Centiman: Elastic, High Performance Optimistic Concurrency Control by Watermarking

Bailu Ding, Lucja Kot, Alan Demers
Cornell University
{blding, lucja, ademers}@cs.cornell.edu

Johannes Gehrke
Microsoft Corp., Cornell University
johannes@cs.cornell.edu

Abstract

We present Centiman, a system for high performance, elastic transaction processing in the cloud. Centiman provides serializability on top of a key-value store with a lightweight protocol based on optimistic concurrency control (OCC).

Centiman is designed for the cloud setting, with an architecture that is loosely coupled and avoids synchronization wherever possible. Centiman supports sharded transaction validation; validators can be added or removed on-the-fly in an elastic manner. Processors and validators scale independently of each other and recover from failure transparently to each other. Centiman's loosely coupled design creates some challenges: it can cause spurious aborts and it makes it difficult to implement common performance optimizations for read-only transactions. To deal with these issues, Centiman uses a *watermark* abstraction to asynchronously propagate information about transaction commits through the system.

In an extensive evaluation we show that Centiman provides fast elastic scaling, low-overhead serializability for read-heavy workloads, and scales to millions of operations per second.

Categories and Subject Descriptors H.2.4 [*Database Management*]: Systems—Concurrency, Transaction processing, Distributed databases

Keywords optimistic concurrency control, elastic transaction processing, sharded validation

1. Introduction

Transaction processing has long been a cornerstone of database functionality, and it remains so today, in the era of the cloud. Notwithstanding the popularity of key-value stores [1, 2, 21, 25, 39] that prioritize maximum scalability and do not support multi-item transactions, there is a lot of interest in supporting ACID transactions in the cloud.

The cloud setting introduces new challenges for data management due to a distributed, loosely coupled infrastructure [44]. Storage is sharded and/or replicated, and the nodes performing the processing are separate from the storage system. Each component may fail independently; we need to handle this with efficient recovery and/or migration of work. In addition, as workloads grow and shrink, a cloud-based transactional solution should scale elastically. Finally, the proposed solution should be easy to deploy on the commodity cloud, so it should make minimum demands on the APIs of the components that form its building blocks.

We have seen a large effort from the research community to support transactions in the cloud and in distributed systems more generally. Various tradeoffs have emerged: for example, there are systems which provide weaker consistency guarantees than serializability in order to gain scalability benefits [10, 42, 43, 53]. Among systems that support strong consistency guarantees, some restrict the type of transactions permitted [59, 60], or partition data and support ACID semantics only within a partition, with weaker guarantees provided across partitions [11, 24, 34, 35, 46].

In this paper, we present Centiman: a transaction processing solution that is designed for the cloud. Centiman provides full ACID guarantees and supports millions of operations per second without restricting the permitted transaction types or semantics. It is simple to deploy on commodity infrastructure on top of any existing key-value store. Finally, Centiman allows elastic scaling for every component of the system (data storage, processing, validation).

Centiman maintains transactional guarantees using a variant of the optimistic concurrency control (OCC) protocol [37], which is justifiably popular in distributed transaction processing [11, 17, 18, 22, 28, 46, 47] due to its low overhead in the low-contention setting [5].

The distinguishing feature in Centiman is that validation is sharded and proceeds in parallel on a set of validators that can grow or shrink as needed. Thus, in a multi-tenant setting, small tenants can use one validator and reap the advantages

SoCC '15, August 27-29, 2015, Kohala Coast, HI, USA
2015 ACM. ISBN 978-1-4503-3651-2/15/08$15.00
DOI: http://dx.doi.org/10.1145/2806777.2806837

of locality while large tenants can scale out elastically. Under normal operation, the only points of synchronization in the system are the start of validation (to ensure that transactions enter validation in timestamp order), and the point where a processor responsible for a transaction collects all the validators' outcomes and combines them into a single commit or abort decision. All other phases proceed asynchronously. In particular, the writes of a transaction can be propagated to storage asynchronously and interleave with the reads and writes of other transactions. Moreover, the validators never need to communicate with each other.

This loosely coupled design does come with challenges. The first is *spurious aborts*. Validators do not know the global outcome of the validation; even if a transaction aborts, an individual validator may believe it to have committed. The validator will thus validate future transactions against it, potentially detecting false conflicts.

The second challenge relates to read-only transactions. In normal OCC, we can allow such transactions to bypass validation if we ensure that they read a consistent snapshot of the database [7, 38]. Because Centiman does not require a transaction's writes to be installed to storage atomically, it is difficult to ensure that a transaction reads a recent and consistent snapshot of the database. Thus, the above optimizations are harder to implement.

Both of the above issues could be solved with synchronization; we could communicate abort decisions to the validators and we could require transactional guarantees on the installation of writes to storage. However, this would introduce additional points of blocking. Instead, we use a *watermark* abstraction to asynchronously disseminate through the system information about which transactions have committed and completed writing to storage, and consequently which records in storage are "stable." This addresses both issues without introducing points of blocking in the system. Watermarks allow validators to learn about aborted transactions and maintain their state more precisely, and we can use them to bypass validation for read-only transactions in certain cases. The watermarking technique is applicable beyond Centiman to improve performance in other OCC systems.

In the Centiman system, we make the following specific contributions. First, we show how to perform sharded OCC validation with minimum system-wide synchronization through the use of watermarks. Second, we explain how to implement watermark-based OCC validation in a cloud setting that supports elastic scaling. Finally, we present an extensive experimental evaluation of our implemented system. The system scales to over 230 thousand transactions per second for a variant of the TPC-C benchmark and over 4 million transactions per second for the TATP workload when running on commodity Amazon EC2 nodes.

In the remainder of this paper, we introduce the overall architecture of Centiman (Section 2) and our watermark-based sharded validation approach (Section 3). We discuss the system implementation (Section 4), present our experimental evaluation (Section 5), discuss related work (Section 6) and finally conclude and discuss future work (Section 7).

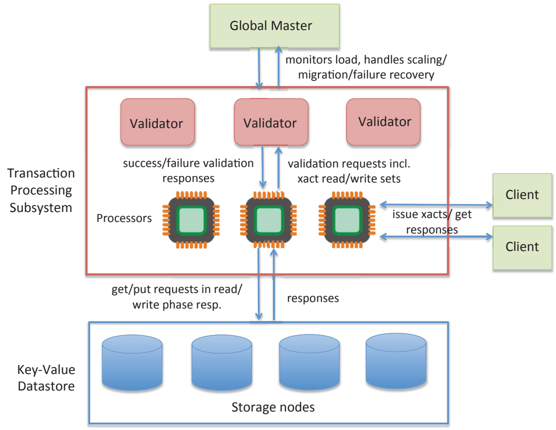
Global Master
monitors load, handles scaling/ migration/failure recovery
Transaction Processing Subsystem
Validator
Validator
Validator
success/failure validation responses
validation requests incl. xact read/write sets
Processors
issue xacts/ get responses
Client
Client
get/put requests in read/ write phase resp.
responses
Key-Value Datastore
Storage nodes

Figure 1: Centiman System Architecture

2. The Centiman system

In this section we describe the architecture of Centiman at a high level and outline the life cycle of a transaction.

2.1 System architecture

Centiman contains the following main components: a *datastore*, a *transaction processing subsystem* consisting of *processors* and *validators*, a *global master*, and *clients* that issue transactions as shown in Figure 1.

The *datastore* is a key-value store that may be partitioned and/or replicated. Both the partitioning and replication are transparent to Centiman. The datastore may be an external, third-party datastore. We require a particular get/put API, as described below. If the datastore does not support it natively, the API can be implemented as a layer between the datastore and the transaction processing subsystem.

We refer to the key-value pairs in the datastore as *records*. Every record is associated with a *version*, which is the timestamp of the transaction that wrote this update.

Centiman requires the following datastore API:

- $put(key, value, timestamp)$ where $timestamp$ is the identifier of the transaction making the write. This updates the record as follows. If $timestamp$ is smaller than the current *version* for the record, the request is ignored as the write is stale (a higher-timestamped transaction has already updated the record). Otherwise, the system sets the value to *value* and the version to $timestamp$.
- $get(key)$. This returns $(value, version)$, where *version* is as explained above.

We also require that each processor has a way to determine when a write to the datastore has been *installed* [6]. A write to record with key *key* by transaction *i* is installed when it is guaranteed that every subsequent *get(key)* request returns a value with version equal to or greater than *i*. The moment when a write is installed may be when the *put* call returns, or at a later time if the system uses asynchronous replication and allows the *put* to return before a write is available everywhere.

Clients issue transactions to the system. Whenever a client issues a transaction, it communicates with a specific *processor* and the transaction remains associated with that processor for the duration of its lifetime. During the life cycle of the transaction, the processor issues the transaction's read requests to the datastore, caches its writes in a private workspace, assigns each transaction a *timestamp*, sends validation requests to one or more validators, issues write requests to the storage if necessary, and replies to the client.

The *validators* perform a variant of optimistic concurrency control (OCC) validation [37]. Each validator A is responsible for a subset of the data, as defined by a subset K_A of the overall key space. This partitioning does not necessarily coincide with the partitioning at the datastore.

The *global master* monitors system performance, coordinates elastic scaling, and handles failure recovery.

2.2 Transaction life cycle

The transaction life cycle follows the OCC model [37]. Every transaction has a *read phase* when it reads from the datastore and writes to a private workspace, a *validation phase* when the system determines whether the transaction may commit, and – if validation was successful – a *write phase* when the transaction's writes are installed in the datastore.

Whenever the transaction needs to read a record, the processor issues an appropriate $get(key)$ request to the storage. The *get* returns a pair $(value, version)$. The processor adds $(key, version)$ to the *read set* of the running transaction.

Whenever the transaction needs to write a record with key *key*, the processor buffers the write in a local private workspace, and adds *key* to the *write set* of the transaction.

When a transaction is ready to commit, the processor assigns it a timestamp i. As soon as a transaction's timestamp is assigned, it is ready for validation. The validation process makes use of the read and write sets collected during execution; we denote them by $RS(i)$ and $WS(i)$ respectively.

Validation for each transaction happens at one or more *validators* and is carried out in timestamp order. That is, a validator will only process a transaction with timestamp i after it has validated (its portion of) all transactions with timestamp $j < i$. We discuss possible ways of implementing timestamps and timestamp-order validation in Section 4.1.

When a transaction with timestamp i enters the validation phase, each participating validator receives the timestamp i and the appropriate portion of i's read set and write set. Formally, validator A receives i, $RS_A(i) = \{(a,v) \in RS(i) \mid a \in K_A\}$ and $WS_A(i) = \{b \in WS(i) \mid b \in K_A\}$.

At each validator A, $RS_A(i)$ is used to validate only i itself and discarded after validation. $WS_A(i)$ may be needed in order to validate subsequent transactions with timestamps $j > i$ and is thus cached at the validator. The validation algorithm itself is described in Section 3.

```
 1: procedure VALIDATE($i$, $RS_A(i)$, $WS_A(i)$)
 2:     for $(key, v) \in RS_A(i)$ do
 3:         for $j \in (v, i)$ do
 4:             Get $WS_A(j)$ from $WriteSets(A)$
 5:             if $key \in WS_A(j)$ then
 6:                 return abort
 7:             end if
 8:         end for
 9:     end for
10:     $WriteSets.add(WS_A(i)))$
11:     return commit
12: end procedure
```

Figure 2: Naïve algorithm for validating transaction with timestamp *i* at validator *A*.

When each validator completes validating its portion of transaction i, it communicates its decision to the processor responsible for i. If the processor receives "commit" responses from all participating validators, it determines that the transaction will commit, otherwise (it receives an "abort" response or a timeout from at least one validator) it determines the transaction will abort.

If the transaction aborts, there is no further action taken and the transaction is *complete*. If the transaction commits, the processor sends to the datastore a *put* request for every write in $WS(i)$. These *put* requests may interleave with *put* and *get* requests from other transactions. Once all the writes are installed in the datastore, the transaction is *complete*.

3. Validation in Centiman

In this section, we present the main conceptual contribution of Centiman: sharded validation with watermarks. We begin with a general discussion of sharded validation (3.1), introduce watermarks (3.2), show how watermarks allow certain read-only transactions to bypass validation (3.3), and describe the protocol for elastic scaling of validators (3.4). The validation algorithm we present guarantees serializability; it can be adapted to enforce snapshot isolation [15] instead.

3.1 Sharded Validation

We begin with a naïve sharded validation algorithm for validating a transaction with timestamp i, shown in Figure 2.

Validation proceeds in timestamp order. Thus when validating i, every validator A involved has available the write sets of all previous transactions, i.e. it knows $WS_A(j)$ for all $j < i$. These are stored in a buffer called $WriteSets_A$.

The algorithm processes each of i's reads in sequence (Lines 2-9). For each read that saw version v of a record with key *key*, it examines the write sets of all transactions j where $v < j < i$ (Lines 3-8). If any such transaction j wrote to the same record (with key *key*), then validation fails (Line 6). Intuitively, i read a stale version of the record as it should have seen j's write instead. If validation is successful, $WS_A(i)$ is added to $WriteSets_A$ for future validation of transactions with timestamps greater than i.

As explained earlier, the processor responsible for the transaction collects responses from all participating validators. It allows the transaction to commit and install writes if and only if all participating validators reply *commit*.

The correctness of this algorithm follows from the fact that it is a sharded version of the classic algorithm for OCC validation given in [37]. For correctness of that classic algorithm, it is required that a write by transaction i never overwrites a write by transaction j, $j > i$. In [37] this is done by forcing the validation and write phases of each transaction to occur in a critical section. In our case we do so by ensuring that the storage rejects all writes to a record r by transaction i if a version $j > i$ is already installed for r.

As the system runs, $WriteSets_A$ grows and may impose a nontrivial storage overhead at the validator. There are known truncation-based strategies to deal with this problem [37]. However, overly aggressive truncation may cause unnecessary aborts. Consider Line 3 in Figure 2. Suppose a transaction has read a very old version of some record, but the version is still current as the record is infrequently updated. Then the interval (v,i) is very large, and it may be that for some j in that interval $WS_A(j)$ has been garbage collected. A conservative approach has no choice but to abort that transaction. Alternately, if we know a bound on the time window within which the write of a committed transaction is guaranteed to become persistent in storage, we can use a more precise approach. We can garbage collect write sets of transactions when their writes are guaranteed to be persistent, and adjust the validation algorithm accordingly. However, it is not always possible to obtain such timing information, especially in a cloud setting.

The algorithm in Figure 2 is correct in that it never allows a violation of serializability; however, it may cause **spurious aborts**. Suppose transaction i passes validation at validator A but fails validation at validator B. Then validator A will add $WS_A(i)$ to $WriteSets_A$, "polluting" $WriteSets_A$ with the write set of a transaction that did not actually commit. When validating a subsequent transaction j, the validator may detect an overlap between $WS_A(i)$ and $RS_A(j)$ and fail validation, even though the conflict between i and j is spurious.

Garbage-collecting write sets partially addresses this problem as write sets of aborted transactions "age out" from the system. However, if the garbage collection is not agressive the aging out is slow, and if it is aggressive we expect a higher abort rate due to missing state, as explained above.

The alternative solution is to eliminate spurious aborts by broadcasting commit decisions back to the validators. Doing this synchronously would reduce the number of spurious aborts to zero, but requires blocking and is incompatible with our goal of a loosely-coupled system. Instead, we disseminate this information asynchronously using *watermarks*.

```
 1: for $(key, v, w) \in RS_A(i)$ do
 2:     for $j \in (MAX(w, v), i)$ do
 3:         Get $WS_A(j)$ from $WriteSets(A)$
 4:         if $WS_A(j) \notin WriteSets(A)$ then
 5:             return abort
 6:         end if
 7:         if $key \in WS_A(j)$ then
 8:             return abort
 9:         end if
10:     end for
11: end for
```

Figure 3: Innermost loop of the watermark-based algorithm for validating transaction with timestamp i at validator A; replaces Lines 2-9 of the algorithm in Figure 2

3.2 Validation with watermarks

Watermarks are metadata that propagate through the system information about the timestamp of the latest completed transaction. In classical OCC, where the validation and write phases are in a critical section, it is relatively easy to obtain this information; we now explain how our watermarks help to obtain it in a sharded and asynchronous setting.

In Centiman, when a transaction reads a record r from storage and receives the pair $(value, version)$, the processor computes a watermark w, which has the following property:

Property 3.1. *If record r has watermark w at time t, then at time t all transactions with timestamp $i \leq w$ have either already installed their writes on r or will never make any writes to r.*

Intuitively, this says that any update to r made by a transaction with timestamp less than w has been reflected in the read. The processor computes the watermark at the time of the read and stores it in the read set together with the key and version. Thus read sets now contain triples (key, v, w) where v is the version and w the watermark.

We explain how the watermarks can be computed in Section 4.2. Once they are available, we can use them in validation by modifying Lines 2-9 of the naïve algorithm to those shown in Figure 3. We can now start checking write sets after $MAX(v,w)$ rather than after v. Correctness of the new algorithm follows from the correctness of the original algorithm and from Property 3.1.

Watermarks reduce spurious aborts without the need for extra communication because the write sets of aborted transactions eventually "age out" and fall below the watermark, thus they are never considered in validation again.

Watermarks can also inform the validator garbage collection strategy. It is safe to garbage collect $WS(i)$ once all read watermarks of all in-flight and future transactions will be greater than i. If we garbage collect $WS(i)$ more aggressively than that, in certain cases the validator will have insufficient state to guarantee the absence of a conflict and must conservatively reply with an abort (Lines 4-6).

3.3 Local check for read-only transactions

Read-only transactions present a special opportunity for optimization in OCC. If we ensure that they read a consistent snapshot of the database, then they do not require validation [7, 38]. In Centiman, we do not require transaction writes to be installed atomically; thus, ensuring that a transaction reads a consistent snapshot is not trivial. However, in certain cases we can retroactively determine that a read-only transaction saw a consistent snapshot; if we can determine this at the processor, the transaction does not need to be sent to the validators. Our watermarks allow the processor to run a simple, conservative check on read-only transactions; if the check passes, the transaction can commit immediately. If the check fails, the transaction undergoes normal validation.

To explain our check, we define the following:

Definition 3.2. *For a timestamp i, we say that a datastore is* at snapshot i *if no version of any record with a timestamp $j \leq i$ will ever be installed in the future, and no version of any record with a timestamp $k > i$ exists in the datastore.*

Definition 3.3. *We say that a transaction* reads at snapshot i *if all the reads that it performs see the same values and versions as they would in a datastore at snapshot i.*

It is possible for a transaction to read at snapshot i even if the datastore is not at snapshot i, as long as the transaction sees the “correct” versions for the records it actually reads.

If a transaction T runs in Centiman and we can determine that there exists some i such that T reads at snapshot i, then T can safely bypass validation. We present a simple, conservative algorithm to detect such situations in Figure 4. The basic idea is as follows; suppose the read set of T includes a triple (key, v, w). Suppose $v \leq w$. Then we can define an interval $[v, w]$, which intuitively represents a subset of the time frame during which the version T read is the most current installed version in the system. Our algorithm is based on computing such an *effective interval* for every record T has read and checking whether the intersection of all these intervals is nonempty.

The algorithm in Figure 4 can be run when the full read set $RS(T)$ is available, or incrementally as the read set grows. For every triple (key, v, w) in the read set, it inspects the version v and the watermark w (Lines 3-9). If $v \leq w$, it determines the interval $[v, w]$ and adds it to the running intersection of intervals (Lines 4-5). Otherwise it adds $[v, v]$ to the intersection (Line 6). Finally, it returns true if and only if the running intersection interval is nonempty after the entire read set is processed (Line 10).

The algorithm returns $true$ if and only if T can read at snapshot c for any c in the final $Interval$. This correctness property follows from the way the $Interval$ was constructed. Suppose the final $Interval = [a, b]$ and we have $a \leq c \leq b$. Consider any record in $RS(T)$, and suppose T read that record at version v and watermark w. We need to argue that the version T saw is the same as in a datastore at snapshot c. However, we know that no transaction with timestamp between v and b writes to the record (since either $b \leq w$ or $b = v$). As $v \leq c \leq b$, the desired property follows.

```
 1: procedure LocalCheck($RS(T)$)
 2:     $Interval \leftarrow (-\infty, \infty)$
 3:     for $(key, v, w) \in RS(T)$ do
 4:         if $v \leq w$ then
 5:             $Interval \leftarrow Interval \cap [v, w]$
 6:         else
 7:             $Interval \leftarrow Interval \cap [v, v]$
 8:         end if
 9:     end for
10:     return $(Interval \neq \emptyset)$
11: end procedure
```

Figure 4: Local check for determining whether a read-only transaction T read at snapshot i for some i.

3.4 Elastic validator scaling

Centiman is designed so that the functionality at each layer – the datastore, the processors and the validators – can scale by adding more nodes and/or migrating workload as necessary. At the storage level, we assume that scaling is handled by the datastore itself. Processors can be added to and removed from the system in a straightforward manner. We next describe our protocol for validator scaling.

Fundamentally, validator scaling (up or down) and validator migration require the same action in the system: we need to change the way the key space is partitioned among a set of validators. The global master monitors the system load, decides when to perform scaling and/or migration, and decides on the new key space partitioning.

We now describe how to scale up validators. Scaling down and migration are similar. Any new validator nodes register themselves with the global master and connect to all processors. Next, the global master decides on the new key space partitioning. It informs all processors to send validation requests based both on the old and the new partitioning.

The system is now in a transitional period where validators operate both under the old and the new partitioning, and issue validation decisions under both. However, the authoritative decisions are the ones made under the old partitioning. When a validator is operating under the new partitioning, it may not always have sufficient state to validate transactions (Figure 3 Lines 4-6); in this case, it replies either with “abort” or with a special message saying “unknown.”

Initially the validation decisions made under the old and the new partitioning may be different. It is possible that the global decision is “commit” under the old partitioning but “abort” under the new partitioning because at least one validator has not yet built up sufficient state. However, the reverse can never happen: the decision under the new partitioning is always at least as conservative as the decision under the old partitioning. As time passes and the validators accumulate state under the new partitioning, the decisions made under both partitionings converge.

At a suitable point, the global master decides to switch to the new partitioning exclusively. It can make this decision based on the convergence of the validation decisions under both partitionings – for example, switch over when no disagreement in the validation decisions has occurred for a specific period of time. Alternately, individual validators could notify the master that they have accumulated sufficient state to switch over, e.g. using the watermark as an indicator.

Switching over to the new partitioning requires synchronization between the master and all processors. Once the switch is made, the old partitioning is "forgotten"; processors send validator requests only under the new partitioning. Validators do not need to be notified of the switch; they just stop receiving validation requests under the old partitioning.

The validator scaling protocol only introduces two sources of overhead. First, during the transitional period, twice the normal number of validation requests is sent for a portion of the data. Second, the system requires global synchronization to finalize the switch to the new partitioning. However, no state needs to be moved between validators.

4. Implementing sharded validation

We now discuss three important aspects of implementing Centiman's sharded validation: timestamp-order validation (4.1), watermarks (4.2), and failure recovery (4.3).

4.1 Timestamps and timestamp-order validation

In our validation algorithm (3.1), transactions must be timestamped at the processor and validated in timestamp order.

Processors can assign timestamps based on physical system time, using technologies like the TrueTime API [22] or Logical Physical Clocks [36]. Alternately, each processor can just assign monotonically increasing timestamps from a set of integers that is unique to the processor.

Each validator maintains a sliding time window that starts in the past and ends at the present. The window is defined either in physical time or in logical "time", depending on the timestamps. The assumption is that validation requests for transactions with timestamps within the window may still be "in flight" to the validator, but for timestamps before the window all validation requests are available.

The validator buffers arriving validation requests. It processes them by repeatedly advancing the sliding window; any requests that move outside the window are processed in timestamp order. If the validator receives a request with a timestamp *before* the current sliding window, it responds with an "abort." Thus it is possible that some transactions will abort because their validation requests do not reach a validator in time. However, as long as we deploy Centiman on dedicated machines and use a fast network with a low, predictable latency, we can tune the validator time window to minimize the number of such aborts.

If the timestamps are not based on physical time, some extra logic may be required on the processor side. For instance, a processor that is idle for some time or a new processor that joins the system must communicate with the validators to "catch up" its timestamps to the rest of the system.

4.2 Implementing watermarks

Every time a record is read, the processor needs to compute a watermark for the record that satisfies Property 3.1. Centiman runs an *approximate, conservative* watermark computation algorithm which makes use of the fact that Property 3.1 is *downward-closed*: if for a given record r the integer w has the property, so does any $w' \leq w$. That is, we can use approximate watermarks which are not "as large as possible," but still allow correct validation.

Suppose the set of all processors in the system is $\mathscr{P}$. In Centiman, each processor $P \in \mathscr{P}$ maintains a *local processor watermark* W_P. The meaning of W_P is that every transaction associated with P with timestamp $i \leq W_P$ has already completed (aborted or committed and installed all writes). Denote $min_{P \in \mathscr{P}} W_P$ as W_G. By construction, it is guaranteed that every transaction with timestamp $j \leq W_G$ has completed. Thus at any given time W_G satisfies Property 3.1 for any record r, because no transaction below W_G will install any more writes to any record in the datastore.

Each processor periodically recomputes its local watermark, and caches information about the other processors' local watermarks. The cached information must be refreshed periodically, for example using a *gossip protocol* [26]. To compute a watermark for a read, each processor uses the minimum of its own local watermark and the cached watermarks of other processors; this is smaller than or equal to W_G, so it also satisfies Property 3.1.

The frequency of recomputing and disseminating the local processor watermarks are tunable parameters. Updating watermarks more often is more expensive, but reduces the number of spurious aborts and allows more read-only transactions to pass the local check from Section 3.3.

4.3 Failure recovery

We now explain how to recover from processor and validator failures in Centiman. We assume that the datastore uses existing techniques to achieve fault-tolerance [25] and that the global master is implemented using robust infrastructure such as Apache Zookeeper [4]. We also assume that the processor and validator nodes are fail-stop.

Each processor node maintains a write-ahead log to enable a redo of writes to the datastore. Because no writes are performed until validation is complete, no undo logging is needed. On receiving a transaction T from a client, the processor makes an *init* log entry for T and writes it asynchronously (without forcing). Before sending validation requests to validators, it asynchronously logs the write set of T. After hearing from all validators, it decides either to commit or to abort T and logs the decision for T; the log entry includes the timestamp of T. If the decision is to commit, the processor forces the log, and subsequently informs the client of the commit decision and sends the write set to the datastore. When the writes have been installed, the processor makes a *completed* log entry for T asynchronously. To recover a failed processor, we read the log and redo the writes of any transaction T for which the log contains a commit decision but no *completed* entry.

From the perspective of the validators and/or other processors, a processor failure is transparent; the only negative consequences are a delay in installing certain writes to storage and aborts of transactions for which the processor had not yet reached a decision to commit. In particular, it is possible that the processor fails after it reaches a decision to commit but before it forces the commit log entry; in this case, upon recovery, the transaction will be aborted. This does not violate correctness because the client will not yet have received a commit response. The validators will believe that T has committed, which may lead to spurious aborts; however, this problem will resolve itself with time.

It is not necessary to log the processor watermarks (Section 4.2); upon recovery, the processor can simply set its local processor watermark to the highest timestamp of the last transaction for which it has a commit log entry once it determines that the writes of this transaction have been installed.

Validators do not require logging for recovery even though they maintain state (i.e. transaction write sets). When a validator fails, we need to abort all transactions with pending validation requests at that validator - this will happen automatically if the processor uses timeouts when waiting for responses. To recover from a validator failure, we add a new validator to take the place of the failed one and use a variant of our protocol from Section 3.4 where the old and the new partitioning are identical and the switch between them happens immediately. Note that a validator failure is transparent to all other validators and processors.

5. Experiments

In this section, we present an experimental evaluation of Centiman. We investigate the following questions:

- How do spurious aborts (Section 3.1) affect the system, with and without watermarks?
- How does the system behave when we scale elastically by adding a new validator as discussed in Section 3.4?
- How effective is the local check for read-only transactions (Section 3.3)? How many read-only transactions bypass validation in workloads with a different mix of read-only and updating transactions?
- How well does the system scale and perform under load?

5.1 Implementation details

We implemented the system in about 20K LOC in C++, including the logic for the processor, storage, validator, a synthetic benchmark, and TPC-C and TATP benchmarks.

Each processor node is implemented as a single worker thread that multiplexes concurrent transactions up to the node's *concurrency level*. We simulate each storage node with an in-memory hash table. The *put* operation returns after the update is stored in the hash table and is available for reads. Each validator caches the write sets of transactions that pass its validation in a fixed size buffer implemented as an in-memory hash table.

Each processor assigns consecutive integers as timestamps, using the processor ID to break ties. If a validator has at least one pending request from each processor, it repeatedly handles the lowest-numbered request, otherwise it waits. To handle 'holes' due to a processor which is slow, failed or just does not communicate with this particular validator, there is a bound on the number of permitted pending requests at a validator. If this bound is exceeded, the validator processes the lowest-numbered pending request, in effect forcibly advancing the timestamp window. In our experiments, the processors are deployed on homogeneous hardware, and execute transactions at similar speed. Thus, forcibly advancing the window is a rare event.

All the system components (validators, processors and storage nodes) batch their outgoing messages and send them periodically (every 10ms) to manage the network overhead. If an application setting requires lower latency, or if the 10ms latency causes a high conflict rate, the batching and sending frequency can be increased. Our experiments on TPC-C show that our choice of batching frequency does not cause problems even with a non-trivial workload that has reasonably high contention.

We conduct four sets of experiments, focusing on spurious aborts, elastic scaling, read-only local check, and scalability. For the first three sets of experiments we use up to 12 machines; for the last set, we use up to 108 machines. All the results we show are averages over three runs. Differences between runs were not significant, so error bars are omitted.

5.2 Spurious aborts

As discussed in Section 3.1, spurious aborts arise due to write sets of transactions that pass validation locally but abort globally. At the validator where validation was successful, the $WriteSets$ buffer is polluted by the write sets of this transaction. Subsequent transactions validated against this polluted $WriteSets$ may abort due to spurious conflicts.

We examine the frequency of spurious aborts under a naïve truncation-based approach to validator state management and under our watermarking scheme. In the naïve approach, we use a "sliding window" scheme and we store 10 (B10), 20 (B20), 30 (B30), and 60 (B60) seconds' worth of past write sets in the validator buffer.

It may happen that during validation we need to examine a version from a transaction j which is not found in the buffer. In this case, we assume that the read saw the most recent version of the record and that there is no conflict. Since our experimental focus is on spurious aborts only, we

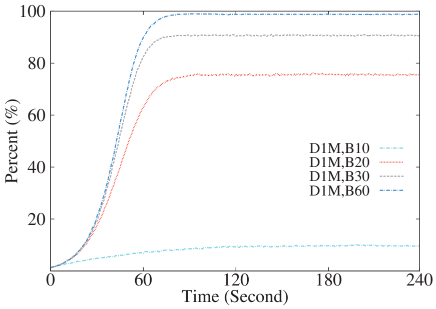

Figure 5: Abort rate with naïve truncation

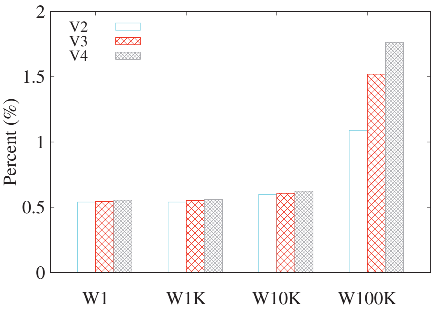

Figure 6: Abort rate with watermarking

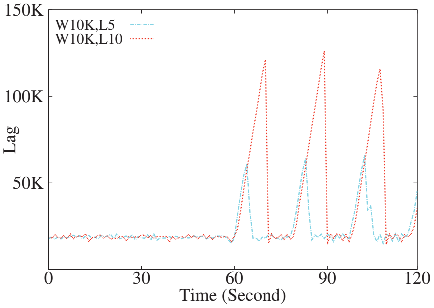

Figure 7: Watermark lag with outliers

set up our experiments so that the above assumption always holds, i.e., so that there are no "spurious commits."

We run all experiments on Amazon EC2 m3.xlarge nodes with 4 vCPUs, 15 GiB RAM and a high performance network. We use a database with one million records, with 8-byte keys and values. Each transaction performs 4 reads and 4 writes to records that are chosen uniformly at random. The concurrency level of each processor is 250. We do not overload the system, so the throughput is determined by the processor speed. We observed a throughput of about 11K transactions per second in all our runs.

Figure 5 shows the abort rate of the system with a configuration of 2 processors, 2 storage nodes, and 4 validators. As time passes, the $WriteSets$ buffers at the validators become more polluted. The abort rate therefore increases with time until it reaches a plateau; the position of the plateau rises with increased buffer size, due to a larger number of spurious conflicts. With a buffer size of 60 seconds, eventually almost all transactions abort. If we use a smaller number of validators the lines look similar but shift slightly downward.

As expected, our watermark-based approach alleviates the problem, as write sets from aborted transactions "age out" faster and spurious conflicts become less frequent. Figure 6 shows the abort rate when each processor updates its local watermark every 1 (W1), 1K (W1K), 10K (W10K), and 100K (W100K) transactions. The number of validators varies from 2 to 4 (V2, V3, V4).

When the processor updates its local watermark after every transaction completes, we observe the fewest spurious aborts. We can use this as an approximation of the true abort rate. As shown in Figure 6, it is less than 1%. This confirms that the aborts observed under the naïve approach are indeed spurious. With lower watermark update frequencies, we see the rate of aborts increase due to spurious aborts; however, even updating the processor watermark every 10K transactions is sufficient to keep the spurious abort rate very low.

Watermarking allows us to perform precise and safe validator garbage collection which correctly handles "outlier" scenarios as described previously, where a write by a transaction takes a very long time to reach storage. Figure 7 illustrates this idea by measuring the watermark lag, i.e. the maximum observed difference between the watermark of some read in a transaction and the timestamp of the transaction.

Time	V1 to V2			V2 to V3			V3 to V4		
	P	O	N	P	O	N	P	O	N
Before	8	8	0	8	4	0	8	2.7	0
During	12	8	4	10.7	4	2.7	10	2.7	2
After	8	4	4	8	2.7	2.7	8	2	2

Table 1: Number of keys in validation requests per transaction for processor (P), and processed per transaction for old validator (O), and new validator (N).

The figure shows this maximum watermark lag observed each second over a period of 120 seconds. The watermarks are updated every 10K transactions.

In the first 60 seconds, we run all transactions normally; the watermark lag is close to the watermark update frequency. Next, we introduce outliers to simulate slow storage updates. Every 20 seconds, a transaction waits due to a simulated slow write response from the storage. Until this transaction finishes, the processor is not able to advance its watermark, and the read watermarks of other in-flight transactions remain low. Figure 7 shows the effect of the outliers when the storage wait time is set to be 5 (L5) and 10 (L10) seconds. While the outlier is waiting, the watermark lag increases. When the outlier completes, the watermark is updated, and the lag drops sharply. Introducing an outlier does not appreciably increase the rate of aborts in the system, as few transactions conflict with the outlier.

5.3 Elastic scaling

Next, we examine how the system behaves during elastic validator scaling, as discussed in Section 3.4. The major overhead of this scaling process is the duplicate validation requests that processors need to send for a portion of the key space. However, this overhead decreases as the number of validators increases. For example, consider our workload where each transaction involves 8 keys. Suppose we use uniform partitioning, and when scaling from k to $k+1$ validators we shift $\frac{1}{k\times(k+1)}$ of all keys from each old validator to the new validator. Then the overhead is as shown in Table 1. The table shows the total number of keys in validation requests per transaction at the processor (P) and the number of keys processed per transaction at the old (O) and new (N) validators, before, during, and after the scaling.

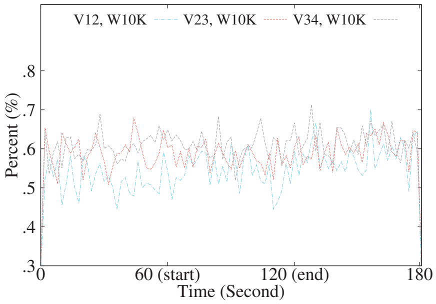

Figure 8: Abort rate of scaling experiment

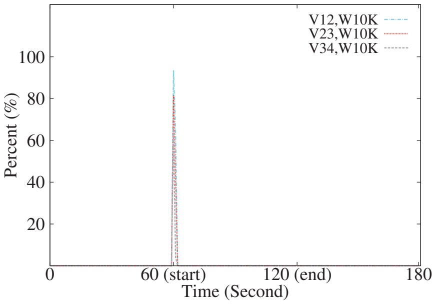

Figure 9: Unknown rate at new validator

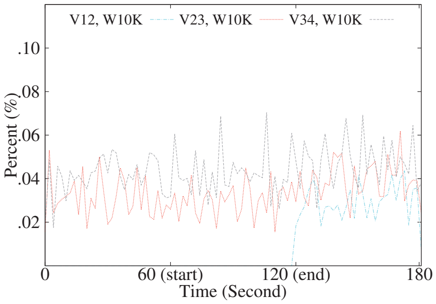

Figure 10: Spurious abort rate

We run our scaling experiments over a period of 180 seconds; we fix 2 processors and 2 storage nodes, and use up to 4 validators. We start with either 1, 2, or 3 validators and add a new validator (V12, V23 and V34 respectively). The scaling begins at the 60th second and finishes at the 120th second. The watermarks are updated every 10K transactions.

Figure 9 shows the percentage of transactions for which the new validator cannot guarantee a safe commit - i.e., those which it must "abort" due to insufficient state (the word abort is in quotes as the decisions of the new validator are not authoritative at this stage). As expected, there is a sharp peak when the scaling starts. The peak disappears within seconds, and the new validator is ready to take over. With a suitable mechanism to detect when the new validator is ready, the system can switch to the new configuration very quickly.

Figure 8 and Figure 10 show the abort rate and spurious abort rate during scaling. Both the abort rate and the spurious abort rate increase slightly after the scaling. When we scale from 1 validator to 2 validators, the spurious aborts suddenly appear once the scaling is complete. This is because no spurious aborts are possible with a single validator.

5.4 Local check for read-only transactions

Next, we investigate how many read-only transactions can be pre-validated with a local check at the processor and bypass validation (Section 3.3), how many of the read-only transactions that fail the local check actually abort, and how the local check affects throughput and latency.

To simplify the analysis, we use a workload that consists of a mix of one-shot read-only and write-only transactions. Read-only transactions may abort due to inconsistent reads. Write-only transactions always commit.

Our system has three configurations: `bypass`, `nobypass`, and `novalid`. In `bypass`, read-only transactions are checked locally and only sent to validators if the local check fails. Thus, a read-only transaction that passes the local check completes in one round trip; other transactions require two round trips. In `nobypass`, the local check optimization is disabled; all transactions are sent to validators and require two round trips. In `novalid`, transactions commit without validation, so consistency of the database is not guaranteed. Thus, `novalid` is a baseline showing the raw performance of the storage and serves only to show the overhead of enforcing strong consistency.

The database contains 100K key-value pairs. Each key and value is 8 bytes. Every transaction contains 10 key-value pairs in its read or write set. Each processor node can issue at most 500 transactions concurrently. The watermarks are updated every 10K transactions. We run 4 storage nodes, 4 processors, and 2 validators on Amazon EC2 m1.xlarge instances. Each instance has 4 vCPUs, 15GiB memory, and a 1Gb/s network.

In Figure 11, we show what happens as we increase the percentage of write-only transactions. The line `ReadOnly` shows the percentage of read-only transactions in the system - this is simply 100% minus the percentage of write-only transactions. The `ReadOnlyCommit` line shows what percentage of the read only transactions commit, and the `Bypass` line shows what percentage of the read-only transactions pass the local check and bypass validation.

When the workload is read-heavy, most transactions commit without validation. 86% of all the read-only transactions pass the local check for read-heavy workloads, and this rate is always more than 58% . In addition, a read-only transaction is sent to validators only if it is likely to abort. In workloads with 30% or more write-only transactions, more than 60% of the read-only transactions that fail the local check abort upon validation.

Figure 12 shows the throughput of the system in `novalid`, `bypass`, and `nobypass` mode. In `bypass`, when the workload is read-heavy, most read-only transactions complete in one round trip, so the performance is closer to that of `novalid`. As the number of write-only transactions increases, the throughput of `bypass` drops closer to that of `nobypass`, because more transactions require two round trips (write-only transactions and read-only transactions that fail the local check). Transaction latency exhibits a similar pattern, as shown in Figure 13. The average latency of read-only transactions in `bypass` is comparable to that in `novalid` for a read-heavy workload, and increases with the fraction of write-only transactions.

Our results demonstrate that the local check optimization is a powerful and low-overhead optimization that is especially worthwhile for read-heavy workloads.

5.5 Scalability on synthetic data

In this set of experiments, we use synthetic workloads to benchmark Centiman's performance and scalability.

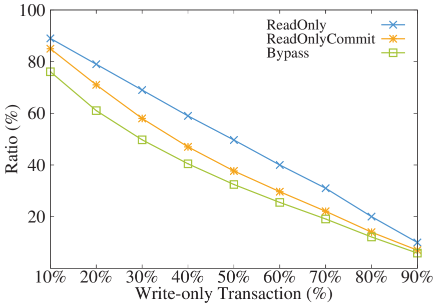

Figure 11: The percent of read-only transactions, its commit rate and bypass rate

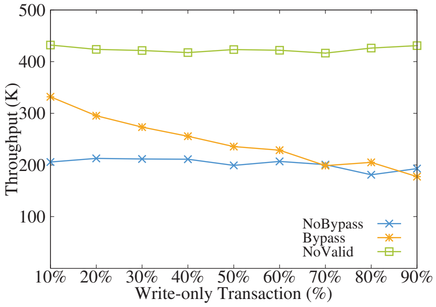

Figure 12: The throughput of the system in bypass, nobypass, and novalid mode

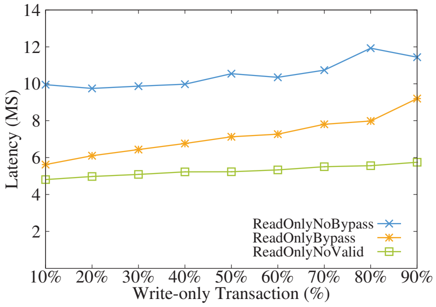

Figure 13: Transaction latency in bypass, nobypass, and novalid mode

Workload	Update-ReadOnly	Distribution
high-update	50%-50%	uniform
medium-update	25%-75%	uniform
low-update	10%-90%	uniform
high-skewed	50%-50%	power law (a=4)
medium-skewed	50%-50%	power law (a=2)

Table 2: Synthetic dataset workloads. Each workload is a mix of updating and read-only transactions. Each updating transaction updates 50% of the data. Skewed workloads follow power law distribution $P[X = x] = x^{-a}$.

We generate five workloads with different update rates and data access patterns, as shown in Table 2. Each workload has a mix of *updating transactions* (that perform writes) and read-only transactions. Each updating transaction writes half the data it accesses, e.g. a transaction of size 8 reads 4 records and writes 4 records. Each transaction accesses between 8 and 24 records. For high-update, medium-update, and low-update workloads, all data accesses follow a uniform distribution; for skewed workloads, they follow a power law distribution. The database consists of 64 million records, each with a 64-byte key and a 4-byte value. Since the value size is irrelevant for validator nodes, we chose a small value size to reduce the work at processor and storage nodes. The watermarks are updated every 10K transactions.

To stress the validator nodes and to simplify the analysis, the local-check optimization for read-only transactions is turned off, so all transactions go through validator nodes.

We run Centiman on Amazon EC2 high CPU medium instances. Each instance has 5 EC2 Compute Units, 1.7GiB memory, and a 1Gb/s network. We stress the validator nodes with a sufficient number of storage and processor nodes (with a maximum of 20 storage nodes and 20 processors).

The load on a validator node depends on the number of validation operations (conflict checks), the per-transaction processing overhead, and the cost of network message traffic. As the number of validator nodes grows, the cost of conflict checks is shared among them. However, since a transaction can be split to multiple validators, the per-transaction overhead is not reduced linearly. In addition, as we add more processors, each validator node has more network connections, and receives and sends more messages, so the overhead of networking increases.

Figure 14 shows the throughput of Centiman under a high-update workload. As expected, the throughput of the system increases sublinearly with the number of validator nodes. In addition, the throughput falls with increasing transaction size. The latter effect is not prominent, since larger transactions result in lower overhead for per-transaction processing and networking at the validator node. The throughput for medium-update and low-update workloads is similar to the high-update case. For skewed workloads, the throughput is slightly lower as we see more aborts.

Figure 15, Figure 16 and Figure 17 show the abort rate of low-update, medium-update, and high-update workloads. The abort rate increases with more validators, mostly due to feeding more concurrent transactions into the system to stress the validators. As in Section 5.2, the effect of spurious aborts is negligible. In addition, workloads with larger transactions have higher abort rates. As the size of the transaction increases, each transaction accesses more records and thus has a higher chance of conflicts. Skewed workloads demonstrate similar patterns, but with higher abort rates.

5.6 Scalability on TPC-C and TATP

Our last set of experiments explores how Centiman scales and how well the read-only local check optimization works on realistic benchmarks. To achieve this, we implement two transactions (New Order and Payment) of the TPC-C benchmark and the TATP benchmark.

For both benchmarks, we run Centiman on Amazon EC2 high CPU extra large instances. Each instance has 20 EC2 Compute Units (8 virtual cores with 2.5 EC2 Compute Units each), 7GiB memory and 1 Gb/s network. We load the system with enough clients to stress the validator nodes (with a maximum of 50 storage nodes and 50 processors). The watermarks are updated every 10K transactions.

TPC-C Benchmark. TPC-C is a traditional OLTP benchmark that represents a high-update workload of medium-size transactions. It has five types of transactions. Two of those transactions, New Order and Payment, which amount to 87.6% of the total transaction mixture, are required to run at serializability. Since the other three transaction types may run over a multi-version database at a relaxed level of consistency, we stress Centiman on a mix of New Order and Payment transactions at a 1:1 ratio. Each transaction makes 16

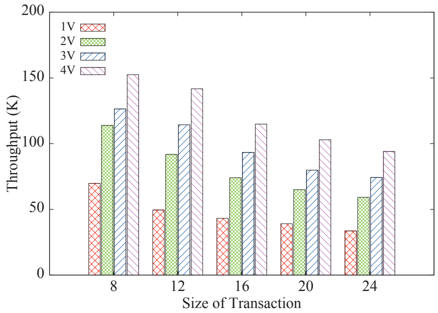

Figure 14: Throughput of high-update workload

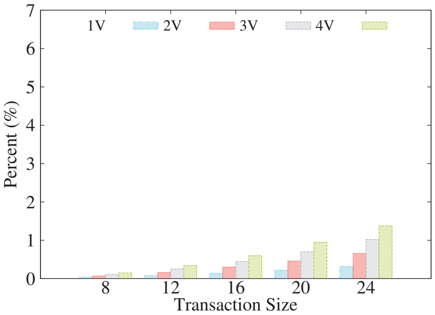

Figure 15: Abort rate of low-update workload

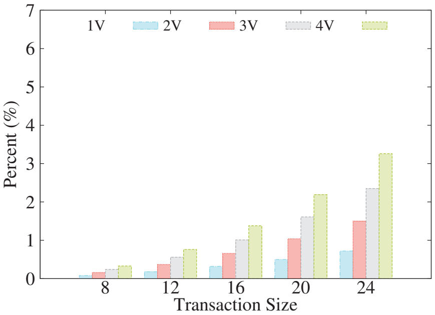

Figure 16: Abort rate of medium-update workload

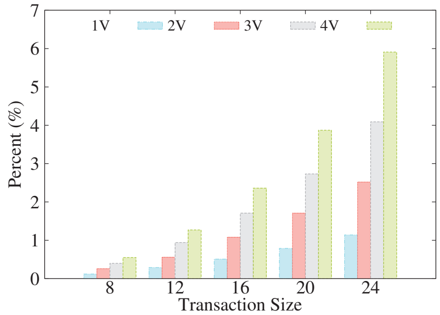

Figure 17: Abort rate of high-update workloads

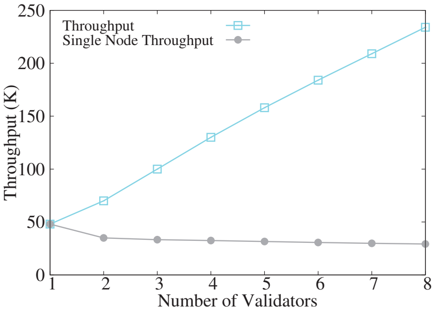

Figure 18: Throughput of TPC-C benchmark

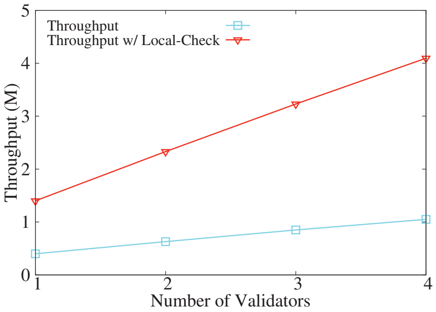

Figure 19: Throughput of TATP benchmark

reads and 15 writes on average. Since all transactions update the database, the local check optimization is not applicable.

The terminals and the data are randomly distributed over processor and storage nodes. Since we don't simulate 'think-time' in the benchmark, conflicts are frequent due to hotspots. We distribute a few records, i.e. WAREHOUSE_YTD and DISTRICT_YTD, over each terminal to avoid conflicts. We also vertically partition the WAREHOUSE, DISTRICT, and CUSTOMER tables to avoid false conflicts due to updates to different fields of a row.

Figure 18 shows the throughput of the TPC-C workload as we vary the number of validator nodes. The blue line shows the overall throughput of the system; this increases to over 230K transactions per second with 8 validator nodes. The gray line shows the throughput per validator in each system configuration. In an ideal scale-up scenario this would remain constant; in our case, the decrease is very slight. Even with 8 validators, the per-validator throughput is 89% of what it was with 2 validators.

The number of concurrent transactions (the number of terminals) scales with the size of the database (the number of warehouses) as required by the benchmark specification, so the abort rate is stable (less than 3%) in all cases. Most of those aborts are due to conflicting updates to the STOCK table in New Order transactions. Spurious aborts are rare.

TATP Benchmark. The Telecommunication Application Transaction Processing (TATP) benchmark [3] is an open source workload designed specifically for high-throughput applications. It represents a read-heavy, conflict-rare, key-value store like workload. It uses 4 tables and 7 transaction types. The workload consists of read-only and updating transactions. We use the default transaction mix, i.e. 80% read-only and 20% updating transactions. On average, each transaction issues 1.65 reads and 0.18 writes.

Figure 19 shows the throughput of the system with and without the local check optimization. We achieve over a million transactions per second with 4 validator nodes, and over 4 million with the local check optimization. Since the read-only transactions in the workload issue one single-row read, all of them bypass validation, so the local check optimization gives a huge throughput boost. The system scales well and the single-node throughput of 4 validators reaches 86% of that of 2 validator nodes. Because the database scales with the number of processors as required by the specification, the abort rate is similar (less than 1%) for all the tests.

6. Related Work

Systems with weaker guarantees than full ACID Key-value stores are the dominant system of the NoSQL movement, which embraces lightweight transaction semantics and weak consistency models. A typical key-value store supports single key-value pair operations at eventual consistency, providing excellent throughput, scalability, availability, and reliability [2, 21, 25, 39]. These systems have been successful for many use cases; however, for applications with nontrivial transaction logic, development using key-value stores can be difficult.

Other systems support transactions but explore weaker consistency models, e.g. causal consistency [43], session consistency [56], application-specific consistency models [9, 10, 14, 42] or the "classical" lower isolation levels like snapshot isolation [47] and parallel snapshot isolation [53].

Our work differs from the above in supporting full ACID-style serializability, or snapshot isolation if preferred.

Systems with partition-based guarantees The next group of systems that support distributed transactions make use of data partitioning [11, 23, 24, 35, 46, 55] by sharding data to multiple machines, and either relax consistency for transactions that span partitions, or perform more expensive cross-machine transactions for stronger consistency. These systems best fit applications where the data is easy to shard and transactions can be distributed accordingly. The performance of such systems is sensitive to how well the data is partitioned, and achieving a good partitioning may not be easy. Some follow-up work addresses those issues [24, 34, 50, 54] through clustering and online re-partitioning.

Centiman is different from the above in that the data partitioning has no impact on transactional guarantees, but does help performance by enabling sharded validation.

Systems with strong consistency guarantees Spanner [22] is a locking based system with optimizations for read-only transactions and snapshot isolation in a multi data center environment. It uses special hardware to implement the TrueTime API, which provides accurate physical clock time. We have explained (Section 4.1) that TrueTime or Logical Physical Clocks [36] can be used to implement our timestamps.

Hyder [17, 19] is a shared-data system designed for flash storage. It uses a log-structured database with a novel validation algorithm [18] to implement OCC. Essentially, Hyder employs a centralized validation design, where each node in the system performs validation of all the transactions independently, and reaches the same database state. Hekaton [27, 40] is a high performance single machine main memory system. It avoids all the complications of a distributed system, and achieves high throughput by optimizing in-memory OCC and B+ trees. Calvin [59] is a transaction scheduling and data replication layer for partitioned databases. It uses deterministic locking to reduce the contention costs associated with distributed transactions. It is best for applications where transactions can perform all their reads at once; otherwise, the deterministic locking protocol becomes prohibitively complicated. Omid [30] uses snapshot isolation with client-side replication of transaction metadata for scalability. Yet other approaches [29, 45, 49, 62] enhance concurrency by performing analysis on transaction code prior to execution. There have also been calls to rethink transaction processing architecture at a more fundamental level to eliminate unscalable communication [33].

The above systems represent a broad spectrum of architectural choices; however, most of them do not explicitly satisfy the desiderata for a cloud system: loosely coupled components and elasticity. Deuteronomy [41] is designed for elasticity, although the system still has a single monolithic TC (transaction component) rather than the multiple processors and validators of Centiman. Centiman is thus unique in privileging modularity at all levels of the system for easy scale-up without sacrificing strong consistency guarantees.

Optimistic Concurrency Control OCC [37] was proposed as an alternative to locking-based approaches and its performance has been studied extensively [8, 13, 31, 52]. Hybrid schemes combine the benefits of OCC and permissive locking [32, 57, 58, 61]. There has been a large number of extensions/adaptations of OCC to the distributed context [5–7, 20, 28, 38, 48, 51]. Among these, the systems most closely related to Centiman are Jasmin [38], although it does not use watermarks or support elastic scale-up like Centiman does, and the interval-based validation technique in [20], which is similar to our read-only transaction optimization. There is more recent work on OCC systems that combine partitioned validation with a shared log [12, 16]; the key distinguishing feature of Centiman is its simplicity, which makes it easy to scale each component independently.

7. Conclusion and Future Work

We have introduced Centiman, a high performance scalable OLTP system that provides serializability guarantees. Centiman uses OCC with sharded validation and is designed for a loosely-coupled cloud architecture; it can be deployed on top of an existing key-value store and its components scale elastically with the application's demands. Centiman avoids synchronization where possible and uses watermarks to mitigate the resulting challenges of spurious aborts and difficulties in optimizing for read-only transactions. An in-depth experimental study demonstrates the performance and scalability of the system.

In future work, we plan to explore alternatives in the system design space, such as eliminating spurious aborts by having the validators share state with each other or by having the processor broadcast commit decisions back to the validators. We also plan to extend Centiman to use OCC for low-contention settings and other protocols in high-contention settings [61], and to apply watermarking to other OCC-based systems. In addition, we would like to create a practical implementation of the system on top of a real key-value store, and see how this performs compared to other commercial distributed transaction processing systems.

8. Acknowledgments

We thank Nitin Gupta who contributed significantly to earlier versions of this project. We also thank our shepherd, Mahesh Balakrishnan, as well as the anonymous SoCC reviewers for their feedback. This work was funded by NSF Grants IIS-0911036 and IIS-1012593. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

References

[1] AmazonS3. `http://aws.amazon.com/s3/`.

[2] HBase. `http://hbase.apache.org/`.

[3] TATP. `http://tatpbenchmark.sourceforge.net/`.

[4] Apache Zookeeper. `http://zookeeper.apache.org/`.

[5] A. Adya and B. Liskov. Lazy consistency using loosely synchronized clocks. In *PODC*, pages 73–82, 1997.

[6] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In *SIGMOD*, pages 23–34, 1995.

[7] D. Agrawal, A. J. Bernstein, P. Gupta, and S. Sengupta. Distributed optimistic concurrency control with reduced rollback. *Distributed Computing*, 2(1):45–59, 1987.

[8] R. Agrawal, M. J. Carey, and M. Livny. Concurrency control performance modeling: Alternatives and implications. *ACM Trans. Database Syst.*, 12(4):609–654, 1987.

[9] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Coordination avoidance in database systems. *PVLDB*, 8(3):185–196, 2014.

[10] P. Bailis, A. Fekete, J. M. Hellerstein, A. Ghodsi, and I. Stoica. Scalable atomic visibility with RAMP transactions. In *SIGMOD*, pages 27–38, 2014.

[11] J. Baker, C. Bond, J. C. Corbett, J. J. Furman, A. Khorlin, J. Larson, J. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, pages 223–234, 2011.

[12] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck. Tango: Distributed data structures over a shared log. In *SOSP*, pages 325–340, 2013.

[13] N. S. Barghouti and G. E. Kaiser. Concurrency control in advanced database applications. *ACM Comput. Surv.*, 23(3): 269–317, 1991.

[14] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a needle in haystack: Facebook's photo storage. In *OSDI*, pages 47–60, 2010.

[15] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. In *SIGMOD*, pages 1–10, 1995.

[16] P. A. Bernstein and S. Das. Scaling optimistic concurrency control by approximately partitioning the certifier and log. *IEEE Data Eng. Bull.*, 38(1):32–49, 2015.

[17] P. A. Bernstein, C. W. Reid, and S. Das. Hyder - A transactional record manager for shared flash. In *CIDR*, pages 9–20, 2011.

[18] P. A. Bernstein, C. W. Reid, M. Wu, and X. Yuan. Optimistic concurrency control by melding trees. *PVLDB*, 4(11):944–955, 2011.

[19] P. A. Bernstein, S. Das, B. Ding, and M. Pilman. Optimizing optimistic concurrency control for tree-structured, log-structured databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1295–1309, 2015.

[20] C. Boksenbaum, M. Cart, J. Ferrié, and J. Pons. Concurrent certifications by intervals of timestamps in distributed database systems. *IEEE Trans. Software Eng.*, 13(4):409–419, 1987.

[21] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *PVLDB*, 1(2): 1277–1288, 2008.

[22] J. C. Corbett et al. Spanner: Google's globally-distributed database. In *OSDI*, pages 261–264, 2012.

[23] S. Das, A. El Abbadi, and D. Agrawal. ElasTraS: An elastic transactional data store in the cloud. In *HotCloud*, 2009.

[24] S. Das, D. Agrawal, and A. El Abbadi. G-store: a scalable data store for transactional multi key access in the cloud. In *SoCC*, pages 163–174, 2010.

[25] G. DeCandia et al. Dynamo: amazon's highly available key-value store. In *SOSP*, pages 205–220, 2007.

[26] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *PODC*, pages 1–12, 1987.

[27] C. Diaconu, C. Freedman, E. Ismert, P. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL server's memory-optimized OLTP engine. In *SIGMOD*, pages 1243–1254, 2013.

[28] R. Escriva, B. Wong, and E. G. Sirer. Warp: Lightweight multi-key transactions for key-value stores. Technical report, Cornell University, Ithaca, NY, USA, 2013. URL `http://rescrv.net/pdf/warp-tech-report.pdf`.

[29] J. M. Faleiro and D. J. Abadi. Rethinking serializable multiversion concurrency control. *CoRR*, abs/1412.2324, 2014. URL `http://arxiv.org/abs/1412.2324`.

[30] D. G. Ferro, F. Junqueira, I. Kelly, B. Reed, and M. Yabandeh. Omid: Lock-free transactional support for distributed data stores. In *ICDE*, pages 676–687, 2014.

[31] R. E. Gruber. Optimism vs. locking: A study of concurrency control for client-server object-oriented databases. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1997. URL `http://dspace.mit.edu/handle/1721.1/10762`.

[32] M. Herlihy. Apologizing versus asking permission: Optimistic concurrency control for abstract data types. *ACM Trans. Database Syst.*, 15(1):96–124, 1990.

[33] R. Johnson, I. Pandis, and A. Ailamaki. Eliminating unscalable communication in transaction processing. *The VLDB Journal*, 23(1):1–23, Feb 2014.

[34] E. P. C. Jones, D. J. Abadi, and S. Madden. Low overhead concurrency control for partitioned main memory databases. In *SIGMOD*, pages 603–614, 2010.

[35] R. Kallman et al. H-Store: a high-performance, distributed main memory transaction processing system. *PVLDB*, 1(2): 1496–1499, 2008.

[36] S. S. Kulkarni, M. Demirbas, D. Madappa, B. Avva, and M. Leone. Logical physical clocks. In *OPODIS*, pages 17–32, 2014.

[37] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981.

[38] M. Lai and W. K. Wilkinson. Distributed transaction management in Jasmin. In *VLDB*, pages 466–470, 1984.

[39] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *Operating Systems Review*, 44(2):35–40, 2010.

[40] P. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. *PVLDB*, 5(4):298–309, 2011.

[41] J. J. Levandoski, D. B. Lomet, M. F. Mokbel, and K. Zhao. Deuteronomy: Transaction support for cloud data. In *CIDR*, pages 123–133, 2011.

[42] C. Li, D. Porto, A. Clement, J. Gehrke, N. M. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *OSDI*, pages 265–278, 2012.

[43] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *SOSP*, pages 401–416, 2011.

[44] D. B. Lomet, A. Fekete, G. Weikum, and M. J. Zwilling. Unbundling transaction services in the cloud. In *CIDR*, 2009.

[45] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li. Extracting more concurrency from distributed transactions. In *OSDI*, pages 479–494, 2014.

[46] S. Patterson, A. J. Elmore, F. Nawab, D. Agrawal, and A. El Abbadi. Serializability, not serial: Concurrency control and availability in multi-datacenter datastores. *PVLDB*, 5(11): 1459–1470, 2012.

[47] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *OSDI*, pages 251–264, 2010.

[48] E. Rahm. Design of optimistic methods for concurrency control in database sharing systems. In *ICDCS*, pages 154–161, 1987.

[49] S. Roy, L. Kot, G. Bender, B. Ding, H. Hojjat, C. Koch, N. Foster, and J. Gehrke. The homeostasis protocol: Avoiding transaction coordination through program analysis. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1311–1326, 2015.

[50] M. Serafini, E. Mansour, A. Aboulnaga, K. Salem, T. Rafiq, and U. F. Minhas. Accordion: Elastic scalability for database systems supporting distributed transactions. *PVLDB*, 7(12): 1035–1046, 2014.

[51] M. K. Sinha, P. D. Nanadikar, and S. L. Mehndiratta. Timestamp based certification schemes for transactions in distributed database systems. In *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data, Austin, Texas, May 28-31, 1985.*, pages 402–411, 1985.

[52] X. Song and J. W. S. Liu. Maintaining temporal consistency: pessimistic vs. optimistic concurrency control. *Knowledge and Data Engineering, IEEE Transactions on*, 7(5):786–796, Oct 1995.

[53] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *SOSP*, pages 385–400, 2011.

[54] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era (it's time for a complete rewrite). In *VLDB*, pages 1150–1160, 2007.

[55] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing. In *VLDB*, 2014.

[56] D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. B. Welch. Session guarantees for weakly consistent replicated data. In *PDIS*, pages 140–149, 1994.

[57] A. Thomasian. Concurrency control: Methods, performance, and analysis. *ACM Comput. Surv.*, 30(1):70–119, 1998.

[58] A. Thomasian. Distributed optimistic concurrency control methods for high-performance transaction processing. *IEEE Trans. Knowl. Data Eng.*, 10(1):173–189, 1998.

[59] A. Thomson, T. Diamond, S. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD*, pages 1–12, 2012.

[60] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *ACM SIGOPS*, pages 18–32, 2013.

[61] P. S. Yu and D. M. Dias. Analysis of hybrid concurrency control schemes for a high data contention environment. *IEEE Trans. Softw. Eng.*, 18(2):118–129, Feb. 1992.

[62] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *SOSP*, pages 276–291, 2013.