Optimistic Concurrency Control in Geographically Distributed Databases for Concurrent OLTP Transactions

Lukas Miedema
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
p.l.miedema@student.utwente.nl

ABSTRACT

The need for distributed, scalable databases is larger than ever. Applications handling user-generated data like Twitter and Facebook have a global presence, but require performance characteristics from their database as if it was close to the user. Current database solutions either only offer low-latency reads by replicating the data, or skip ACID compliance by letting transactions do dirty writes with the possibility of overwriting the work of others. To solve this, we propose a way of using optimistic concurrency control (OCC) between nodes in the network. Every node holds a full copy of the data, but takes ownership over only a specific partition of the data that is geographically close to the users who may need to write to it. To evaluate this approach, a distributed database testing system is implemented, simulating latency between the nodes representative of various real-world network sizes. On this database our solution is then compared to an implementation of two-phase locking (2PL) using the TPC-C benchmark.

Keywords

Transactions; Optimistic Concurrency Control; Geographically Distributed

1. INTRODUCTION

Databases are typically used by many users at the same time. These users, however, must be allowed to treat the database as if they were the only user. It is up to the database to solve any problems caused by this. Such an interaction with the database is called a *transaction*. The technique used to solve these problems is called a *concurrency control* technique. For example, imagine two users adding money to the same account. User 1 adds €20, user 2 adds €30. The database typically does not support an atomic *increment* operation but instead requires both users to read the value, increment it in application code, and write a new value. A possible interleaving of the writes ($w_n(x)$) and reads ($r_n(x)$) could then look like this: $r_1(x), r_2(x), w_1(x), w_2(x)$.

If there was no concurrency control, user 2 would overwrite the value x written by user 1 without having seen it. If for instance the original amount in the account was €100, both users have read €100, user 1 has incremented it with €20 and written €120, which user 2 has then overwritten with €130. The end result is €130 in the account, whereas €50 was added. This behavior is unacceptable for certain applications, especially in finance. An *ACID* [5] compliant database may never allow such interference.

Isolation, the *I* in ACID, is violated in the example above. Isolation requires that the interactions from one user with the database must appear as if they are the only user of the database. The result of multiple users interacting with the database concurrently should not differ from the same users interacting with the database sequentially. If user 2 in the example above were to perform a read after $w_1(x)$, it would have read a different value than at the start of its transaction. This is a *non-repeatable read*. Likewise, the write $w_2(x)$ is a *dirty write*. Both non-repeatable reads and a dirty writes are signs of an isolation violation, as they would not have occurred if the transactions had been executed sequentially. It is the task of the concurrency control mechanism to generate a *serialization*, which is a sequence of transactions ensuring that the current state is the same as the state that would be obtained if the transactions were executed one by one in that sequence.

Various concurrency control techniques exist to solve these problems, like two-phase locking (2PL) and optimistic concurrency control (OCC).

Two-phase locking.

2PL is a commonly used technique where locks are used to ensure transactions do not interfere. Locks are only released when the transaction terminates. Two-phase locking is characterized by differentiating between read locks (read "phase") and write locks (write "phase"). When the write lock for an object is not held, any transaction can obtain a read lock. The read lock can be held by many transactions at the same time. The write lock, however, can only be obtained by one transaction, and while it is held no transaction can obtain a read lock. 2PL is vulnerable to deadlocks, where two transactions wait for each other to complete. A common strategy to resolve deadlocks is to detect them and restart one of the offending transactions.

In the previous example, transaction 1 and 2 would obtain a read lock on x for $r_1(x)$ and $r_2(x)$ respectively. Before any transaction can write to x, it will have to wait until all read locks on x are released. In this example, that results in a deadlock, as both transactions wait for the each other's read locks to be released on x. Transaction 1 may be aborted, releasing the read lock on x. Transaction 2 can now obtain the write lock and complete. Next, transaction 1 can restart and finish contention-free. The effective order

27th *Twente Student Conference on IT* July 7th, 2017, Enschede, The Netherlands.

with 2PL would be: $r_2(x), w_2(x), r_1(x), w_1(x)$.

2PL ensures serializability by ordering the transactions as they commit and blocking any transaction that conflicts until the conflicting transaction completes. When two or more transactions cannot be serialized like this, 2PL will detect the deadlock and restart one of the transactions involved in the deadlock.

Optimistic Concurrency Control.

Another common technique is Optimistic Concurrency Control (OCC), which takes a central role in this paper. OCC typically consists of three phases:

1. Startup phase. Here the transaction obtains a snapshot of the current state of the database.
2. Transaction phase. The transaction can read and write to its own snapshot.
3. Commit phase. The read and write actions done by the transaction on the snapshot are compared to the current state of the database. If no data was written or the data read has not changed, the changes are integrated into the current state. If this is not the case, the transaction is either aborted or restarted.

During the transaction phase, all reads done by the transaction are tracked in the set $S' \subset S$, where S is the snapshot taken in step 1. If none of the fields in S' have been written to by other transactions, the writes from this transaction can be integrated into the new state, placing it at the end of the current sequence of transactions. When there are no writes, the transaction can always be placed at the point in time where the transaction started. When there are reads to places that have been written to, like in the banking example from before, the transaction cannot be serialized and it must be rejected.

This paper introduces a *geographically distributed optimistic concurrency control* (*GDOCC*) method which makes use of OCC to achieve faster response times in geographically distributed databases. Because OCC requires no synchronization except for the final part of the transaction, it is no problem to start work with an older snapshot of the data. If the most recent data cannot be accessed (for instance because it is stored across the Atlantic), but a 50 milliseconds old version is available, the transaction can start anyway using this as a snapshot. Additionally, every node controls the write to a partition of the data. Transactions which need data controlled by the node they are connected to, do not need to wait for updates from other nodes. By placing the control over the data strategically at the node closest to a user who might need it, the average latency will be minimized and as such it should reduce the number of rejected transactions. When a transaction does need access to data from more than one node, an algorithm is required to make sure that all nodes agree on the interleaving of the transactions, and to validate that there are no conflicts.

TPC-C.

The TPC-C benchmark is used to test and compare the performance of various concurrency control methods. TPC-C is an Online Transaction Processing Benchmark (OLTP) which simulates a typical workload of a database. The TPC-C benchmark models a set of warehouses and interactions within the warehouses.

1.1 Problem Statement

Classical methods of concurrency control cannot efficiently be implemented in distributed databases. Even when traveling at the speed of light, the round-trip latency between Amsterdam and Los Angeles is approximately 60 milliseconds. When adding the overhead introduced by network switches and other equipment along the way, the latency becomes unacceptable to acquire locks as is done with the 2PL concurrency control method. As such, other methods concurrency control techniques are needed.

In this paper, we suggest a new way of using OCC (GDOCC) for increased concurrency in geographically distributed databases, which is then compared using the TPC-C benchmark with 2PL.

In section 4, the benchmarking framework is introduced on top of which our 2PL and OCC variants will be implemented. Section 5 and 6 discuss the distributed 2PL and OCC variants respectively. In section 7, preliminary results are shown using a microbenchmark. Section 8 discusses the implementation of TPC-C and the final results. Finally, section 9 presents the conclusion and looks at further possible improvements of the GDOCC method.

1.2 Research Questions

The following questions will be addressed:

1. How can two-phase locking be implemented on a distributed database?
2. How can GDOCC be implemented?
3. How does GDOCC compare to two-phase locking in a distributed database?
4. At which scales and latencies does GDOCC perform best compared to two-phase locking?

2. RELATED WORK

Concurrency control techniques have long been researched. Chiu et al. [4] combines optimistic concurrency control in a distributed system. However, Chiu et al. do not replicate the entire database to every node. Instead, all reads for data not present on the node the transaction is run on need to pass through the network, which differs from the approach we take. It also utilizes properties of OCC by having transactions read potentially-inconsistent values, however, the implementation differs wildly. They let transactions see each other's uncommitted work. If transaction T_1 has seen uncommitted work from transaction T_2 and T_2 is aborted, T_1 will be restarted as to not violate isolation and atomicity.

Bernstein et al. [1] put forth a framework for comparing distributed database concurrency control mechanisms and compare variants of two-phase locking with timestamp ordering concurrency control. *Timestamp ordering concurrency control* is similar to 2PL, but instead of locking it rolls back any conflicting transaction. Bernstein et al., however, do not look at optimistic concurrency control.

Other methods for concurrency control also exist. Wevers et al. [7] introduce a completely different method of concurrency control. They use lazy evaluation and have the readers of transactions perform the actual transaction. This amortizes the cost of bulk transactions over longer periods of time without blocking other transactions. However, the paper does not cover methods for scaling the technique and takes a white box approach to transactions. The database needs to be able to have full control over the execution of the transaction and how they execute.

Boral et al. [2] also take a white box approach to transactions with the Bubba project, executing them in a distributed database on the node that stores the data to reduce the strain on the network. Boral et al. introduce a parallelizing compiler which allows large portions of consumer applications of the database to be directly executed on the database's nodes.

3. METHOD OF RESEARCH

The first and second question are answered by means of a literature research in combination with the creation of new techniques. For the third and fourth question, an implementation is created of both 2PL and of the GDOCC method. Finally, a framework for testing has been produced which can simulate a set of nodes that communicate with a set latency to mimic network delays introduced by geographical distance.

A microbenchmark is introduced to find which parameters are interesting to measure. Then, based on the result of the microbenchmark, TPC-C will be run with selected parameters. All transactions from the TPC-C benchmark are implemented. Every warehouse entity and its related data in the TPC-C benchmark is stored on its own node. With TPC-C, 1% of the orders from a warehouse have to be retrieved from other warehouses, which is where the inter-node communication is put to the test. Simulations of a network with various latencies are tested. This data will then lead to answers to the research questions.

All benchmarks are executed via the Java Microbenchmark Harness (JMH). JMH is part of the OpenJDK project specifically for performance testing. It provides extensive facilities to ensure minimal interference with any benchmark. All tests are executed on an Intel i5-3750K-based desktop system with 16GB of memory, running OpenJDK 1.8.0_131. The tests are compiled with `kotlinc` v1.1.2-5. and JMH 1.19 (both via their respective Maven plugins). The number of concurrent transactions varies per test.

4. MEASUREMENT AND BENCHMARK FRAMEWORK

To answer the third and fourth question a benchmark framework is needed. This framework simulates a fixed number of nodes interacting with messages. It also provides a set of primitives that are used by the implementations of 2PL and GDOCC. Transactions come in at one node but may read, write and modify data stored on any node via the same API. The nodes exchange messages via channels that simulate latency. The framework itself is not actually distributed - latencies are only simulated and the whole benchmark runs in a single application. The data held by the database is never persisted - it only exists in-memory for the duration of the benchmark.

4.1 Channels

The nodes communicate with each other via *Channels*, an abstraction which provides first-in-first-out (ordered) message passsing between nodes. An implementation may set up multiple channels between any two nodes, and communication may only happen via the channels. The channels facilitate *Remote Procedure Calls* (RPC) - a technique where one node can run code on another node, and then do something with the returned value. Delay is introduced both before the Remote Procedure Call is invoked to simulate the call traveling to the other node, and again after the call to simulate the response traveling back. There is no restriction on how much data is sent or returned, and this does not affect the latency. This simulates a "fat long pipe" type of connection, where there is ample bandwidth but, due to the distance covered, a high latency.

4.2 Common API

All concurrency control techniques implement a set of interfaces via which database operations can be performed. This is done such that no benchmark is specifically written for any of the concurrency control algorithms, but rather every benchmark can run on every implementation by changing which `DatabaseFactory` is used. Each concurrency control technique implements four interfaces:

DatabaseFactory to create instances of the database.

Database is the product of the *DatabaseFactory*. Its API only provides a list of nodes.

Node is where transactions are submitted to.

TransactionManager (or *TM* for short) is the class transactions interact with. It contains methods like *set*, *insert*, *get*, among others. A new `TransactionManager` is created for every transaction, and it handles the lifecycle of the transaction. After a successful commit the TM is destroyed. Transaction managers themselves are not thread safe, as they assume the transaction is running on just one thread.

No database implementation is specifically written with the schema and structure of the tables of the benchmark in mind, nor are they only limited to those structures. Instead, the tables are defined at runtime before the benchmark begins. Multiple tables can be defined, each with a primary key. Table records ("rows") are represented by classes and the primary key by a class that implements the Java `Comparable` interface. When defining the table, a special bridge function must be supplied which can take an instance of the record class and extract the primary key from it. Optionally, a key generator function can be supplied which facilitates the generation of new primary keys for the *insert* operation. Without this, *insert* is not supported and any transaction must supply its own primary keys instead of having the database generate them. The tables and the number of nodes and latencies are set at creation time via the *DatabaseFactory*. After instantiation there is no way to change the schema or the number of nodes. The benchmark framework does not simulate nodes going offline or new nodes coming online during the benchmark, as those aspects are not relevant to concurrency control.

In order to support the rerunning of transactions, the thread starting a transaction offers a function which represents the transaction. This function is then started by the database implementation, and possibly rerun. However, the transaction is always executed on the thread that started the transaction and will not be moved to another thread. This is to simulate a typical database application, where the application and the database itself are separated. The application has a client-side library (for instance a JDBC driver in case of Java), which communicates with the database. The actions the client-side library can take are limited: it can only block the execution of the transaction on an API call, and it can restart the transaction. No other operations are supported. It cannot for instance move the code of the transaction to another node for execution, or perform some kind of static analysis on the code to determine if the transaction is read only.

As such, any database implementation may only do two things to a transaction: *restarting* and *blocking*. Aborting and restarting transactions happens by throwing an exception from any method invocated on the transaction manager, or waiting until the transaction returns control to the transaction manager at the end of the transaction. *Blocking* refers to suspending the transaction thread until some condition becomes true, for instance to wait until a lock is acquired or until a message is received by the transaction manager.

4.3 TPC-C Benchmark

The TPC-C benchmark is a mixture of read-only and write intensive transactions which simulates a large business using a database. As mentioned, TPC-C models a retailer with multiple warehouses. Each warehouse serves 10 districts, 100,000 customers per district and all their orders. TPC-C has a total of 9 tables and 5 transactions which run at various frequencies. TPC-C is very suitable to test the performance of a distributed database with, as its model lends itself well to being distributed. Every record, besides the read-only *Item* record, has a warehouse id (`w_id`) of the warehouse it belongs to in its primary key. By assigning a warehouse to every node, the transaction manager can easily determine where a record is stored by looking at the requested primary key. A full implementation of the TPC-C benchmark tests a wide breadth of functionality around the database, including column storage and disk I/O. Not all of these features are applicable in this study, and as such only a small part of TPC-C is implemented.

5. TWO PHASE LOCKING

To test 2PL, an implementation needs to be created capable of running on the distributed environment laid out in section 4. Deadlock detection and resolution, features inherent to 2PL, become more complex when no single node has the full picture. In our 2PL implementation, every node has its own set of channels to other nodes called *probe channels*. These channels are solely used for deadlock detection between the nodes. When a transaction starts, its accompanying transaction manager obtains a *transaction id* and an independent set of channels to every other node. These channels are used for reading and, during commit, writing to other nodes. The assigned transaction id consists of a sequential id generated by the originating node and the node id, ensuring that every transaction id is globally unique. The channels are not shared with other transaction managers, such that if another transaction manager's remote procedure call takes a long time to process this transaction manager is not blocked by it.

Two-phase locking is implemented by means of a first-in-first-out lock queue on every records' primary key. Transactions can request either a read lock or a write lock on a record. They will be appended to the lock queue. The algorithm for determining which locks are granted is as follows (simplification):

```
val iterator = queue.iterator()
if (!iterator.hasNext())
  return

val head = iterator.next()
grant(head)
if (head.type == WRITE)
  return

for ((index, request) in iterator) {
  if (request.type == WRITE) {
    if (index == 0 && request.transactionId ==
    head.transactionId) {
      grant(request)
    }
  } else {
    grant(request)
  }
}
```

In words: the head of the queue is always granted, regardless if it is a read or write. If it is a write, the algorithm stops there and no other lock requests are granted. If the head is a read, all consecutive reads are also granted until the first write request is encountered. However, if the second lock request is a write request from the same transaction as the head, it is also granted as it is an upgrade request (a transaction does not have to wait for its own lock to be released). A write lock completely supersedes a read lock - if a write lock is granted on a given value, a read lock is implicitly also granted. Note that this algorithm assumes no read locks follow write locks from the same transaction. This is ensured by simply not adding a read lock request to the queue when a write request from the same transaction is already present.

The choice to have a first-in-first-out style lock queue has certain consequences. Take the following queue: $Q = \{r_1, r_2, w_3, r_4, w_5\}$. In this case, the above algorithm would grant r_1 and r_2, but r_4 has to wait because there is a write ahead of it. A read-write lock implementation could give r_4 priority over the write lock w_3, but this could lead to an extended period of waiting and starvation if new read requests keep on coming in. The other extreme is to grant write locks priority, moving them together behind all granted read locks. In that case, w_5 would be moved ahead of r_4. This change makes sense in a read-heavy environment, where writes are rare and potentially slow. However, TPC-C is no such environment where 92%[6] of all operations are read-write operations. As such, the decision was made for a first-in-first-out queue approach.

5.1 Distributed deadlock detection

As mentioned, 2PL is very susceptible to deadlocks, especially because it allows "lock upgrading" from a read lock to a write lock. Deadlock detection is rather simple in a centralized system, as the complete state information is present. However, in this implementation of 2PL no node has the full picture. As such, an adaptation of Chandy et al.'s [3] distributed deadlock detection algorithm was necessary. The proposed algorithm makes use of independent channels for deadlock probe messaging. Once it has found a deadlock, it aborts a transaction. The adaptation for use with 2PL is illustrated in Figure 1.

In Figure 1 we see one transaction manager *TM(A)* which is associated with node A. There is also a node B which stores the value on which TM(A) wants to obtain a lock. Note that node A and node B could be the same node - in that case the latency between TM(A) and node B would be zero. Actions in white are part of the normal processing of the request. Actions in gray are part of the deadlock resolution algorithm, which is only shown partially. Messages passed between nodes in black are over the Transaction Manager's own channel, those in blue are over the deadlock probe channels.

With 2PL, transactions wait for each other's completion when they are dependent on each other. A deadlock occurs when a transaction is effectively waiting for itself. An intuitive formulation of Chandy et al.'s deadlock detection algorithm is that all deadlocks can be detected by having the transaction manager send a probe message when it starts waiting to all transaction managers it is waiting for. Those transaction managers determine if they themselves are waiting. If they are, they in turn relay it to the transaction managers they are waiting for. If the message comes back around to the starting transaction manager, it knows there is a dependency cycle and can abort.

In the 2PL implementation, transaction managers themselves are not responsible for detecting their deadlocks, but instead their associated node is. Additionally, trans-

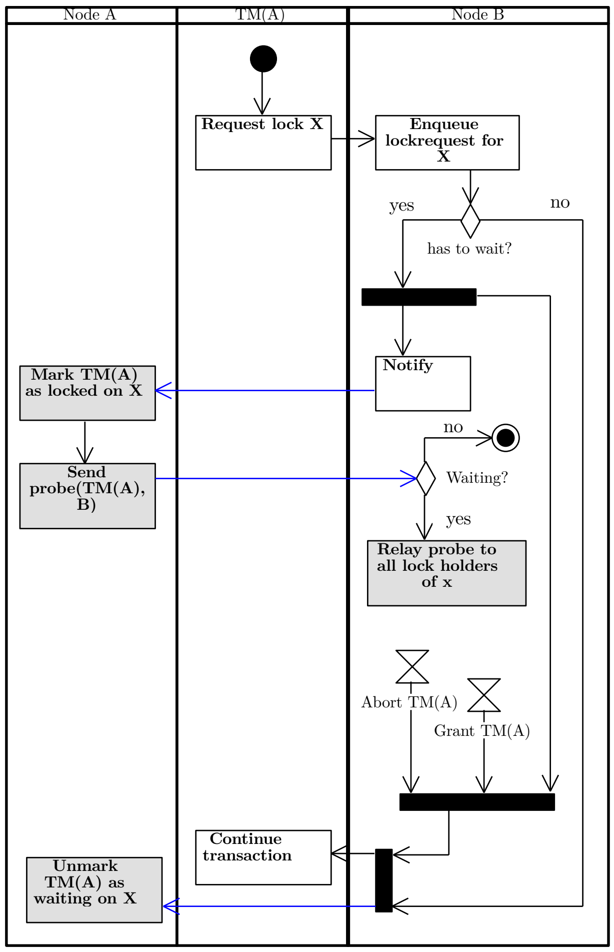

Figure 1: The process of acquiring a lock with 2PL. The process of relaying a probe is not shown.

action managers do not know which transaction managers are ahead of them in the lock queue - only the node holding the lock queue knows this. As such, the adaption of this deadlock detection algorithm has a few extra steps. A few of these steps can be seen in Figure 1, namely the marking as locked of TM(A) and the beginning of sending a probe message. Missing are the steps where probe A either relays a message from a value an associated transaction manager is waiting for, and the step where probe A aborts a transaction because the probe message went in a cycle.

5.1.1 Preventing livelocks

It was found during early testing that Chandy et al.'s [3] algorithm can have multiple transaction managers detect the same deadlock at the same time. This happens because a transaction manager sends a probe message when the deadlock does not yet exist, however due to network latency the deadlock is formed while the probe is in transit. This means two probes, one that happened to already be in transit, and one send out by the lock request that created the deadlock, manage to complete the cycle. In turn this causes two transaction managers to detect the deadlock and restart. After these restarts the transaction managers potentially end up in exactly the same deadlock again. This deadlock is once again detected twice, and the whole cycle repeats. This is a *livelock*, where both transaction managers aren't waiting for each other, but aren't progressing either. This problem was found to typically occur with two transaction managers hosted on two different nodes with high latency in between. Both transaction managers try to lock on a value on their own node and then a value on the remote node. Since both transaction managers can access their own nodes very quickly, they obtain a lock on the value on their own node before the other transaction.

Livelocks are fragile - after a while they are resolved due to random interleavings in the execution of the transaction. However, it was found that when this happend during a benchmark it would seriously degrade the performance of 2PL and could take a very long time to resolve. As such a solution was required. Livelocks can be prevented by making sure only one transaction manager in any given cycle can determine the deadlock. This is done by *probe upgrading*. When a transaction manager starts, it obtains an assigned transaction id. These transaction ids have a meaningless but consistent order based on the `sequenceId` and `nodeId` - a combination which is guaranteed to be unique. When a transaction manager encounters a deadlock probe from a transaction manager which is ordered "lower" than itself, it will not relay it but instead replace ("upgrade") it with its own deadlock probe. When every transaction manager follows this procedure, only the transaction manager with the highest transaction id will detect the deadlock and restart, preventing a livelock. This does incur a minor performance penalty. The time complexity increases from $O(n)$ of the algorithm without probe upgrading to $O(2n)$, where n is the number of nodes. The worst case doubles because the probe may have to go around the deadlocked dependency graph twice when the highest-ordered transaction manager is dependent on the transaction manager that sends the final deadlock probe.

6. OPTIMISTIC CONCURRENCY CONTROL

Where 2PL does not replicate the data across the network, our OCC implementation does. With 2PL, replication provides no benefits as the communication about who owns the lock takes just as long as transferring data guarded by it across the network, so they might as well be bundled. Any value read via the 2PL implementation would be an actual, committed, consistent value. Our OCC implementation on the other hand can let transactions read partial commits or otherwise inconsistent values, and as long as it does not allow such a transaction to actually commit, it does not violate isolation. It does not show uncommitted values, which is where it differs from the work of Chiu et al. [4]. Additionally, Chiu et al. do not choose for replication.

Besides holding a copy of the whole database, every node takes authority over a certain partition of data, and only that node is allowed to publish changes to the data. This retains the benefit that 2PL had: transactions which exclusively need data hosted on their associated node do not need to wait for the network. What is different is that every node additionally has this potentially-outdated copy of the rest of the database. These copies may be inconsistent - if we imagine a transaction which increments every record in the database, it could be that this change is not visible yet in every partition of the dataset on a particular node. The optimistic transaction manager is allowed to offer this potentially inconsistent data to the transaction. When the transaction manager has offered this data to the transaction, the transaction manager needs to contact the other node in the commit phase to check for consis-

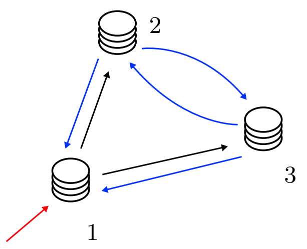

Figure 2: Messages sent during a multi-node commit originating from node 1

tency. All the nodes involved in the transaction need to agree that the data is fine, that no dirty reads occurred before the data may actually commit. This costs (measured in time) at most ($O(t) = max(RTT_{transaction})$), where ($RTT_{transaction}$) is the set of all round trip times to nodes involved in the transaction. During the transaction itself, the transaction manager performs no writes to any node, and only reads from its associated node.

6.1 Commits

OCC has most of the complexity in the commit phase. No messages are exchanged between the nodes about transactions prior to the commit phase. The recipe to a successful commit is rather simple:

1. Partition all the reads and writes done by a transaction into `RequestToCommit` messages
2. Send these `RequestToCommit` messages to all nodes involved, including the originating node even if no reads or writes occurred there
3. Then, on every node involved
   (a) Obtain locks to all the values either read or written to in the `RequestToCommit`
   (b) Validate the `RequestToCommit` and send a *clear to commit* signal to every other node involved in the commit
   (c) Wait for a *clear to commit* signal from every other involved node
   (d) Apply the commit to the database
   (e) Release all locks

Steps *c* and *d* are skipped in case any node reports a validation failure. In that case, the node executing the original transaction will create a new `TransactionManager` and run the transaction again.

To facilitate the processing of a transaction on a receiving node, the node will instantiate a `CommitMananager`, which is responsible for handling the `RequestToCommit` on the node that received it. This means that in the case of a multi-node transaction, there will be multiple `Commit-Managers` active on different nodes, processing the same transaction.

In Figure 2 it is shown how the commit's message passing plays out of a transaction that modifies data on multiple nodes. The reads and writes are split up by the originating node (node 1) and sent to the target nodes at once, including a list of other nodes involved (black arrows). If the transaction was found to be valid, every node sends a message to every other node that the transaction is "clear to commit" (blue arrows). All nodes enter the `WAITING_FOR_CTC` mode, where they wait for the *clear to commit* (ctc) signal. In this mode, a *commit lock* has been

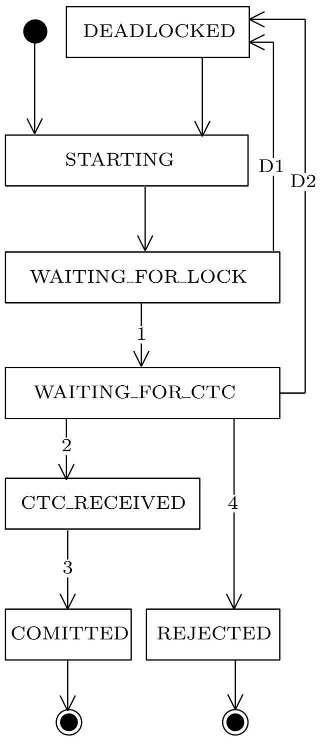

Figure 3: The states of the `CommitManager`

obtained on all the values read and written by the transaction. The nodes confirm that the transaction was valid. When a node concerned in the transaction has received this message from every other node, it commits the data and releases the commit locks. The data is now committed. Finally, every node sends the newly committed data to the other nodes to complete their view of the entire database. Although locks are used for atomicity, the locks are not held during the transaction. A slow transaction cannot starve the database, which can happen with 2PL.

6.2 Detecting deadlocks during commit

Commit locks are at risk of deadlocking - when one transaction commit acquires commit locks on node N_1 and waits for N_2, but a competing transaction holds a lock on N_2 and waits for N_1, the system is deadlocked. This is a typical example of left-right deadlocking.

To detect and abort this type of deadlocking, Chandy et al.'s [3] distributed deadlock detection algorithm is used once again.

The addition of deadlock detection and resolution leads to the `CommitManager` states shown in Figure 3. On transition 1, validation occurs. When the transaction is valid, the `CommitManager` moves to the `WAITING_FOR_CTC` state, where it waits for a clear to commit signal from all nodes as previously discussed. When it has received clearance from all nodes involved it moves to state `CTC_RECEIVED` where the transaction is applied to the database. The transaction is now comitted. In case a *reject to commit* is received from any of the nodes, the `CommitManager` immediately jumps to the `REJECTED` state.

There are two paths via which a `CommitManager` can be aborted due to a deadlock, which are labeled *D1* and *D2* in Figure 3. When the `CommitManager` itself is directly part of a deadlock cycle, it will be notified in state `WAIT-ING_FOR_LOCK` (D1). Once all the locks are acquired, this `CommitManager` cannot directly cause a deadlock anymore, but one of its companion `CommitManangers` running on another node can still be part of a deadlock. When that is the case, this other `CommitManager` cannot have performed validation yet, so all other `CommitManagers` are still waiting for the *clear to commit* signal. As such, we find them in the `WAITING_FOR_CTC` state, where they will be notified of the deadlock (D2).

When a `CommitManager` is notified of a deadlock, it does not abort. Instead, it simply releases any locks that it had already acquired and repositions itself in the `STARTING` state, which is illustrated by the transition after `DEADLOCKED`.

Deadlocks are much rarer with OCC than with 2PL. With 2PL, a deadlock can occur on different values on the same node. This is not possible with OCC, as all reads on a given node are handled by the same `CommitManager`. This `CommitManager` requests all the locks atomically. As such, deadlocks only emerge when two or more multi-node transactions have read or written to the same values. Additionally, 2PL suffers greatly from *upgrade* deadlocks, where two transactions have a read lock and both want to obtain a write lock. Because OCC only obtains locks during the commit phase, where it already has the full picture, it never needs to upgrade. Finally, deadlocks are often cheaper to deal with in the case of OCC. Instead of restarting the transaction, only the associated `CommitManagers` have to be restarted.

7. MICROBENCHMARKS

7.1 Measurements

First, a simple microbenchmark called *transfer test* is presented which attempts to be somewhat representative of what the TPC-C benchmark would do. This microbenchmark is used to determine what aspects of TPC-C are interesting to look at, as TPC-C is complex and takes a long time to benchmark.

The transfer test microbenchmark is inspired by TPC-B, a now discontinued performance benchmark. The microbenchmark conducts money transfers between bank accounts. It consists of one table with a fixed number of bank accounts initialized before the benchmark starts. During the benchmark, it repeatedly picks two accounts from this table and an amount. If the amount is available in account 1, it will then deduct it from account 1 and add it to account 2. Finally, it stores these accounts again and repeats. The account selection is completely uniform, and during the benchmark no new accounts are created or deleted. Because the account selection is uniform, most transactions will involve more than one node, and often not even the home node from which the transaction originates.

The transfer test microbenchmark is tested with three parameters:

Round-Trip Latency

- **Global** with < 300 ms latency
- **Continental** with < 50 ms latency (18% of Global)
- **Regional** with < 10 ms latency (18% of Continental)
- **Datacenter** with no channel-induced latency

Number of nodes $\in \{5, 25\}$
Number of accounts $\in \{100, 200, 300, 400, 600, 800, 1000\}$

There are 56 unique combinations for these parameters.

7.1.1 Round-Trip Latency

Round-trip latency is set in a way that is modeled after real world Internet latency. The 25 largest metro areas were selected, whose latitude and longitude have been used to compute latency using Vincenty distance. After measuring internet latency to various servers around the globe, a message speed of $s = 1.40 \cdot 10^8 m/s$ was established. This speed, in combination with the Vincenty distance between any two cities, is used to create the final latency. The *Continental* and *Regional* tiers are created by simply scaling the global tier latencies by 18% and 3.24% (18% of 18%) respectively, in order to not change the shape of the graph and by extend introduce other factors that can affect the outcome of the tests. The tests with 5 nodes are the 5 largest cities from the test with 25 nodes.

Java Microbenchmark Harness was used for this test, with 10 warmup iterations and 10 test iterations, both taking 1 second per iteration. The JVM was given 4GB of memory.

7.1.2 Deadlocks

Unfortunately, it was found that both implementations would occasionally still get in a proper deadlock, where deadlock detection and resolution failed. A cause of these deadlocks could not be determined, but it is assumed to be a simple programming oversight somewhere in the implementations, and not a theoretical problem. As such, the tests where this occurred where rerun until results could be obtained with a deadlock. These deadlocks occurred primarily in the high contention tests with low latencies.

7.2 Results

See Figure 4 and Figure 5 for the number of operations per second compared to varying levels of contention. Additionally, see Figure 6 for the number of deadlocks and restarts for the *global* tier.

7.3 Discussion

The datacenter scale tells an interesting story. 2PL is fastest, however its performance does depend on the number of nodes. Both OCC and 2PL appear to perform better in a datacenter on *more* nodes instead of less. This is probably due the overhead of internal locks used inside each node. Actions on a table segment are guarded by a single lock. This pattern is used both in the GDOCC implementation and in the 2PL implementation. Because the throughput here is very high, the effects of contention on a table segment increase as these segments contain a larger portion of the data.

There's a striking similarity between the other graphs. Besides the y-axis, all three graphs are practically identical. This means that both GDOCC and 2PL scale similarly over the latency size, at least in the interval tested. This makes sense - the amount of round trips doesn't dramatically change by increasing the size of the network. The transactions in the test are really short, as such a restart is very cheap. This means that OCC does not suffer greatly from increased contention, and that restarts are less likely because the time between a read and a commit is short.

A similar effect is observed with the number of nodes. Increasing the number of nodes decreases performance approximately linearly. This can be best explained by the uniform random nature of the benchmark. With 5 nodes there's a 20% chance an account is stored on the originating node, whereas with 25 nodes this drops to just 4%.

If we look at Figure 6 we see that OCC takes the crown with the number of restarts. OCC however rarely deadlocks, which was expected. With 2PL, a restart and a

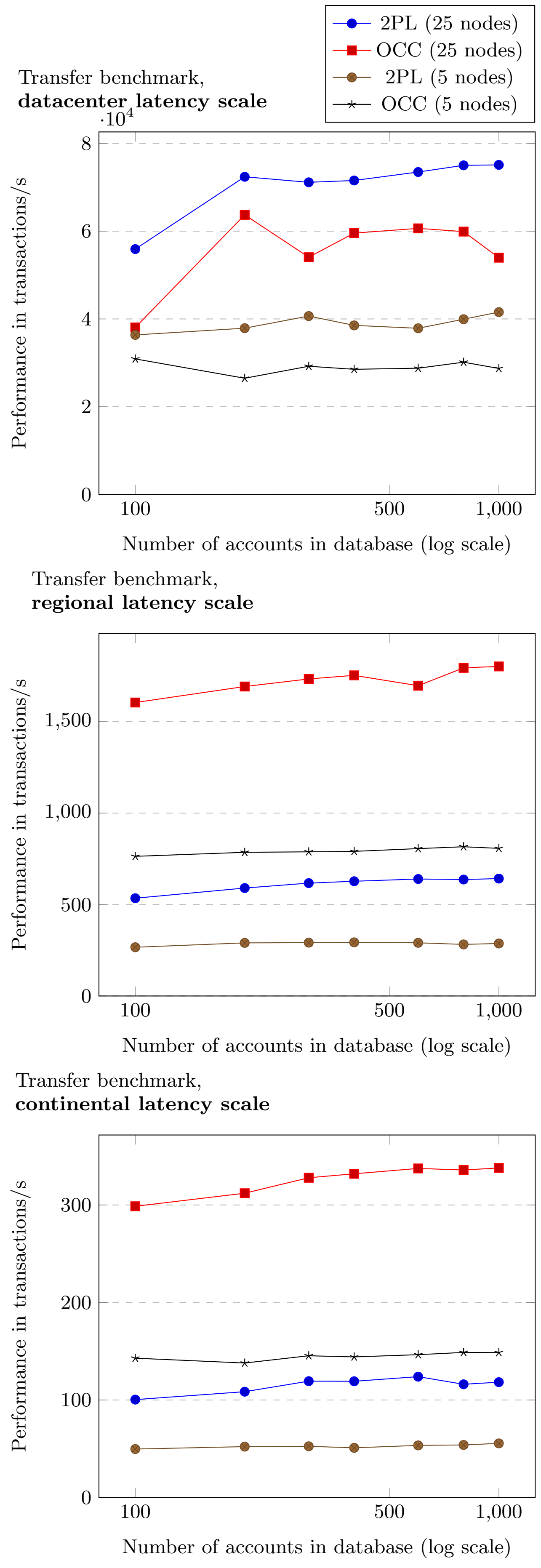

Figure 4: Results of the transfer test with datacenter, regional and continental latency scales.

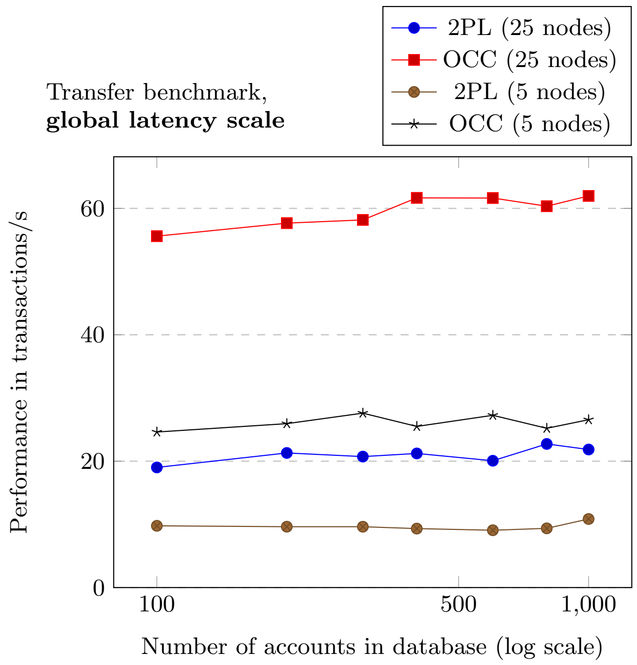

Figure 5: Results of the transfer test with global latency scale.

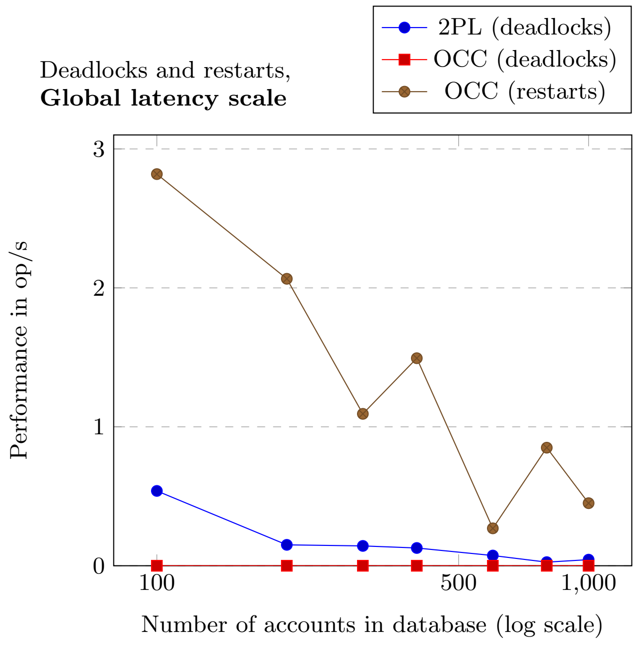

Figure 6: Restarts and deadlocks during the transfer test at global latency scale. With 2PL, a deadlock always causes a restart, which is why 2PL restarts aren't shown.

deadlock always go hand in hand, as the deadlocks happen during the transaction instead of after it as is the case with OCC.

As a result of these findings, the final TPC-C benchmark will be conducted with a fixed number of nodes. 10 latencies will be tested, each in steps of 80% of the latency of the previous step.

8. TPC-C BENCHMARK

8.1 Adaptation

TPC-C is a benchmark testing all aspects of a database system. However, our implementation of 2PL and GDOCC is not a full database system implementation. They provide only a few data access operations, namely the *set*, *get* and *insert* operations. Some of the transactions done by TPC-C were adapted so that they could be executed using just those operations. TPC-C uses sometimes secondary keys in two of the five transactions, and always a secondary key in one transaction. Secondary keys however are not supported by the testing framework. Additionally, the minimal runtime for a proper TPC-C benchmark is 8 hours, of which at minimum 2 hours must be spend benchmarking. Because neither database implementations support storing data to disk, and all TPC-C transactions create more records than they delete, this would prevent such a long test from being executed.

As such, the five TPC-C transactions have been adapted to be able to work on the implemented databases. Special attention has been put into making sure the data access patterns were left as much unchanged as possible. TPC-C measures it performance in `tpcC`, which is the throughput of the five transactions measured in how many *orders* - a business-level concept inside TPC-C - were completely processed, from start to end. Some of the adaptations necessary however make it impossible to completely process each order. As such, the results of this benchmark are instead reported in operations per second, where an operation is any successfully completed transaction. The transactions are selected at random with a probability are listed in Table 7.

Figure 7: Rates of the five TPC-C transactions

Transaction	Probability
New Order	45%
Payment	43%
Order Status	4%
Delivery	4%
Stock level	4%

These values are also an adaption from the TPC-C benchmark. The TPC-C benchmark specification only provides minimum rates for each of the transactions, besides *New Order*. The rate of *New Order* is used to compute the systems score in `tpcC`, which is not possible in this benchmark.

The `ITEM` table in TPC-C is read-only, and both the *New Order* and *Payment* use this table. Our benchmark framework has no concept of a read-only table, and as such these records would need to be stored on one node with all the locking and communication overhead it incurs either during a read/write or during commit for 2PL and GDOCC respectively. To spare this overhead, the `ITEM` table has been amended with a new column - `w_id`, the warehouse id. A full copy of the 100,000 items are stored on every node, just with a different `w_id`.

8.2 Parameters

With data obtained from the *transfer test*, parameters were devised for the final TPC-C-based benchmark. More granularity in the levels of latency is needed. 10 levels of latency have been selected, ranging from the *Global* latency tier as it appeared in the transfer test down to approximately 13% of the global latency, in decrements of 80%. During early testing it was found that TPC-C's initialized database with more than 10 nodes is too large to fit into the 8GB of memory allocated for it. As such, all tests have been done with 10 nodes.

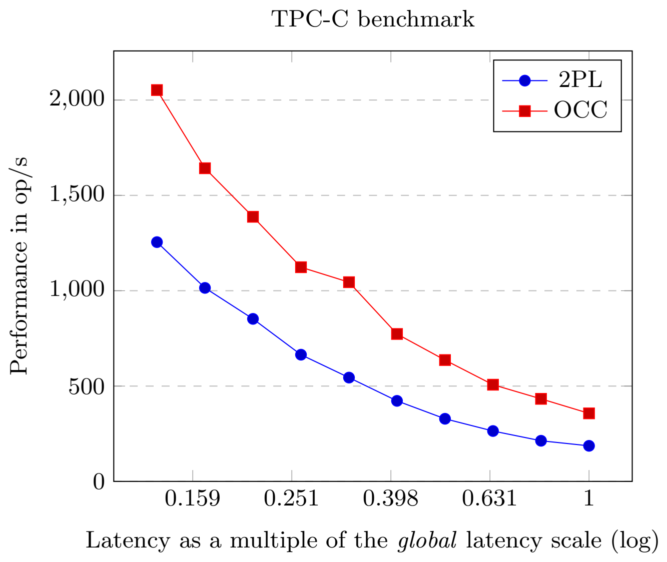

Figure 8: Performance of TPC-C vs latency

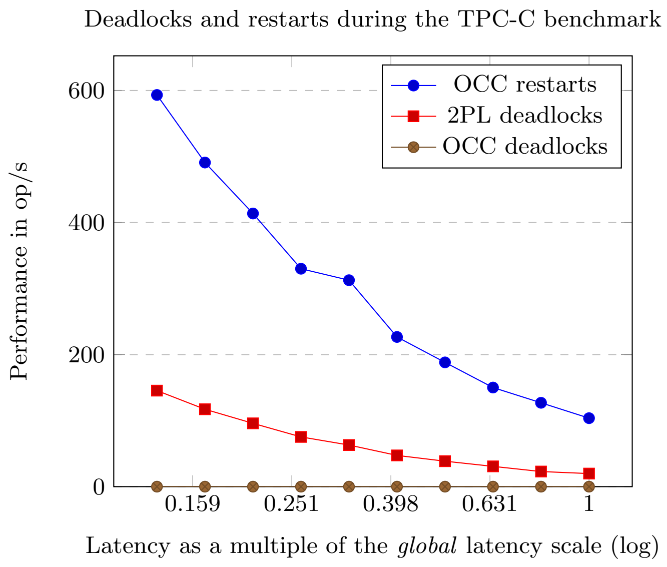

Figure 9: Deadlocks during TPC-C vs latency. Note that with 2PL a deadlock always causes a restart, which is why 2PL restarts aren't shown.

Java Microbenchmark Harness was configured with 8 benchmark threads, 10 warmup iterations each taking one seconds and 10 measurement iterations of 10 seconds each. Iteration synchronization was once again disabled. The JVM was given 8GB of memory, 4GB more than in the previous test, with `-XX:+AlwaysPreTouch` enabled. `+AlwaysPreTouch` makes sure the memory pages given to the JVM at startup are "touched", which forces the operating system to clear them ahead of time which would otherwise happen on demand.

No unrecoverable deadlocks like those occuring during the transfer test occurred during the TPC-C test.

8.3 Results

Results from the TPC-C benchmark can be seen in Figure 8 and Figure 9. GDOCC clearly performs better in distributed transactions than 2PL, especially under extreme latencies. Even with the much more involved transactions of the TPC-C benchmark, GDOCC still pulls ahead.

9. CONCLUSION

This paper has introduced a new way of leveraging Optimistic Concurrency Control features in a distributed environment, and shows its effectiveness against 2PL. It shows that it outperforms 2PL over all tested latencies, but the differences become more pronounced as network latencies increase. Even with the more involved TPC-C transactions the costs of a restart is still lower than the costs associated with the long waits of 2PL.

There are also downsides to using GDOCC over 2PL. Transactions need to be programmed defensively - even if for instance every transaction always inserts both a `PhoneNumber` and an associated `Employee` record, with GDOCC it can happen that a transaction reads the `PhoneNumber` from its associated node, but the `Employee` from an outdated copy of another node which does not yet contain this record. GDOCC is ACID compliant, but it is a bit of a stretch in the definition of ACID. The transaction described above would never commit - it would be rejected during the validation phase. However, the transaction itself must be written in a way capable of safely reading these erroneous values.

9.1 Future work

Various features and solutions have been worked out for the GDOCC method, but could not be implemented due to time constraints and scope set for this study.

9.1.1 Abnormal commits and foreign keys

Our GDOCC implementation does not support *foreign keys.* Foreign keys are a type of constraint where one record has to refer to another record. This record must exist at all times. Taking the `PhoneNumber` and `Employee` problem from the previous paragraph - if these are inserted by two transactions, and there is a foreign key from the `PhoneNumber` record to the `Employee` record, how does the transaction creating the `PhoneNumber` know if the `Employee` exists? When it does not exist in the copy that it has of the node that should store the `Employee`, can it safely raise an exception? No, because its copy might be outdated. As such, it would need to perform an *abnormal commit.* The transaction manager initiates a commit, but instead of sending `RequestToCommit` messages with the work done by the transaction, it uses the commit mechanism to validate if the constraint really is violated. If this commits, it knows that yes, the constraint was violated so it can safely raise an exception aborting the transaction. If it does not commit, it restarts the transaction as usual.

9.1.2 Secondary keys and iterators

As mentioned, secondary keys are currently not supported. Secondary keys are a nuisance - in 2PL they cannot simply be implemented by obtaining a lock on all values that map to the key. Another transaction might add a new value matching the same secondary key, which should have been seen by the transaction manager. This same problem also exists for GDOCC, during the validation phase it is not enough to simply check if the records retrieved are unchanged, but after the use of a secondary key the `CommitManager` must also check if there aren't any unseen records matching the secondary key. For TPC-C this would be enough - all secondary keys still contain the `w_id` - the *warehouse id*, via which the hosting node can be determined. However, for more sophisticated benchmarks like TPC-E, this is not enough. Records do not have a "home" node and the place where they are used the most can change during the TPC-E benchmark. As such, either every node needs to be asked to validate a certain query, or a special shared "index" must be maintained, much like ordinary records are maintained now. During the commit phase, the `CommitManager` must also ask the node storing the index if that bit of the index was unchanged.

9.1.3 Select and fetch

TPC-C often speaks in terms of *select* and *fetch*, which are two distinct phases in retrieving a value. The first indicates intention to read something, but does not yet request the actual value. The second is actually used to retrieve the value. In case of 2PL, supporting these as separate operations would speed up certain transactions. When a 2PL transaction wants to obtain two values from far-off nodes, it could do a *select* on both of them so that the transaction manager can provision the required locks concurrently.

9.1.4 Concurrent validation

GDOCC can be further optimized by utilizing *concurrent validation.* When an OCC transaction reads a value that is not owned by the node, the node may see an update for this value come in before the transaction tries to commit. At that point, the node can restart the transaction without incurring any overhead on the network. Our implementation does not do this, and instead will still send a `RequestToCommit` to the node owning the value, which will then be rejected forcing a restart.

10. REFERENCES

[1] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, June 1981.

[2] 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 Transactions on Knowledge and Data Engineering*, 2(1):4–24, Mar 1990.

[3] K. M. Chandy, J. Misra, and L. M. Haas. Distributed deadlock detection. *ACM Trans. Comput. Syst.*, 1(2):144–156, May 1983.

[4] L. Chiu and M. T. Liu. An optimistic concurrency control mechanism without freezing for distributed database systems. In *Data Engineering, 1987 IEEE Third International Conference on*, pages 322–329. IEEE, 1987.

[5] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, Dec. 1983.

[6] P. Tözün, I. Pandis, C. Kaynak, D. Jevdjic, and A. Ailamaki. From A to E: analyzing TPC's OLTP benchmarks: the obsolete, the ubiquitous, the unexplored. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 17–28. ACM, 2013.

[7] L. Wevers, M. Huisman, and M. van Keulen. Lazy evaluation for concurrent OLTP and bulk transactions. In *Proceedings of the 20th International Database Engineering & Applications Symposium*, IDEAS '16, pages 115–124, New York, NY, USA, 2016. ACM.