# Low-overhead Transaction Conflict Detection for Key-Value Storage Workloads

Aleksei Vasilev
Technical University of Munich
alex.vasilev@tum.de

## ABSTRACT

Traditional database systems guarantee the correct execution of concurrent transactions. These systems were mostly designed in a time period when the amount of data was not as big as today. Due to these reasons, it is very hard and expensive to achieve horizontal scalability with them. On the contrary, NOSQL database systems were designed to scale well on distributed systems and to operate on petabytes of data. However, in order to perform this, NOSQL database systems either give up transactional support or only offer transactions on the parts of the database. For example, Cassandra provides atomicity and isolation within a single partition on a single node [1]. This, of course, assumes, that applications using the database only produce a workload which works on parts of the database at a time.

The main obstacle for achieving both scalability and transactional support is concurrency control. With hundreds of clients running in parallel, it becomes more and more complex to coordinate competitive access to the data on different nodes. In this paper, we show that using optimistic concurrency control schemes can help to overcome this issue and to extend the transactional support to more than one database node without significant throughput decrease. We adapted the TicToc concurrency control algorithm [11] to the distributed system. Our results show that the algorithm scales well and the main bottleneck of performing communications between nodes during the transaction is eliminated because of the exploitation of more parallelism.

## 1. INTRODUCTION

This paper consists of three main parts: introduction, method and implementation, results. In the first section of the introduction, we briefly describe existing concurrency control algorithms. The second section of introduction provides more insight into the original TicToc algorithm and the third section gives examples of transactional guarantees in several existing NOSQL databases.

In the second part of the paper, we first describe the benchmark and the workloads that we used for evaluating the performance of our solution (Sections 2.1 and 2.2). Sections 2.3 and 2.4 provide the technical aspects of the implementation of our solution as well as of the transactional support. And finally, in Section 3 we present the experimental results achieved by our solution.

### 1.1 Concurrency Control

Online transaction processing DBMSs provide end users (clients) the opportunity to read and modify data by sending requests. A transaction in the context of one of these systems is the execution of a sequence of one or more operations (e.g., SQL queries) on a shared database to perform some higher-level function [4]. Transactions execution have to follow certain ordering constraints. In the highest isolation level (i.e., serializable), the execution schedule must be equal to a schedule with all the transactions executed sequentially. A concurrency control algorithm in OLTP DBMSs allows executing operations of simultaneous transactions so that th data is modified in such a way as if each transaction is running exclusively on the database. Concurrency control algorithms can be separated into two main categories: *p*essimistic and *o*ptimistic.

Pessimistic strategies assume that collisions between transactions will arise very often and thus acquire locks on the database entities from the time they were firstly accessed till the end of the transaction. Two-phase locking (2Pl) is an example of pessimistic concurrency control algorithm which was the first method proven to guarantee correct execution of concurrent transactions [2]. There can be two types of locks: read lock and write lock. Write locks prevent other transactions from reading and modifying the entity. An entity with a read lock can be read by other transactions but is not expected to be modified. If a transaction is unable to acquire the necessary lock on an element it is forced to wait until the lock becomes available. If waiting is not controlled, then deadlocks can occur. So the way to avoid deadlocks is the main difference between different Pessimistic algorithm variants [10].

In contrast with Pessimistic schemes, Optimistic Locking assumes that conflicts between different transactions will be infrequent. Instead of preventing collisions by locking, Optimistic algorithms try to detect collisions and resolve them when they occur. This is usually done by assigning timestamps to transactions and processing conflicts in the proper order.

DBMSs with optimistic concurrency control execute transactions in three steps: read, validation and write. During the read phase, a system performs reading and modification operations without locking. Usually, this is done by maintaining separate read and write sets for each transaction. Every entity that was read by the transaction is copied to the read set. All updates and writes are made only to the write set and not to the original data storage. Thus, they are visible only for their transactions.

After all read and update operations are finished by the

transaction, it starts the validation. During this phase, the conflicts with other transactions are being detected. If there are no conflicts or they can be easily resolved the transaction starts the write phase and propagate all changes to the original storage and makes them visible to the other transactions. Otherwise, the transaction is being aborted which means that its read and write sets are cleaned and the DBMS client receives the abort signal.

The main difference between Optimistic algorithms is in the validation phase in the way of resolving conflicts. Many solutions were proposed to reduce the abort rate and increase the throughput. We next discuss the TicToc algorithm which calculates the transactions timestamp *l*azily at their commit time in a distributed manner based on the tuples they access.

## 1.2 The TicToc Algorithm

The TicToc algorithm [11] uses timestamps to provide the serial order of transactions. But instead of assigning timestamps using centralized allocator, it computes timestamps at commit time. In the original TicToc paper [11], it was shown that this approach avoids bottlenecks connected with timestamp allocation (using a unique increasing timestamp for every transaction can become a bottleneck with a high number of transactions [8]) and moreover exploit more parallelism in the workload, reducing aborts and increasing throughtput.

Original OCC schemes use static timestamps assignment to transactions, which means that only a fixed sequential schedule for concurrent transactions can be accepted. This approach helps to detect conflicts easily, but at the same time limits concurrency.

By contrast, TicToc does not use static timestamps allocation and thus can accept different potential orderings of transactions. TicToc computes the timestamp lazily for each transaction at its commit time by inspecting only accessed tuples.

Serialization information in the tuples is encoded by the write timestamp (wts) and the read timestamp (rts). A particular version of the tuple is created at write timestamp and is valid until read timestamp. A version of the tuple that was read by a transaction is valid if and only if the transactions commit timestamp is in the range of versions wts and rts. And a write to a tuple is valid if and only if the transactions commit timestamp is greater than the rts of the previous version of this tuple. These rules lead to the serializable execution of transactions since all transactions writes and reads occur at the same timestamp. A read always returns the version valid at that timestamp and a write is ordered after all the reads to older versions of the same tuple. Formal proof of the fact that the TicToc algorithm is able to correctly enforce serializability is presented in the original TicToc paper [11].

## 1.3 NOSQL Systems

The majority of existing NOSQL DBMSs give up transactional support in order to achieve high performance. However several distributed systems do provide some low guaranties to the users.

For example, Apache Cassandra database does not use ACID transactions with rollback or locking mechanisms but instead provides atomic, isolated, and durable transactions with eventual/tunable consistency. Cassandra offers write operations that are atomic at the partition level, which means that in one partition insertions, updates or deletes of two or more rows are handled as one write operation.

To determine the most recent update to a column Cassandra uses client-based timestamp assignment. So if the same tuples are updated by multiple client sessions concurrently, the readers will see only the most recent update.

Cassandra write and delete operations also provide full row-level isolation within a single partition, meaning that on a single partition a write to a row is only visible to the client performing the operation.

MongoDB provides atomic operations only on a single document (analogs of rows in traditional databases) [5]. MongoDB also offers isolated operator. Using it, write operations on multiple documents can prevent other processes from interleaving once the write operation modifies the first document. This means that no client sees the changes until the write operation completes or errors out.

However isolated operator does not work with sharded clusters and does not provide all-or-nothing atomicity. So an error during the write operation does not roll back all the changes within the transaction that preceded the error.

There is also a way to implement two-phase commit protocol and even a rollback-like functionality using MongoDB API. However, this will lead to a dramatic performance decrease in case of transactions accessing documents on multiple nodes, since the number of communications between nodes (that is usually done by the network and thus too expensive) will increase.

In the next sections, we show that usage of OCC scheme like TicToc can help to extend transactional support for distributed databases from a single to multiple nodes without significant throughput decrease.

# 2. METHOD AND IMPLEMENTATION

To evaluate the performance of the algorithm and to compare it with other existing solutions we decided to use Yahoo Cloud Serving Benchmark (YCSB), a common benchmark to evaluate distributed databases performance.

## 2.1 YCSB

YCSB is a framework and common set of workloads for evaluating the performance of different key-value and cloud serving stores [7]. It consists of two parts: *the YCSB Client*, an extensible workload generator and *the Core workloads*, a set of workload scenarios to be executed by the generator.

The Client has drivers that provide connection to the most popular NOSQL systems like MongoDB, Cassandra or HBase. However, it is easy to add a new database support, since the database interface layer in YCSB hides the details of the specific database from the Client. The database interface layer is a simple abstract class that provides read, insert, update, delete and scan operations for the database [9].

Also, the set of core workloads can be extended with self-defined scenarios, not adequately covered by the core workload.

However, YCSB does not provide transactional workloads. Thus we extended the original benchmark by adding startTransaction, commit and abort operations to the database interface layers and by creating a transactional workload, that will be described in the next section.

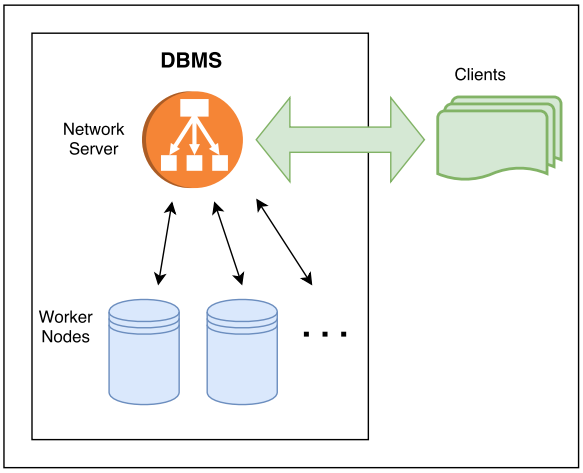



**Figure 1: Solution architecture**

YCSB core workloads operate on a table of ten text fields. But for our workload, we wanted to use long fields to simulate a common transactional scenario of moving "money" between bank accounts. Because of this we also added support for long values to the YCSB.

## 2.2 Workloads

The performance of our system and other competitors was tested with two workloads:

1. **Read-only:** Each query read a single tuple based on a Uniform distribution. For this workload, we use basic YCSB data layout with one key-value storage, where value consists of ten text records, each of ten symbols.

2 **Read-modify-update:** This workload implements transactional scenario, where each entry in the storage is a bank account with string key encoding account Id and long value with account's balance. During the transaction, the client read a specified number of records, randomly change the balances by moving "money" between accounts and update the records. Records are chosen uniformly. All updates are made atomically. To support this scenario original YCSB benchmark was extended by:

- adding support for long values
- adding startTransaction, commit and abort operation to database interface layer
- implementing defined scenario alongside with existing core workloads.

## 2.3 Key-Value storage

We implemented our own lightweight main memory key-value storage to evaluate the concurrency control algorithm performance. To simulate distributed system we used the architecture shown in Figure 1.

Our solution consists of several working nodes and one network server. The source code is avaliable at [6].

Each working node is a physical thread with its own data partition and a working queue with requests from the server. A working node can accesses only its own data and provide insert, update, read and delete operations. Since scan operations were not necessary for the task of evaluating concurrency control performance, hash-based indexing was used. As a hash function we chose Murmur Hash to distribute data uniformly between the nodes. However this choice does not affect the number of interactions between nodes during the transaction, thus the concurrency control algorithm we used can be applied to any other indexing strategies without changes in the performance. Data is distributed between nodes by the hash values of keys.

The network server accepts connections from YCSB clients via Unix Domain Sockets (UDS). Each client connection is assigned own session instance by the server. Messages between client and server were serialized using Google Protobuf framework.

A common interaction scenario between a client and our solution can be described as following:

**1.** Client tries to connect to the server via UDS.

**2.** Server establishes connection and creates new Session instance, the session starts waiting to request.

**3.** Client sends a request to the server (for example: read(key)).

**4.** Server deserializes the request from the protobuf message; maps the request to the proper worker node, based on the hash value of the key; posts the task to the worker node's working queue.

**5.** Worker node processes the task (read data from its data partition) and sends the result back to the server.

**6.** Server receives the result from worker node, serialize it to the protobuf message and propagate back to the client via UDS.

**7.** Client receives the result and can send a new request (step 3).

The network server works asynchronously, which means that while one session is waiting for the worker node response, the server can handle other connections from clients in the same thread. The server was implemented using boost asio library [3].

## 2.4 Algorithm

In this section, we show how TicToc algorithm can be applied to discussed architecture. The main difference between it and the architecture that was used by the TicToc paper authors is that in our architecture each process can only access its own part of data. This means that to validate transaction worker nodes should communicate with each other which is also true for real distributed ower network systems.

### *2.4.1 Original TicToc*

As discussed in Section 1.2, the original TicToc store read and write timestamps for each record to encode the serialization information. It's transaction consists of three phases: read, validation and write.

Like standard OCC algorithm, during the read phase, TicToc accesses database records without acquiring locks. Accessed records are copied to the read set, modified - to the write set. Each tuple in both write and read sets encoded as {tuple, data, wts, rts}, where tuple is a pointer to the tuple in the database, data is the data value of the tuple, and wts and rts are the timestamps copied when the tuple was first accessed by the transaction.

When the transaction invoke commit operation, TicToc starts the validation phase. During this phase, TicToc inspects whether the transaction can be committed or should be aborted. It computes transaction's commit timestamp and starts the write phase if the transaction can be committed.

```
Data: read set RS, write set WS
# Step 1 – Lock Write Set
for w in sorted(WS) do
    lock(w.tuple)
end
# Step 2 – Compute the Commit Timestamp
commit_ts = 0
for e in WS ∪ RS do
    if e in WS then
        commit_ts = max(commit_ts, e.tuple.rts +1)
    else
        commit_ts = max(commit_ts, e.wts)
    end
end
#  Step 3 – Validate the Read Set
for r in RS do
    if r.rts < commit_ts then
        # Begin atomic section
        if r.wts ≠ r.tuple.wts or (r.tuple.rts ≤ commit_ts and
        isLocked(r.tuple) and r.tuple not in W) then
            abort()
        else
            r.tuple.rts = max(commit_ts, r.tuple.rts)
        end
        # End atomic section
    end
end
```

**Figure 2: TicToc validation phase**

```
Data: write set WS, commit timestamp commit_ts
for w in WS do
    write(w.tuple.value, w.value)
    w.tuple.wts = w.tuple.rts = commit_ts
    unlock(w.tuple)
end
```

**Figure 3: TicToc write phase**

The validation phase is performed within 3 steps (Figure 2): locking the write set, computing the commit timestamp, validating the read set. The first step prevents other transactions from updating the rows concurrently. Locking is done in primary key order to avoid possible deadlocks. The commit timestamp is computed according to the restrictions described in Section 1.2. Finally, TicToc performs validation of the read set and extend the records read timestamp where necessary if these records were not modified by other transactions and are not in the write set of the current transaction. Our results show that the algorithm scales well and the main bottleneck of performing communications between nodes during the transaction is eliminated because of the exploitation of more parallelism. Within the write phase, the algorithm propagates changes to the database. Records that participated in the transaction also change their write and read timestamp with the commit timestamp, computed during validation phase (Figure 3).

In the original TicToc paper [11] there is a detailed explanation and a formal proof of why this algorithm leads to the serializable isolation level of transactions execution.

```
Data: working set WS
# Step 1 – Lock Working Set
for w in sorted(WS) do
    lock(w.tuple)
end
# Step 2 – Compute the Commit Timestamp
commit_ts = 0
for w in WS do
    commit_ts = max(commit_ts, w.ts + 1)
end
# Communication between nodes
# Step 3 - and Validate the Working Set
for w in WS do
    if w.ts ≠ w.tuple.ts then
        abort()
    end
end
# Communication between nodes
```

**Figure 4: Distributed TicToc validation phase**

### 2.4.2 Distributed TicToc

In our implementation, we made several simplifications to the original TicToc algorithm due to the type of workloads that we use (Section 2.2). For the read-only workload, all read operations can be considered as one atomic operation since the tuples are not modified. Thus there is no need to implement specific transactional support for this workload. So the algorithm was implemented only for the read-modify-update workload.

In this workload, all tuples that were read by a transaction are then updated. Because of this there is no need to handle separate write and read sets and they can be combined into one *w*orking set. Moreover, there is no need to store write and read timestamps separately, since they will be always equal. So record in the working set can be encoded as {tuple, data, ts} with tuple as a pointer to the tuple in the database, data as the data value of the tuple and ts as the timestamps copied when the tuple was accessed by the transaction.

Despite the fact that these simplifications are very strict assumptions on the types of allowed transaction, they do not affect the amount of communication between the nodes in a distributed database and thus do not affect the scalability of the algorithm. To support every kind of transactions only functions inside the worker nodes should be modified.

The validation phase inside the worker node is shown in Figure 4.

In the first step, the transaction acquires locks on the tuples from the working set to prevent multiple transactions from updating one tuple concurrently. Locking is done in primary key order. In our implementation instead of locks we use unsigned values that encode the id number of session locking the tuple. To avoid deadlocks and unnecessary waiting if the transaction cannot acquire a lock on the tuple, it postpones the operation. This operation will be started again when the tuple will be unlocked. To achieve this logic, worker nodes store postponed operations in a special map with id number of session holding the lock on the operation

```
Data: working set WS, commit timestamp commit_ts
for w in WS do
    write(w.tuplle.value, w.value)
    w.tuple.ts = commit_ts
    unlock(w.tuple)
end
```

**Figure 5: Distributed TicToc write phase**

as the key. When the session finishes write or abort the operation, all locks on tuples it accessed should be released. That means that now operations postponed by this session can be repeated.

The second step of the validation phase is computing the commit timestamp. Comparing with the original TicToc algorithm, our transactions read and write set are combined, thus the commit timestamp is computed as the maximum of tuples timestamps plus one. However, this will be the maximum only for timestamps within one worker node. Therefore there is a need for communication between worker nodes holding tuples participated in the transaction and the actual commit timestamp is the maximum between worker nodes commit timestamps.

After this communication working sets of each node can be validated. Again due to the type of the transactions we use for benchmarking, this process is simplified compared to the original TicToc algorithm. After validating worker nodes should communicate to check whether all of them are allowed to commit the transaction. If at least one worker node failed the validation step, the transaction should be aborted on all nodes. Aborting, in this case, means releasing all locks and clearing the working set.

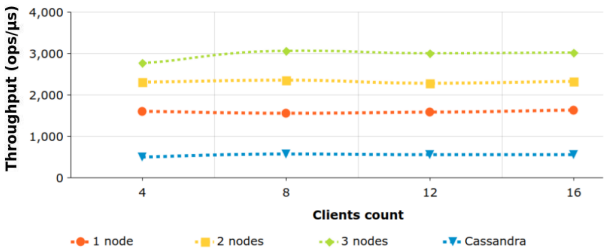
If all worker nodes passed the validation step, writing phase, shown on the figure 5, can be started. It consists of updating data, timestamp and releasing locks.

To reduce the number of communications, each transaction store identification names of worker nodes participating in it, since during the validation phase interaction only between these nodes is required. If a user knows that tuples accessed by the transactions are connected by some prior laws, data can be distributed between the worker nodes according to these laws and thus the number of inter-node connections can be maintained at a low level, increasing throughput.

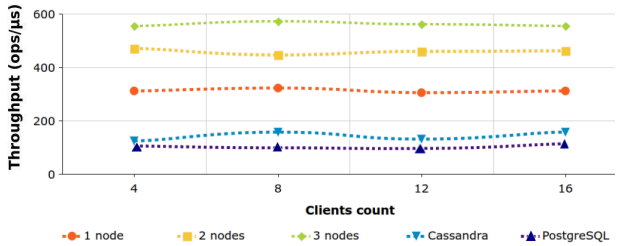
# 3. EXPERIMENTAL RESULTS

We now provide an experimantal evaluation of our solution under different YCSB workloads.

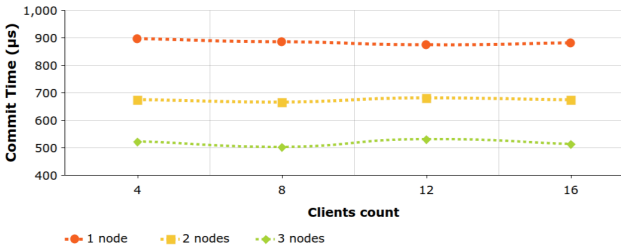
Fig. 6 (a) and (b) show the throughput of our solution with different amounts of worker nodes as well as of a local Apache Cassandra cluster and PostgreSQL DBMS. On the x-axis, four different numbers of concurrent clients session are displayed and on the y-axis, you can see the throughput in operations/microseconds. Fig. 6 (c) shows the relation of time required by our solution to perform commit operation (in microseconds) and a number of connected clients. Curves for our solution with 1,2 and 3 worker nodes are displayed. Fig. 7 provides the information on the scalability of our solution. It shows the dependency of throughput on the number of worker nodes. And finally, on Fig. 8 we provide




(a) Read-only workload throughput




(b) Transactional workload throughput




(c) Transactional workload commit time

**Figure 6: Experimental results**

the average abort rate of PostgreSQL and 3 configurations of our solution.

We evaluated the throughput not only for our solution but also for local Apache Cassandra and PostgreSQL installations. An important note is that YCSB connects to the Apache Cassandra within TCP sockets, whereas our solution utilizes UNIX Domain Sockets (UDS) instead. UDS have explicit knowledge that they're executing on the same system. Thus they avoid the extra overhead of managing connections over the network and perform faster than TCP sockets. However, we provide results for Apache Cassandra and PostgreSQL to show that our solution scales equivalently on a different number of concurrent client connections.

## 3.1 Read-Only workload

At first, we evaluated the throughput of our solution comprising read-only transactions. In this workload, the client read a record chosen from a uniform distribution. Since the tuples during this workload are not modified all read operations can be considered as one atomic operation and there is no need for communications between nodes as well as for acquiring locks. Such a workload provides a baseline before we explore more complex transactional scenarios.

Fig. 6 (a) shows that the throughput of our solution for every number of worker nodes is constant for the various


**Figure 7: Throughput measurements of our solution for different number of worker nodes**

number of client connections. This is an expected behavior since the same property is also true for Apache Cassandra and PostgreSQL. Fig. 7 (a) show that the throughput scales linearly with adding new worker nodes due to the fact that more operations can be processed in parallel. However, the scale factor is less than one because there is only one server processing clients connection and redistribute tasks across worker nodes.

## 3.2 Transactional workload

Fig. 6 (b) provides the performance of our solution on a transactional workload, described in section 2.2. Here as well as with the read-only workload throughput does not depend on the number of client connections. On Fig. 6 (c) we can see that the commit time is also independent of the clients count.

Analysing scalability (Fig. 7 (b)) we could expect that for transactional workload it would not be so good because, in contrast to read-only transactions, this workload requires communication between worker nodes and thus supposed to be less scalable. However, the throughput still increases linearly with new worker nodes, because operations become distributed and more client connections can be processed in parallel. This is the most desired property because it shows us that we can provide transactional support even on the distributed system with linear scalability. Fig. 6 (c) also shows that the time to perform commit operation decreases with increasing number of nodes. This is due to the fact that during the validation phase, all accessed records should be verified to compute the commit timestamp or abort the transaction. And with more worker nodes this job is more parallelized.

Finally, on Fig. 8 we demonstrate another nice property of our concurrency control algorithm: low abort rate. In contrast to PostgreSQL, that use MVCC algorithm, TicToc applies dynamic timestamp allocation and thus can accept more possible orders of the transactions. This leads to a higher number of committed transaction and reduces the abort rate.

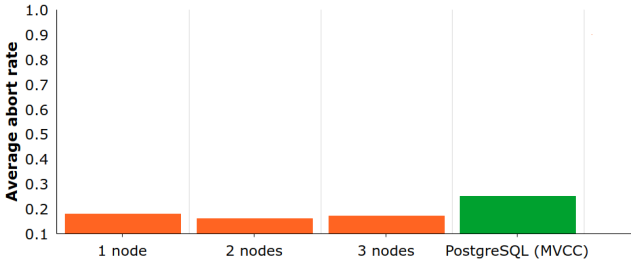



**Figure 8: Average abort rate**

# 4. FUTURE WORK

In our solution, the problem of providing transactional support for data with replications is uncovered. Usually, NOSQL systems cannot rely on the responsibility of a single node and thus have multiple replications of data on different nodes to provide data safety in case of software or hardware failures. To adapt our algorithm for this scenario several changes are required. During the read phase, the modifications should be applied to all replicas. Within the validations, phase records of the working set should also be locked for every replica. However, the commit timestamp can be computed only in one of the nodes as well as the third step of the validation phase. Finally, all changes should be propagated to all replicas to have consistent data.

# 5. CONCLUSIONS

In this paper, we studied how optimistic concurrency control schemes can help to provide transactional support for distributed database systems. We implemented our own lightweight main memory key-value storage. The architecture of our system closely simulated a real distributed system. To provide the transactional support we adapted the original TicToc concurrency control algorithm to our architecture in a specific way to reduce unnecessary communications between worker nodes. To evaluate performance and scalability of our solution, we used Yahoo Cloud Serving Benchmark. However, this benchmark did not support transactional workloads out of the box and thus we made several upgrades in it. Our results show that proposed concurrency control algorithm can scale well and the overhead of inter-node communications during the transaction execution is liquidated due to the parallelization of jobs.

# 6. REFERENCES

[1] Apache cassandra documentation. *https://docs.datastax.com/en/cassandra/3.0/.*

[2] P. A. Bernstein, D. Shipman, and W. Wong. *Formal aspects of serializability in database concurrency control. IEEE Transactions on Software Engineering.* 1979.

[3] Boost asio documentation. *http://www.boost.org/doc/libs/.*

[4] J. Gray. *The transaction concept: Virtues and limitations. InVLDB.* 1981.

[5] Mongodb documentation. *https://docs.mongodb.com/.*

[6] Key-value storage implementation. *https://github.com/VAlex22/MyDB.*

[7] A. Silberstein, B. F. Cooper, R. Ramakrishnan, R. Sears, and E. Tam. Benchmarking cloud serving systems with ycsb. *1st ACM Symposium on Cloud Computing*, June 2010.

[8] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. *SOSP*, 2016.

[9] Ycsb documentation. *https://github.com/brianfrankcooper/YCSB/wiki/*.

[10] X. Yu, A. Pavlo, G. Bezerra, S. Devada, and M. Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. In *Proceedings of the VLDB Endowment*. VLDB Endowment, 2014.

[11] X. Yu, A. Pavlo, D. Sanchez, and S. Devadas. Tictoc: Time traveling optimistic concurrency control. *SIGMOD16*, June 2016.