High Performance Multi-Partition Transactions

Hua Fan
(Supervised by Wojciech Golab and Catherine Gebotys, with input from Brad Morrey, HP Labs)
University of Waterloo, Canada
h27fan@uwaterloo.ca

ABSTRACT

This PhD project addresses high throughput multi-partition, multi-get/put transactions, a.k.a. read-only/write-only transactions, in distributed storage systems providing strong consistency, specifically serializability, on various workload changes. Current systems either sacrifice performance for concurrency control, or trade consistency for performance. The project takes a new angle on the problem by proposing an epoch-based concurrency control mechanism, which separates read operations and write operations into different time slices, referred as epochs, to minimize conflicts among transactions. This mechanism also supports self-adaptation as workload characteristics change using the epoch duration as a tuning parameter. Using this epoch-based concurrency control protocol, our preliminary experiments achieve 1.5 million get/put operations per second per hosts on six server hosts. Our multi-get/put transactions are atomic and serializable.

1. INTRODUCTION

To overcome the performance gap between the traditional relational database and web scale distributed system requirements, researchers have built simpler semantic transaction systems, such as *get/put operation* distributed key-value stores, or systems only guaranteeing weak consistency, such as eventually consistent data stores. However, multi-partition *multi-get/put operation* transactions in distributed storage system are always expensive, especially if concurrent multi-get/put operation transactions have common keys. Sinfonia [1] uses minitransactions to provide strong consistency: more precisely ACID, using two-phase locking to lock the items accessed by the transaction. Two phase locking can be a bottleneck for performance when there is contention among transactions. This kind of contention is common with high throughput multi-partition multi-get/put operation transactions, which are the applications we are targeting. RAMP [2] explores the problem by weakening consistency for performance. They have defined a non-serializable isolation model called *Read Atomic (RA)*, which means all or none of each transaction's updates are visible to others. RA allows non-serializable read, which either limits the usage of the protocol or increases the complexity of application development. Furthermore, with respect to network efficiency, RAMP needs two round trips for write, one or two round trips for read, and possibly large meta-data or large second round message in communication, specifically, linear to transaction size. Note that under scenarios with many concurrent large transactions (the number of keys/partitions is large in one transaction), RAMP tends to have both two rounds for read and large message size overhead depends on size of data set.

More than building a specific high throughput multi-partition transaction protocol, another question is can we build a self-adapting protocol for different workloads?

Researchers usually build different systems for different workload requirements. In the face of read heavy vs write heavy trade-off, some systems design for write-optimized (for instance, Cassandra [5], an open-source version of Dynamo [3], is designed to handle high write throughput), others target read-optimized (for instance, RAMP, section 5.1 [2] targeting read-heavy applications). For (single) get/put operation heavy vs multi-get/put operation trade-off, VoltDB [7] uses a single thread to avoid latches or critical sections in each partition, and has very good performance for single node transactions, but performance suffers for multi-partition transactions, because all multi-partition transactions are sent to and serialized by a special global controller. This project requires that the transaction protocol self-tunes for performance, in the face of dynamic read-heavy and write-heavy workload changes, or between single-get/put operation heavy and multi-get/put operations heavy changes.

To achieve high throughput serializable transactions, and self-adaptation upon workload characteristics changes, this project explores a time-based scheme, which splits time into read-only epochs and write-only epochs. This concurrency control mechanism tends to minimize conflicts between multi-partition transactions, and minimize the meta-data and message size in the transaction for high performance. In our research plan, epoch duration is used as the tunable parameter for adapting to different workloads. The cost of epoch-based concurrency control is potentially large latency variation, because a transaction may need to wait the time up to one epoch duration before making progress, which is acceptable in the applications we are targeting.

Silo[8] is a single machine main-memory DB, which uses epoch number for serializability and recovery. In contrast, we focus on distributed transactions and use epoch to sep-

Proceedings of the VLDB 2015 PhD Workshop

arate read and write transactions, another difference from our project is that Silo uses epoch as commit unit, which means the clients will not get response until all transactions in epoch finish, and serialization order within an epoch is not recoverable.

In summary the objective of the PhD project is to contribute the following: 1) we propose an epoch-based concurrency control mechanism; 2) build a high throughput serializable, multi-partition multi-get/put transaction protocol for distributed storage system in single data center; 3) an approach for system self-tuning based on workload changes.

2. APPROACH

This section explains the system data model of our discussion, proposes the idea of epoch-based concurrency control, and details a basic corresponding transaction protocol. This section focuses on the core modeling of the epoch-based concurrency control, and if not specified we assume it runs in a failure-free scenario. Section 4 will discuss the research plan for other features, including fault tolerance and replication, various improvement and trade-offs for the protocol, and system self-tuning.

2.1 Data Model

We highlight the data models used in these project.

Data partition. Each item, identified by a key, has a single logical copy, residing in its hash partition. Each item has an initial value $\perp$, and can identify its partition using this key.

Multi-versioning. Each key has a latest value as well as snapshots of previous values. Each value is associated with a globally unique version number.

Timestamps and commutative overwrite policy. The version number is a timestamp used for choosing the latest version by a highest-timestamp-wins policy. The timestamp is generated by the server that receives the client's request at the time when the server begins to process the transaction. To generate a unique timestamp, the server could combine a unique server ID, a per-server increasing number and time from the clock. We assume clocks across servers are synchronized, and clock skew is less than 1ms within the data center, and the minimum transaction latency is at least twice the clock skew.

Read-only and Write-only transactions. We focus on read-only and write-only transactions, more precisely, multi-get/put operations. We assume the transaction size may be large, which means multi-put transactions have a high likelihood of write-write conflicts.

Serializable consistency. All committed transactions are serialized by the version number timestamp. Write transactions begin at the version number timestamp and finish at sometime within the write epoch. Because there are no read transactions making progress within write epochs, there are no inconsistent reads if we treat the write transactions as taking effect at the version number timestamps.

2.2 Epoch-based Concurrency Control

In our data model, every transaction is either read-only or write-only. Reads never conflict with reads. Writes are allowed to proceed in parallel, and are ordered using timestamps. Read transactions and write transactions are processed in different epochs to avoid read-write conflicts.

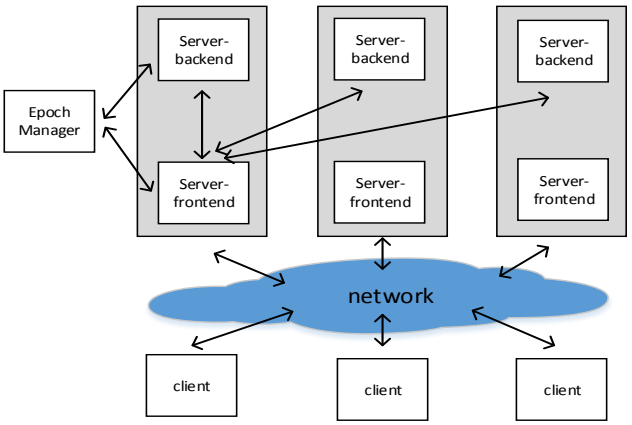

Figure 1: System architecture

Epoch-based Concurrency Control (ECC), splits time into epochs. Each server maintains a local epoch status. Using this epoch status, the server will behave as follows:

1. Transaction happens within an epoch. A transaction that begins within an epoch must also finish in the same epoch.

2. No uncompleted transactions. At the start time of an epoch, the server has no uncompleted transactions.

3. Read epoch. A server will not start any write transactions in a read epoch.

4. Write epoch. Server will not start read transactions in a write epoch. However, one exception is that read transactions only accessing snapshot of old versions (committed in previous write epochs) are allowed.

In order to successfully commit any multi-partition transaction, all the accessing partitions should have the same epoch statuses(read or write) during the transaction execution. To support high multi-partition transaction throughput, ECC needs a protocol to ensure epoch statuses among hosts are synchronized. Synchronized status means all hosts have the same status. Note that during the time of unsynchronized status, which means a subset of the servers have different epoch statuses from others, only transactions accessing the same status servers will succeed, and any successfully committed transactions remain serializable. We suggest the duration of each epoch is long enough compared to time used for each transaction and time to coordinate switching between read and write epochs(epoch switch time), to allow a significant number of transactions within each epoch.

2.3 Architecture

We target a distributed system in one data center, and Figure 1 shows the system architecture.

Server-backend (SBack). SBack stores items of a partition of the database, handles requests accessing these items.

Server-frontend (SFront). SFront accepts requests from clients, starts transactions, generates timestamps for each transaction, and communicates with SBack according to the epoch status, deciding whether the transaction either committed or aborted in the epoch. Clients can connect to any one of the SFronts, and each SFront may contact any SBacks where the required items reside.

EpochManager (EM). SBack and SFront are usually co-

located in the same physical server host (denoted by a grey box in Figure 1), but they may have their own epoch status. The EM communicates with all SBacks and SFronts for the purpose of synchronizing epoch status. Section 4 explains our plans on how to avoid EM being a single failure point or performance bottleneck in the protocol.

2.4 Transaction Protocol

This section first describes how the epoch switch guarantees the requirements of ECC, then specifies the algorithms of SBack and SFront when they agree on the current epoch.

Epoch switch algorithm: (1) The procedure is initiated from the EM, which requests all SFronts to finish their current epoch; SFronts will stop sending new requests upon receipt of this request and respond after finishing all pending transactions. (2) After getting all responses from SFronts, the EM requests SBacks to switch to the next epoch. For safety, SBacks respond after finishing all pending transactions. (3) After getting all responses from the previous step, the EM notifies all SFronts that all SBacks have changed to new epoch, and it's safe to begin sending requests for the new epoch.

Regarding communication between SBacks and SFronts, an SBack will reject requests within wrong epoch type which notifies the sending SFront it has become desynchronized. The protocol for multi-partition multi-get/put occurring within the correct epoch is demonstrated in Algorithm 1.

While maintaining a stronger consistency semantic, our protocol is simpler and has less overhead for multi-get/put than previous work (e.g. RAMP). For example, there is no need for meta-data which includes (in some variations) the write set of the transaction, in storage and message communication. Also it only needs one round trip for read, and one round trip for write when no failure occurs.

In a failure-free environment, transactions will not be aborted because of write-write conflicts for keys across transactions. In case of transaction abortion due to failure, we use a lazy update policy to minimize overhead. If the latest version needs to be aborted we only tag *removed* for that entry (Algorithm 1 L9), the entry will be overwritten upon a new put for that key, and the removed tag will be cleared. If a get operation find the $latestCommit[i]$ is removed, it needs to find the latest version from all committed versions (Algorithm 1 L14). Even if the versions are not sorted by timestamp or epoch order, the search is a one-time cost, because the result is saved to avoid searching for it again.

Algorithm 1: Multi-partition Multi-get/put operation

```
   SBack-side method
   Data: versions: set of
         version< item i, value v, timstamp ts >.
   latestCommit[i]:last committed timestamp for item i
   timestamp ts has a flag removed, default value is false
 1 Procedure Put(v: version)
 2     versions.add(v)
 3     latestCommit[i] ← max (latestCommit[i], v.ts)
 4     latestCommit[i].removed ← false
 5     return
 6 Procedure Abort(v: version)
 7     versions.remove(v)
 8     if v.ts = latestCommit[i] then
 9         latestCommit[i].removed ← true
10     return
11 Procedure Get(i:item, ts: timestamp)
12     if ts = ⊥ then
13         if latestCommit[i].removed = true then
14             latestCommit[i] ← versions.MaxTs(i)
15         return v ∈ versions : v.item = i ∧ v.ts =
           latestCommit[i]
16     else
17         return v ∈ versions : v.item = i ∧ v.ts = ts
   SFront-side method
18 Procedure PutAll(W: set of < item i, value v >)
19     ts ← generate new timestamp
20     V ← {v|∀w ∈ W, v = {w.i, w.v, ts}}
21     parallel-for v ∈ V do
22         invoke Put(v) on respective partition.
23     if all previous Put not rejected or timeout then
24         return
25     else
26         parallel-for v ∈ V do
27             invoke Abort(v) on respective partition.
28 Procedure GetAll(I : set of items)
29     parallel-for i ∈ I do
30         invoke Get(i, ⊥) on respective partition.
```

3. PRELIMINARY EXPERIMENTS

We have conducted a preliminary experiment to evaluate a simple implementation of ECC described in section 2.4. The experiment was deployed in Amazon EC2, using c3.8xlarge instances. Because we were not able to have more than 20 instances in one Amazon placement group, we used 19 instance to test up to 6 server hosts, 12 YCSB client hosts, and one running controlling scripts. We used fbthrift for RPC calls, and libcuckcoo for internal storage in SBack. Each YCSB host had 64 client threads using uniform distribution, half of them were dedicated update clients, which had 100% write workload; The other half were dedicated query clients, which had 100% read workload. The clients grouped sets of gets and puts into multi-get and multi-put transactions, and each transaction had 500 get/put operations. We experimented with epoch durations of 500ms and 1000ms. We also implemented a baseline (no concurrency control), which should be the upper bound for performance. In the baseline, SF and SB will process the requests without any epoch-based or lock-based concurrency control.

Figure 2 shows the aggregate throughput and average latency using various numbers of server hosts. The throughput of operations is throughput of transaction times transaction size. In our experiment, when epoch duration was 1000ms(epoch 1000), the system achieved approximately 1.5 M get/put operations per second per host, which is only 10% less than the baseline implementation without any concurrency control. For the multi-server scenario, the experiment demonstrates nearly linear scalability. Clients blocked because they submitted a read during a write epoch or vice versa had requests with latency nearly the same as the epoch duration. However, because the number of these blocked transactions was negligible compared to the overall transaction count, the average latency is close to that of unblocked transactions. If the clients are not performing a dedicated read or write workload, the average latency could be much higher. We plan to explore these scenario using nonblocking

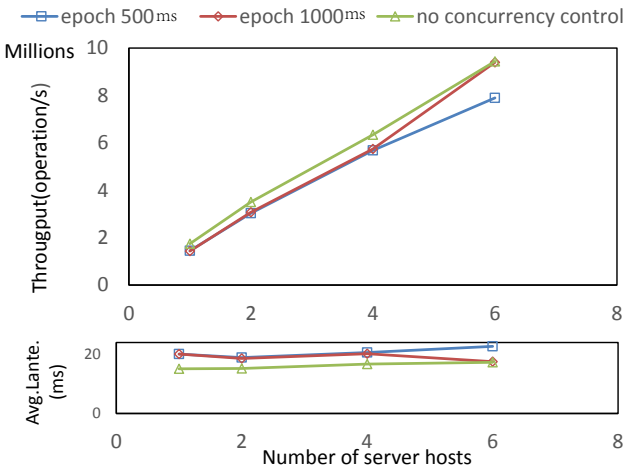

Figure 2: Throughput and latency result

YCSB clients or large number of clients in future work.

4. RESEARCH PLAN

We propose the following research plan to meet the project requirements not discussed in section 2, and overcome some assumptions and limits in our previous work.

Self-tuning based on workload. If the EM allows repeated epochs of the same type, the system can adapt to the read/write ratio of the workload for improved throughput. Multi-get/put operations favor longer epoch durations, because it takes longer to process larger transactions; however, if the duration is too long, get/put operation latency (and variance) will increase with no benefit. We propose tuning epoch type and duration as follows: the SFront will use a metric to track the requests from clients; another process collects these metrics from all SFronts (or by sampling), and suggests future epoch type and duration to the EM based on a predefined strategy. The metric usually includes property and behavior statistics of recently requests, such as type, transaction size, and latency.

Fault tolerance and replication. The system should tolerate the failure of a single server. Server health is exchanged by heart beats. SBacks are replicated in a classic primary-backup manner, with the backup taking over when the primary fails. SFronts operate slightly differently from a classic primary-backup: the backup only finishes the uncompleted transactions, but never takes over the primary for new client requests. Clients can connect to any SFront, and the system can easily add more SFronts. The EM has a replicated backup, which tracks every request and response of the primary. Because SF and SB passively handle epoch change requests from the EM, the backup EM can easily take over the role by sending follow on requests to SF and SB. For a larger data center, the EM service can be implemented by multiple physical hosts.These EM units can be organized hierarchically, or coordinated using a distributed consensus protocol (e.g. Paxos [6]) or distributed coordination (e.g. ZooKeeper [4]).

Agile epoch switch. To maximize overall performance, we want to minimize the duration of each epoch switch.

To allow SF and SB to reduce the time for finishing uncompleted transactions, we plan to explore strategies for epoch planning. The high level idea of *Epoch planning* is to alert SFronts when the epoch switch request will come to allow them to prepare for it. For example, the EM propagates the expected time of next epoch switch to SF; SFront will throttle multi-put/get operations depending on transaction size and remaining time before expected epoch switch.

Differentiated treatment is a strategy to address the problem that the epoch switch is slow because EM waits for a response from a slow server. Plan in detail: EM tracks the history of each server's response time; EM will send epoch switch requests to slow servers (based on the history) earlier than others.

Formal proof of serializability and Model. Even though section 2.1 informally shows the timestamp of version number can be used for the serialization. We plan to give a formal proof of serializability of epoch-base concurrency control mechanism. We also want to identify the key variables that affect the performance, and establish a model for performance of ECC.

5. CONCLUSION

This paper discusses the problem of building high throughput multi-partition, multi-get/put transactions with serializability, by exploring an epoch-based concurrency control method. Currently, based on a simple prototype of ECC, our preliminary experiment achieves around 9 million operations per second using 6 server hosts, which achieves 90% throughput of no concurrency control and 10 times the throughput reported by RAMP under small transaction size and read heavy workload using the same number of servers. We also show our research plans and ideas to extend the assumptions and limits of the protocol.

6. REFERENCES

[1] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. *ACM Trans. Comput. Syst.*, 27(3):5:1–5:48, Nov. 2009.

[2] P. Bailis, A. Fekete, J. M. Hellerstein, A. Ghodsi, and I. Stoica. Scalable atomic visibility with RAMP transactions. In *Proceedings of SIGMOD '14*, pages 27–38, New York, NY, USA, 2014. ACM.

[3] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, Oct. 2007.

[4] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of USENIXATC'10*, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[5] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[6] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[7] M. Stonebraker and A. Weisberg. The VoltDB main memory DBMS.

[8] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of SOSP '13*, pages 18–32, New York, NY, USA, 2013. ACM.