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

Aleksei Vasilev

Technische Universität München
Department of Informatics
Chair of Database Systems

Tuesday 5th December, 2017

Outline

- Introduction
- Goals and Objectives
- Related Work
- Benchmark
- Method
- Implementation
- Results

Introduction

- Traditional Database Systems
  - Strong transactional guarantees
  - Not scalable
- Modern NOSQL Systems
  - Horizontal scalability
  - Give up transactional guarantees

Goal

Extend transactional support to more than one database node without significant throughput decrease using optimistic concurrency control schemes.

Concurrency Control

- Concurrency control - ensure that multiple transactions are executed simultaneously
- Concurrency control - pesimistic and optimistic
- Pesimistic
  - Assume that collisions between transactions will arise very often
  - Acquire locks on the database entities during transaction exceqution
- Optimistic
  - Assume that conflicts between different transactions will be infrequent
  - Try to detect collisions and resolve them when they occur

Optimistic Concurrency Control

Three steps to execute transaction:

- Read
- Validation
- Write

Maintain separate read and write sets for each transaction

Original TicToc algorithm

- Compute a transactions timestamp lazily at commit time based on the data it accesses
- Can accept different potential orderings of transactions
- Serialization information in tuples - write timestamp (wts) and read timestamp (rts)

Original TicToc algorithm

- Transaction can be commited if:
  - For all read tuples: wts $<=$ cts $<=$ rts
  - For all written tuples: rts $<$ cts
- Read phase - access tuples, copy them to the read/write set
- Validation phase:
  - Lock write set
  - cts $=$ max(max(wts) from read set, max(rts)$+$1 from write set)
  - Validate the read set
- Write phase - propagate changes, unlock write set

NOSQL Systems

- Apache Cassandra:
  - Atomic writes within partitions
- MongoDB:
  - Atomic operations on a single document
  - Offers isolated operator
  - Isolated operator doesn't work with sharded clusters
  - Doesn't support all-or-nothing atomicity

YCSB

- Used to evaluate perfomance of key-value and cloud serving stores
- YCSB Client and common set of workloads
- Does not provide transactional workloads
- Was extended to support *startTransaction*, *commit* and *abort* operations

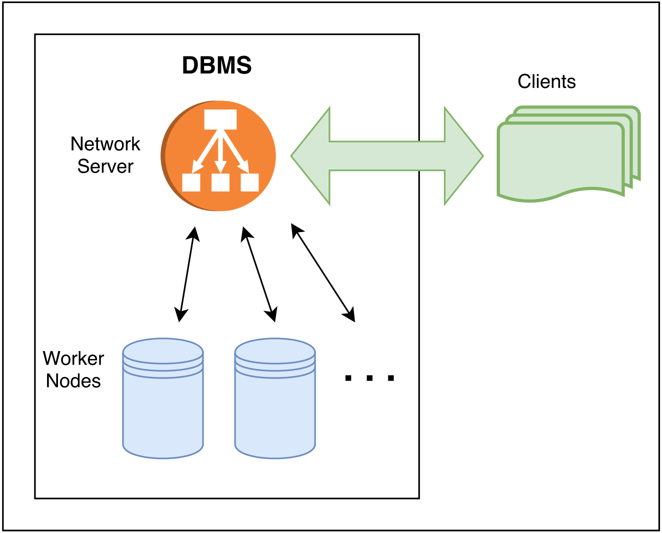
Workloads

- Read-only - each query read a single tuple based on a Uniform distribution
- Read-modify-update - each entry in the storage is a bank, transaction changes balances by moving "money" between accounts and update the records

Key-Value Storage

- 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
- Murmur hash-based indexing
- Hash-based data distribution between nodes
- Benchmark - Storage communications via UDS using google protobuf

Key-Value Storage



Algorithm

- Read and write sets are equal
- Read and write timestamps are equal

These simplifications were made due to the kind of chosen workloads. However they do not affect the amount of communication between the nodes in a distributed database.

Three phases to perform transaction: read, validation and write.

Read Phase

- During the transaction execution all accessed tuples are copied to the working set as {tuple, data, ts}
- All changes are applied only to the working set
- Each working node maintains a separate working set for every transaction

Validation Phase

```
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
```

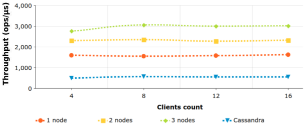
Write Phase

Commiting:

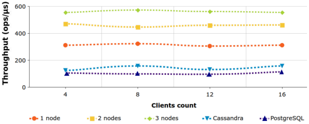
```
Data: working set WS, commit timestamp commit_ts
for w in WS do
    write(w.tuplle.value, w.value)
    w.tuple.ts = commit_ts
    unlock(w.tuple)
end
```

Aborting means releasing all locks and clearing working sets

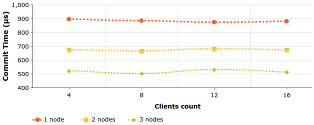
Results



(a) Read-only workload throughput

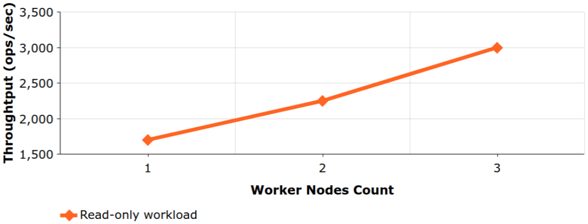


(b) Transactional workload throughput

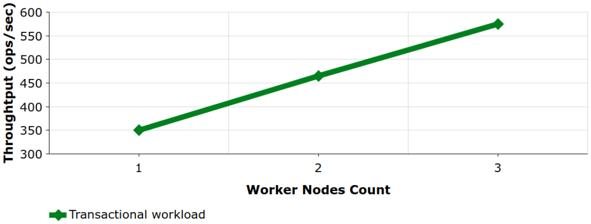


(c) Transactional workload commit time

Results

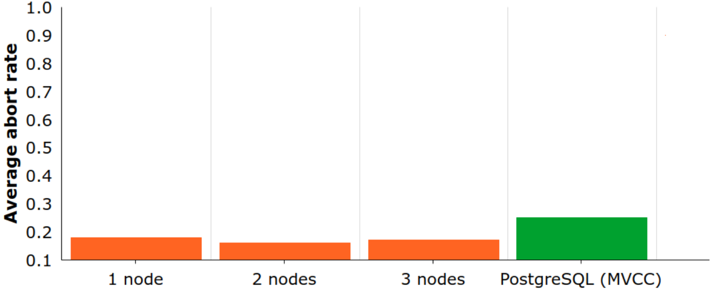


Throughtput (ops/sec)
3,500
3,000
2,500
2,000
1,500
1
2
3
Worker Nodes Count
Read-only workload



Throughtput (ops/sec)
600
550
500
450
400
350
300
1
2
3
Worker Nodes Count
Transactional workload

Abort rate


Average abort rate
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
1 node
2 nodes
3 nodes
PostgreSQL (MVCC)

Conclusion

- I studied how optimistic concurrency control schemes can help to provide transactional support for distributed database systems.
- Implemented lightweight main memory key-value storage.
- Adapted the original TicToc algorithm for distributed systems.
- 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.

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*.

References

[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.

Appendix: Original TicToc Validation phase

```
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
```

```
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 inWS do
  | commit_ts = max(commit_ts, w.ts + 1)
end
# Communication between nodes
# Step 3 - and Validate the Working Set
for w inWS do
  | if w.ts ≠ w.tuple.ts then
  |   | abort()
  | end
end
# Communication between nodes
```