintains the set of local record copies that a transaction modifies which must be sent to the DMs to be updated if the transaction passes the “validate” phase. The additional data accesses required by these algorithms increase the number of RPCs sent to the DMs. In the case of OCC, the number of RPCs issued to the DMs is twice the amount required by the 2PL algorithms. Further, the performance of the OCC scheme suffers from a high abort cost since transactions are aborted only after all requests are finished and validated.

MVCC, the third T/O algorithm, scales slightly better than OCC and TIMESTAMP. This is because MVCC creates and maintains a new version of a tuple each time it is modified. If a transaction needs to access a record that is present in this versioning set, it can do so locally without requesting it from the DM. MVCC also has the added benefit that “old” reads (i.e., read requests with timestamps older than the last write timestamp) are never aborted.

9. Discussion

Our experiments show that 2PL schemes are more scalable than T/O schemes and the trends in the graph are similar to those in the original DBx1000 paper for the same value of $\theta$ and read-write mixture. In particular, we found that the T/O schemes did not scale well due to the high overhead of sending additional requests to DMs since they require local copies of data.

The throughput rates from our experiments are lower than we expected. We believe that these low rates are a
result associated with the overhead of using Thrift as our RPC library. Although Thrift’s server code is optimized to handle a large number of concurrent connections, the client code is surprisingly hard to make multithreaded. The issue is that each thread needs its own socket because Thrift’s transport buffers are not thread-safe. This means that an experiment with 4 threads in the coordinator and 8 DMs requires 32 sockets (8 per thread). Thrift is not known for its performance, but rather for its ability to define and create cross-language services. Before conducting any future evaluations on this system, we believe that using a more appropriate communication mechanism would help to increase the throughput and scale of the system.

10. Conclusion

As systems become increasingly more distributed, scalable concurrency control is crucial. To examine how current concurrency control schemes perform on distributed systems, we evaluate five concurrency control schemes for OLTP workloads on a multinode system with a variable number of nodes. Five concurrency control schemes were tested on a cluster of nodes. Our analysis finds that all five concurrency schemes fail to scale on a multinode system though variants of two-phase locking perform scale better than variants of timestamp ordering.
11. Future Work

There are numerous extensions to this work. Further modifying our current system to be decentralized and then evaluating the same schemes would likely uncover new bottlenecks. For example, a decentralized system would need a to use a consensus protocol, like Paxos, which has the reputation of being slow. Finally, an extension is to implement and evaluate Calvin, a system that supports fast distributed transactions for partitioned databases [2], would also be interesting to compare with our current algorithms. Another algorithm could be implemented and evaluated would be 2PL with deadlock detection (DL_DETECT) which uses a waits-for graph to detect deadlocks and abort and restart transactions.

References


