Specification of a Transactionally and Causally-Consistent (TCC) database

Saalik Hatia, Marc Shapiro

Specification of a Transactionally and Causally-Consistent (TCC) database

Saalik Hatia, Marc Shapiro

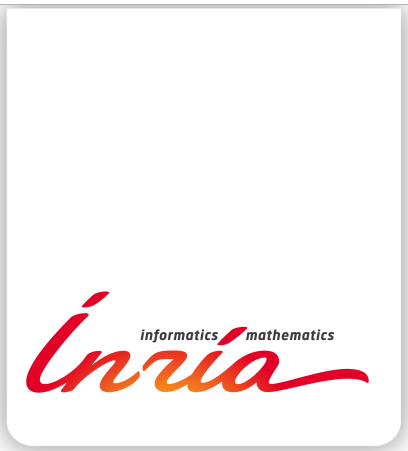
RESEARCH REPORT

N° 9355

Juillet 2020

Project-Team Delys - Sorbonne Université-LIP6, INRIA

ISSN 0249-6399 ISRN INRIA/RR--9355--FR+ENG



Specification of a Transactionally and Causally-Consistent (TCC) database

Saalik Hatia, Marc Shapiro

Project-Team Delys - Sorbonne Université-LIP6, INRIA

Research Report n° 9355 — Juillet 2020 — 16 pages

Abstract: Large-scale application are typically built on top of geo-distributed databases running on multiple datacenters (DCs) situated around the globe. Network failures are unavoidable, but in most internet services, availability is not negotiable; in this context, the CAP theorem proves that it is impossible to provide both availability and strong consistency at the same time. Sacrificing strong consistency, exposes developers to complex anomalies that are complex to build against.
AntidoteDB is a database designed for geo-replication. As it aims to provide high availability with the strongest possible consistency model, it guarantees Transactional Causal Consistency (TCC) and supports CRDTs. TCC means that: (1) if one update happens before another, they will be observed in the same order (causal consistency), and (2) updates in the same transaction are observed all-or-nothing.
In AntidoteDB, the database is persisted as a journal of operations. In the current implementation, the journal grows without bound. The main objective of this work is to specify a mechanism for pruning the journal safely, by storing recent checkpoints. This will enable faster reads and crash recovery.

Key-words: database, checkpointing, log, specification, consistency

RESEARCH CENTRE
PARIS

2 rue Simone Iff - CS 42112
75589 Paris Cedex 12

Specification of a Transactionally and Causally-Consistent (TCC) database

Résumé : Les applications à grande échelle sont généralement construites au-dessus de base de données géo-distribuées qui tournent sur de multiples centres de données répartis dans le monde. Les défaillances de réseaux sont inévitables, pourtant pour la majorité des services en ligne, la disponibilité est essentielle. Dans ce contexte, le théorème CAP prouve qu'il est impossible d'être à la fois hautement disponible et fournir de la cohérence forte. Sacrifier la cohérence forte, expose les développeurs a des anomalies complexes à gérer.

AntidoteDB est une base de données conçue pour être répartie à travers le monde. Avec pour objectif de fournir une haute disponibilité avec le plus haut modèle de cohérence possible. Elle garantit de la cohérence causale transactionnelle (TCC) et des types de données convergents (CRDTs). TCC signifie que : (1) si une mise à jour a eu lieu avant une autre, elles seront observées dans le même ordre (cohérence causal), et (2) les mises à jour appartenant à une même transaction seront vues simultanément ou pas du tout.

Dans AntidoteDB, la base de données est stockée sous la forme d'un journal d'opération. Dans l'implémentation actuelle le journal croit sans limite. L'objectif principal de ce travail est d'écrire la spécification d'un mécanisme de troncature du journal sûr, en effectuant des points de contrôles. Cela permettra d'avoir des lectures plus rapides ainsi qu'une récupération de données plus rapide.

Mots-clés : base de données, point de contrôle, journal, spécification, cohérence

Contents

1 Problem Statement

The geo-distributed database system Antidote persists its updates in a log. To recover after a crash, or to materialize a version of interest, requires to read and to (re-)execute all the corresponding operations from the journal from the initial state. As a database ages, this takes longer and longer.

The aim of this research is to speed this up by adding a checkpointing mechanism. Initializing from a recent checkpoint requires to re-execute only the operations after the checkpoint, which can be much faster. Once a full-system checkpoint exists, the preceding part of the log can be truncated.

Journaling, materializing, checkpointing and journal truncation are well-known mechanisms. However, their interplay is known to be tricky; this will be even more challenging in Antidote, because of its complex geo-distributed and sharded structure, and because of its partially-ordered consistency model.

The aim of this document is to specify rigorously, and to implement correctly and efficiently, a safe journaling, materializing, checkpointing and truncation mechanism for Antidote. This includes the following partial objectives:

- To specify the structures of interest, i.e., distributed journal, materialization cache, and checkpoint store.
- To specify the concepts of consistent cut, snapshot, object version, checkpoint, and to identify the consistent cuts that are important for correctness and performance.
- To specify rigorously the invariants that link the above structures together.
- To identify the actors of interest and their methods: transaction coordinator, version cache, journal manager, checkpoint manager, etc.
- To specify the pre-, post-, rely- and guarantee-conditions of these methods.
- To provide a reference implementation of these methods.
- To show that the implemented methods satisfy their specification.

2 Background and notation

The Antidote database is geo-distributed across a number of Data Centers (DCs). Each DC is strongly consistent; however, updates can originate concurrently at any DC under the Transactional Causal Consistency model.

2.1 Data Centers and shards

Antidote supports concurrent updates occurring in geo-distributed, highly-available DCs; each DC originates its own set of updates. In turn, a DC is partitioned into non-overlapping storage servers called shards, coordinated by Transaction Coordinators (*figure 1*). Thus, the journal is not a single sequential data structure, but is logically the union of a number of sequential *journal streams*, one per shard per DC. The originating shard is the single writer of a journal stream; all other replicas are readers. Furthermore, a stream originating in some shard in some DC is replicated to the same shard in all other DCs.[1]

[1] We ignore here the fact that a shard is itself replicated in its DC for fault tolerance, because this does not change the fact that each journal stream is sequential.

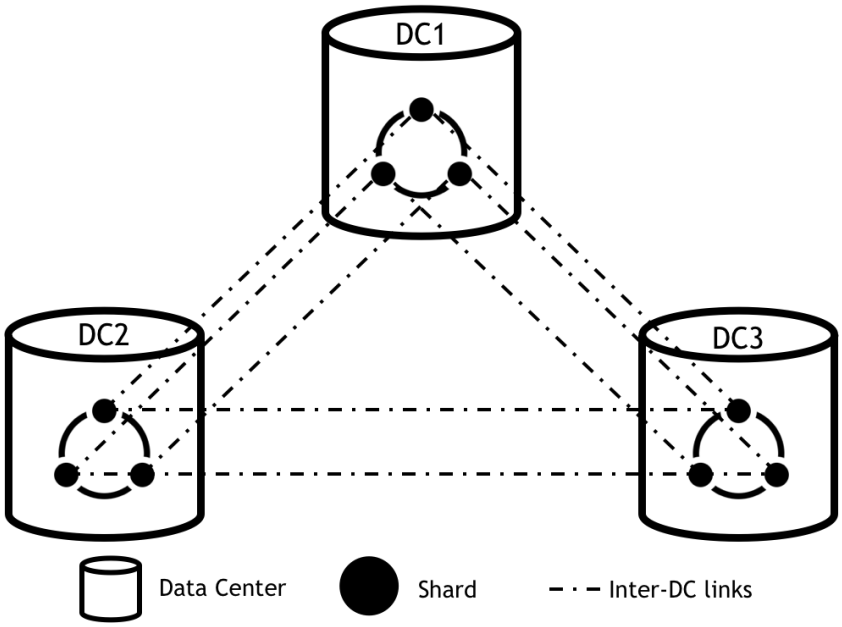


Figure 1: Overall architecture of Antidote

Zooming into a given shard at a given DC, the former contains a *local* journal stream that logs the updates to this shard originating from the latter, and a set of remote journal streams, each one replicating the updates to this same shard at some other DC. [2]

2.2 Transactional Causal Consistency

Another source of complexity is the Transactional Causal Consistency (TCC) model. Each DC is logically sequential, based on a variant of Snapshot Isolation. However, two DCs may update the database concurrently, and updates are related by the *happened-before* partial order (sometimes called causal order).

To keep track of happened-before, Antidote uses vector timestamps with one entry per DC. Every event in the database is tagged by the corresponding vector timestamp. A *consistent vector timestamp* (CVT) marks a *transactionally- and causally-consistent cut*. This means that, if a CVT contains some update u, then it also contains all updates in the same transaction as u, as well as those that happened-before u. The snapshot time of a transaction is called its *dependency CVT*, and its effects are identified by its *commit CVT*.

Recall that a given DC-shard has a local journal stream and one journal stream per remote DC. Similarly, a CVT has a timestamp per DC. We can now map each entry in a CVT (for some DC) to the prefix of the journal stream (of the same DC) that happened-before it. This cut forms a *transactionally- and causally-consistent snapshot*.

[2] In this document we ignore the possibility of adding or removing additional shards and/or DCs while the system is running.

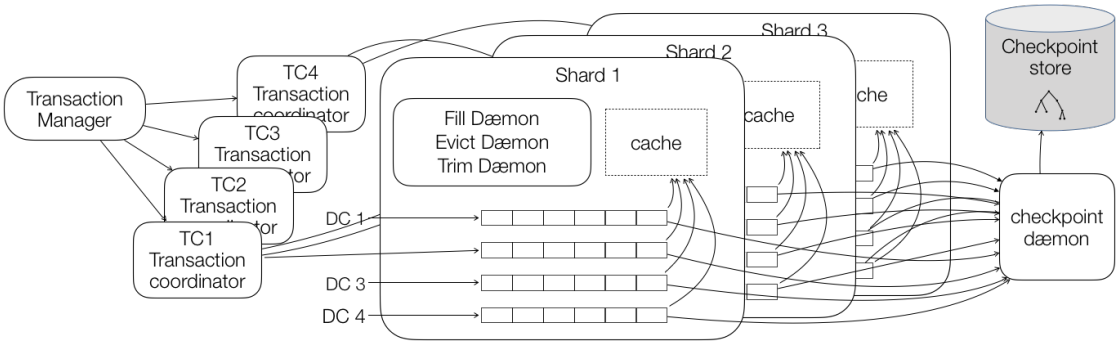


Figure 2: Inside a DC. The Transaction Manager supervises Transaction Coordinators that handle all the client transactions. Every shard has a Cache, a Journal, and three daemons with a specific role. Connected to them is a Checkpoint Daemon that creates checkpoints and stores them into a Checkpoint Store.

2.3 Journal structure

The events that impact the state of the store are persisted in a log, called the Journal herein. The Journal constitutes the ground truth of the database state. The Journal is logically composed of a number of sequential streams, each of which records the events originating from a given shard. A DC-shard's Journal stream is replicated to its counterparts at all other DCs.

2.4 Records

The events recorded in the Journal are called Records. They contain different types of information. It can be system information (such as checkpoint operations), transaction operations (such as commit message) or operations on an object. These latter operations are essential to the integrity of the database. The loss of a single record means that all the objects that are materialized afterward are incorrect. A record contains these entries:

- **Log Sequence Number (LSN):** A unique monotonically increasing sequence number
- **Timestamp**: Real-time timestamp
- **TransactionID:** a transaction identifier
- **Type:** system operation, transaction operation

Depending on the type of the operation the following information is included in the record:

- **Key:** referred object key
- **Operation:** update on the referred object with its arguments
- **Dependency:** CVT of the snapshot the transaction is reading from
- **ListOfParticipants:** participants in the given transaction
- **CommitTime:** commit time of the transaction

These are the different *Types* of records:

- **Begin:** system record that marks the beginning of a transaction. It contains a Timestamp, TransactionID, Type, followed by the Dependency. This record is written once per transaction to each shard participating.
- **Prepare:** system record that is written by shards during the first phase of the two-phase commit. It contains a Timestamp, TransactionID, Type, followed by the ListOfParticipants. The Timestamp in this context is the clock proposed by the shard as the commit time.
- **Commit:** system record that contains the commit time sent by the coordinator to the shards in the second phase of the two-phase protocol. It contains a Timestamp, TransactionID, Type, CommitTime. The commit time is sent to the shard by the transaction Coordinator.
- **Abort:** system record that marks the abortion of a transaction. It contains a Timestamp, TransactionID, Type, ListOfParticipants.
- **Update:** transaction record containing the update operation on a given object. It contains a Timestamp, TransactionID, Type, Key, Operation.
- **Checkpoint:** system record containing information about a Checkpoint. It contains a Timestamp, Type and Dependency. For each checkpoint one record is written.
- **Participant:** system record containing information when a new shard participates in the transaction. It contains a Timestamp, TransactionID, Type.

2.5 Checkpoint store

In this design we periodically persist materialized versions of objects, called checkpoints in a store called the Checkpoint Store. In the worst case after a crash, all volatile state has been lost, the current state of an object can be computed by: (1) loading the most recent checkpoint into memory, (2) reading updates that are more recent than the checkpoint from the journal, (3) applying those updates to the in-memory state.

A checkpoint created from a causally-consistent cut called Checkpoint Time (CT) (Section 3.1) to ensure that there are no consistency anomalies in the checkpoint. Once an object has been checkpointed, any earlier journal record that is part of the checkpoint for that object is irrelevant from the perspective of durability. Once all the objects in a CT have been checkpointed, then the corresponding prefix of the journal can be truncated.

3 Consistent cuts of interest

The causal ordering of events implies that their vector timestamps are ordered in the same way: If Event 1 is causally-before Event 2, their vector timestamps *vt1* and *vt2* are such that *vc1* $<$ *vc2*. Note that the converse is not true in Antidote. The timestamps of two concurrent events may be either incomparable or arbitrarily ordered (a timestamp is a safe approximation of happened-before).

The order between vector timestamps *vt1* and *vt2* is defined as followed:

- *vt1* $=$ *vt2* if every entry of *vt1* is equal to the corresponding entry of *vt2*. They represent the same event.
- *vt1* $\leq$ *vt2* if every entry of *vt1* is less or equal to the corresponding entry of *vt2*.

- *vt1* $<$ *vt2* if *vt1* $\leq$ *vt2* and *vt1* $\neq$ *vt2*. Event 1 being causally before Event 2 implies *vt1* $<$ *vt2*, but the converse is not guaranteed.
- *vt1* is incomparable to *vt2* if *vt1* $\neg \leq$ *vt2* $\wedge$ *vt1* $\neg \leq$ *vt2*. There is some entry in *vt1* that is lower than the corresponding entry in *vt2*, and vice-versa. Events 1 and 2 are necessarily concurrent, but the converse is not true; that is, if two events are concurrent, this does not guarantee that their vector timestamps are incomparable.

A vector timestamp represents a *cut* or time of the data store. We are interested only in *consistent cut* as they have properties that are useful for maintaining information about the database.

3.1 Checkpoint Time (CT)

Every time a checkpoint is created, we persist a checkpoint record in the Journal (Section 2.3).

The variable *Checkpoint Time* designates the *oldest* available checkpoint, i.e., lowest *CVT* that includes, for every object, a checkpoint whose state includes all the update records committed at any time $\leq$ *CheckpointTime*. State prior to *CT* cannot be safely recovered.

Initially, *Checkpoint Time* $= -\infty$, implying that the journal cannot be trimmed. Note that, while recovering from *CT* is safe, typically recovery will proceed from the *most recent* available checkpoint for performance reasons. To save space, a system typically stores only a few numbers of checkpoints, preferably only one. If the checkpoint store does not support versioning (i.e., each object has a single version), then there is a single checkpoint, identified by *Checkpoint Time*.

3.2 DC-Wide Causal Safe Point (DCSf)

Each shard in a DC is replicated to all other DCs. All updates that originate in some DC are sent asynchronously to the corresponding shard in other DCs. Although the shard-to-shard connection is FIFO, the storage state of different shards in the same DC is not causally consistent. Without extra care, a transaction that reads from multiple shards might be unsafe. To avoid this, a transaction should of a shard is missing updates with respect to another.

The Cure protocol [1] is what ensures the TCC properties. Cure has two main objectives:

1. To ensure that transactions in a DC commit atomically, and in a total order across all shards of that DC. It uses the Clock-SI design for this purpose [3]).
2. To ensure that that updates are observed in a causally-consistent order within a DC. To this effect, each shard continuously computes a safe lower bound for that DC, called its DC-Wide Causal Safe Point (DCSf).[3] The DCSf is a *CVT* across all shards of the DC that is *causally safe*, i.e., the corresponding updates, and their causal predecessors, have been received and persisted by all shards of this DC. States that are above the DCSf are not *visible*. A transaction whose snapshot time is not earlier than the DCSf must be blocked until the DCSf advances beyond it.

3.3 Global Causal Stable Point (GCSt)

In our new design, we will also leverage the concept of a *Global Causal Stable point* (GCSt). A GCSt is a state where all concurrent operations have been received and resolved. Formally, any updates that are delivered after the GCSt is computed will have a higher timestamp than the GCSt [2, Definition 5.1]. To simplify the logic, we assume that successive computations of a

[3] Called the Global Stable Snapshot (GSS) in the Cure paper [1].

GCSt at some shard are monotonically non-decreasing (this is always possible). To compute the GCSt each shard shares its DCSf periodically with their counterparts in other DCs. The vague GCSt is computed as the lower bound of all known DCSf.

State that is earlier than the GCSt, can be stored using its sequential representation, avoiding any concurrency-related metadata such as vector clocks or tombstones. The transitions between successive GCSt's can be explained as sequential updates. This makes the representation simpler and more compact and enable the use of any sequential database as a checkpoint store. For instance, both Add-Wins and Remove-Wins sets reduce to classical sequential sets, and RGA reduces to a classical list. We leverage this fact by choosing checkpoint states to be earlier than GCSt whenever possible, this makes the sequentially consistent.

One issue with GCSt is that it makes progress only when every single DC is available. It stops advancing as soon as any single DC does not regularly communicate its metadata.

3.4 Min-Dependency and Max-Committed

In order to satisfy the properties of checkpoints, of the DCSf and the GCSt we keep track of all the dependencies of running transactions but also all the commit times of finished transactions.

Among those we single out those who are the most important.

Min-Dependency represents the oldest snapshot any running transaction is reading from. Because a checkpoints is sequentially consistent, there should be no in-flight transactions at Checkpoint-Time. To ensure that this property holds true Min-Dependency is used to track the point beyond which sequentiality is not guaranteed. When a transaction is finished, committed or aborted, the Min-Dependency advances to the next Dependency. One issue is that while a transaction is running, Min-Dependency will not advance, and consequently checkpointing will be paused.

Max-Committed is the last commit time recorded in the Journal. DCSf's advancement is bounded by Max-Committed, making sure that a new transaction does not read unsafe updates. Every time a transaction commits, its commit time become the new Max-Committed. Similarly to Min-Dependency, if no transaction commits, Max-Committed does not advances, nor does DCSf.

3.5 Low-Watermark and High-Watermark

To represent the persistent portion of the log we use Low-Watermark and High-Watermark. With *Low-Watermark* representing the lower bound and *High-Watermark* the higher bound. Records that precedes *Low-Watermark* may be deleted and records that postdates *High-Watermark* might be volatile.

3.6 Invariants

The overarching goal of this work is to have no perceivable loss of information, meaning that records that have not been checkpointed must not be deleted. Hence, our first invariant is $Low \leq CheckpointTime$. Checkpoints should be sequentially consistent and stable across all DCs. Hence, the invariants $CheckpointTime \leq GCSt$ and $CheckpointTime \leq Min\text{-}Dependency$. By construction the GCSt is computed as the lower bound of all known DCSfs across so this gives us the following invariant $GCSt \leq DCSf$. DCSf represents the point of safety in a DC using shared information between shards about their registered *commit times* therefore $DCSf \leq Max\text{-}Committed$. These relations control the behavior of each shard (*figure 3*).

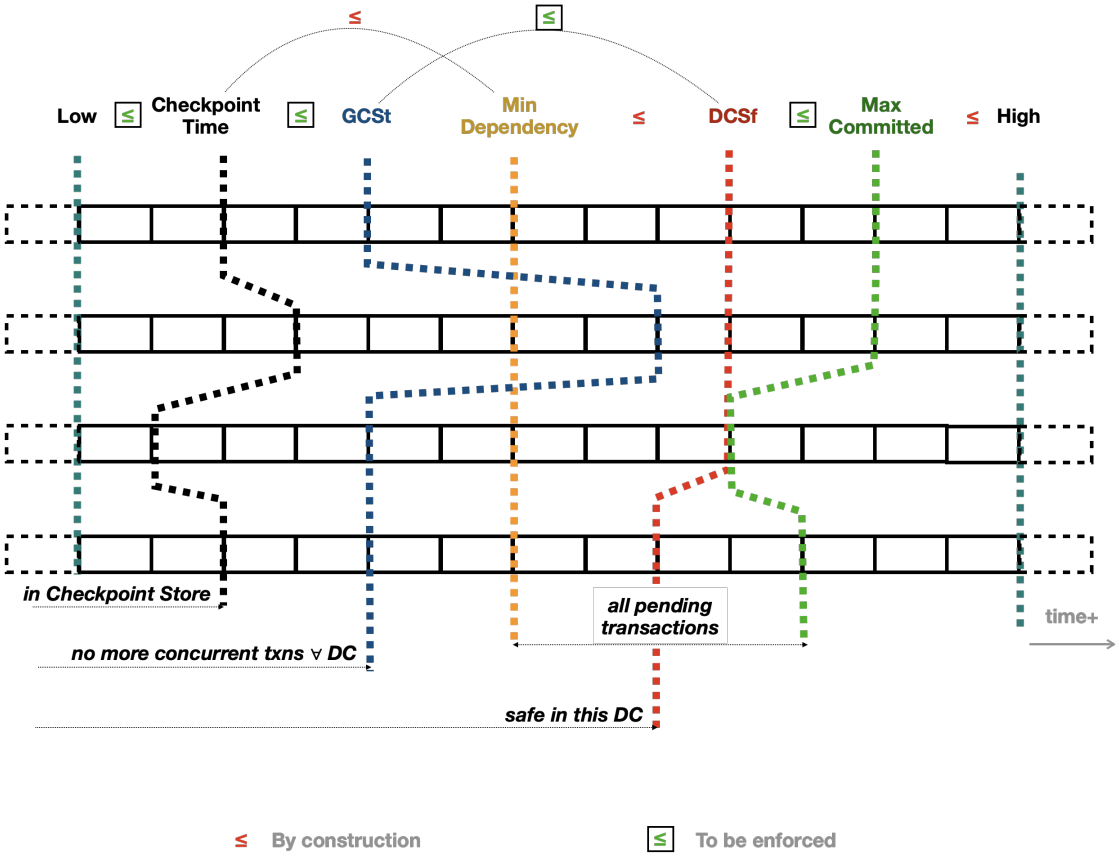


Figure 3: Relevant system states and their relations (for a given shard, at a given DC). Each horizontal tape (one per DC) depicts a journal stream for this shard. Each vertical line depicts a cut of interest and its vector timestamp. Over time, each journal stream grows to the right (and is trimmed to the left), and each state of interest advances monotonically to the right. As this happens, the causal precedence invariants, denoted $\leq$, must be maintained. To enforce $\leq$ requires synchronization between the corresponding processes.

At the Journal level there are several invariants that are necessary to ensure that no data is lost. First if a Checkpoint exist there should be a *Checkpoint* record present in the Journal. All records from a non-checkpointed transaction must have all records present in the journal.

4 Actors

4.1 Transaction Manager

Each DC has a process called the Transaction Manager, which receives transactions from clients. When a client's transaction is received at a DC, it creates a Transaction Coordinator to manage it. Every Transaction Coordinator generated is supervised by the Transaction Manager. It ensures that they handle the transaction and terminate correctly. If a Transaction Coordinator crashes before sending a termination signal the Transaction Manager creates another Transaction Coordinator to recover the transaction.

4.2 Transaction Coordinator

A client transaction is managed by a single process called its Transaction Coordinator, running on some arbitrary server of the DC where the client is located. The role of the Coordinator is to start the transaction at each involved shard, to send each client operation to the correct shard, and to coordinate the two-phase commit of the transaction among all the shards.

A Transaction Coordinator assigns it a unique transaction ID and a *Dependency CVT*. When the transaction issues an operation on some object, the coordinator forwards it to the appropriate shard in the same DC.

4.3 Checkpoint Daemon

The Checkpoint Daemon periodically gets materialized versions from the cache and persist them in the Checkpoint Store. The eviction daemon should signal its intent to evict to the Checkpoint Daemon, enabling the latter to checkpoint opportunistically before the eviction occurs. Furthermore, when the trimming daemon wants to advance, it signals the Checkpoint Daemon to make progress. Because we want the Checkpoint Daemon to be sequentially consistent it may not advance *Checkpoint Time* beyond Min-Dependency.

If the Checkpoint Daemon conflicts in this way with a running transaction, the daemon must wait for the transaction to terminate, possibly by asking the Transaction Coordinator to forcibly abort it.

4.4 Trim Daemon

The Trim Daemon's role is to truncate the Journal. It gets a signal from the Checkpoint Daemon when it has finished a checkpoint. Upon reception of the signal the Trim Daemon scans the Journal to find the first begin record of a transaction that is not part of the last transaction. Low-Watermark is set with the Log Sequence Number of this record.

4.5 Fill Daemon

The Fill Daemon ensures that the objects that are needed in the cache are properly materialized. When an object version is missing, the Cache sends a signal to the Fill Daemon so it can retrieve the needed information in order to build the object.

In order to build an object, the Fill Daemon looks if a version already exist in the cache. If none, then the Fill Daemon retrieves the last checkpointed versions from the Checkpoint Store. If there is no information about the key in the Checkpoint Store, it returns an empty object. The Fill Daemon then reads all the operations from the journal that have been made and committed since.

If the Fill Daemon cannot fill the cache because the *occupancy* is full, then it sends a signal to the Evict Daemon 4.6.

4.6 Evict Daemon

The Evict Daemon is triggered by either a system policy, or the Fill Daemon, in order to free up space. If it is triggered by a system policy, then it sends a signal to the checkpoint daemon so that objects in the cache can be opportunistically checkpointed if needed.

5 Cache

The cache is a data structure of materialized versions of objects that are currently in use. An entry in the cache is identified by a pair *(k,c)* where *key* is the key of the object, and c is the commit timestamp of the version. Each shard maintains its own cache managed by a cache daemon.

5.1 Object-version descriptors

The system maintains the following information about an object's version in an object-version:

- *key (abbreviated k)* is the unique identifier of the object.
- *commit (abbreviated c)* is the commit vector timestamp of the transaction this object is materialized from. An uninitialized object-version has a vector timestamp of $-\infty$.
- *presence (abbreviated p)*: the presence bit, it is true when the descriptor is significant. Otherwise, it must be ignored.
- *valid (abbreviated v)* is a vector timestamp for which the object has not been updated. In other words, it is known that there exist no committed updates between *commit* and *valid*. Equivalently, this version is part of all snapshots between these two values.
- *used (abbreviated z)* a bit that indicates that this version has been recently been used; useful for cache management.
- *blob (abbreviated b)* is a pointer to the materialized value of the object (if any), either in memory or on checkpoint store. The system does not interpret the blob, which is managed by the object's type.[4]

5.2 Functions

The cache has four main functions:

- *get(key k, vts dep)* returns the object k from the snapshot *dep*. This is called by application and by the checkpoint daemon.

[4]In Antidote, the type is itself encoded as part of the object's key. Antidote supports CRDT types such as Counter, LWW-register or AW-set, and quasi-CRDTs such as Bounded Counter.

- *load(key k, vts c, blob b)* Loads an object k from the checkpoint store, and assigns it a c as timestamps. Called by cache daemon.
- *inc(key k, vts c, vts t)* apply operations committed on object k from between time c and t. Called by cache daemon.
- *evict(key k, vts c)* evicts object k timestamped c from the cache. Called by cache daemon.

A cache entry (e) is represented by its primary key (*key k, vts c*).

5.3 Reading

The main application API is *get*, which accesses the materialized version of object with key k at a snapshot with dependency vector *dep*. Its precondition requires that there exists a suitable object-version descriptor e in the cache.[5] It must have the same key k, and a commit time c that satisfies the dependency, i.e., such that $c \leq dep$ and there are no invalidating updates $dep \leq v$, where v is e's validity timestamp. The cache entry must be significant (presence bit p) and this version must be *visible* to the client.

The assertion *visible* abstracts the visibility conditions. A version is visible to the client that wrote it (read-my-writes). If the client is not writing the transaction, the version must be committed and causally stable to ensure all distant updates have been received. Furthermore, *visible* requires that access control conditions (not addressed in this document) are satisfied [4].

If the precondition is not satisfied, *get* may either fail, or (in the case of a cache miss) block and wait. To this effect, it might send a signal to the cache-filling daemon discussed next.

When the precondition is satisfied, *get* returns the object and sets the used bit z to true.

The method relies on the environment not removing or invalidating the cache entry, and not resetting the used bit. Thus, a rely condition forbids the checkpointing daemon from removing old checkpoints too aggressively, as *checkpointed*, the time of the oldest checkpoint, may not advance further than the *dep* of any executing transaction.

The guarantee clause guarantees to modify no more than the used bit of cache entry e.

5.4 Filling the cache

Load allocates and initializes a suitable descriptor e and increments *occupancy*. If the checkpoint store records the validity timestamp, initialize it accordingly; otherwise we initialize it to be equal to a checkpoint timestamp.

The guarantee clause of *load* guarantees to not update memory outside of e. Its rely clause assumes the environment does not overflow the cache nor update e (other than the used bit). This may require synchronization with *inc*.

The *inc* method updates an existing cache entry e = *(k,c)* with any updates from the journal more recent than c up to some time t. The result is a cache entry e' = *(k, c')*, with the same key and a larger commit time. The descriptor e' may be either the same as e (overwrite), or a newly-allocated one.

The precondition blocks until the journal has caught up with t, and requires that e was previously *load*ed, since e must exist with its presence bit true, i.e that, the method initializes e' by applying (in causal order and respecting transaction atomicity) all journal records up to t that are either concurrent to, or later than e. The resulting commit time is the least upper bound of these updates, and the validity timestamp is t. If e' was a new entry, the method increments the occupancy.

[5] A cache entry is named by its primary key *(key k, commit_ vts c)*.

Filling the cache is handled by a daemon called Fill Daemon explained later in this document (*see 4.5*).

5.5 Eviction

The *evict* method invalidates some entry e that is present and not used; in practice, it will be triggered either periodically, or by a signal from the Fill Daemon. Importantly, if *get* sets the *used* bit during its execution, the method exits with no effect. Otherwise, the method simply sets the presence bit to false and decreases the occupancy.

The rely clause state that the environment does not modify e, except for evicting, or setting the *used* bit. The guarantee clause guarantees that, if the *used* bit changes, the method has no effect; otherwise, it modifies only the presence bit.

Eviction is handled by a daemon called Evict Daemon, discussed later in this document4.6.

6 Recovery

After a restart, due to a scheduled restart or a crash, the database needs to reconstruct a stable view of the database. The Transaction Manager is the first process to start, it initializes each shard. Each shards then starts a recovery algorithm. The journal in some shard might have incomplete information about transactions. The goal of the recovery is to reconstruct the information that is necessary for the database to run correctly and ensure that Journal invariants are true.

When recovery is complete the following is true at each shard:

- Time of the last checkpoint is known
- Low Watermark and High Watermark are set
- Every Transaction that has a prepare record in each participating shard also has a commit record in each corresponding Journal.
- Every Transaction that is missing a prepare record has an abort message written in each corresponding Journal.

Recovery proceeds as follows. First the Journal is read to find the last checkpoint record that has been written to the log and set the Checkpoint Time. Using the Checkpoint Time, and the fact that every committed transaction that is not part of the last checkpoint must be complete in Journal. Low Watermark is initialized with the Log Sequence Number (LSN) of the first *Begin* record that comes after the Checkpoint Time. The High Watermark is then initialized with the LSN of the last complete record in the journal with no gaps.

Next, recovery scans the Journal for transactions that have a record in the journal between Low Watermark and High Watermark, and put their IDs in a set called *transactionIDs*. Then two additional sets of transaction IDs, one for transactions with a commit record, called *committed*, and the second for transactions with a prepare record, called *prepared*.

Using these three sets a check of the invariant $committed \subseteq prepared \subseteq transactionIDs$ is performed. Should this test fail the recovery will fail.

For every transaction that has no commit record or prepare record, recovery adds an abort record in the journal in every participating shard.

For every transaction that has a prepare record but no commit record recovery has to re-run two-phase commit. It sends its ID to the Transaction Manager. The Transaction Manager verifies that all participating shard has a prepare record, and that none has an abort record. If

true, and none have a commit record, then two-phase commit is initiated, otherwise the commit record is propagated to the other shards. If not an abort record is sent to every participating shards.

Once all shards have complete journals, the database is ready to operate.

7 Current and future work

While the specification of the concepts of the database are taking shape, the communication between DCs needs to be specified as well.

Once recovery semantics have been set, specifying the lifecycle of the database will allow us to verify that there is no obvious hole in our design. The next step will be writing the related pseudo-code and formally verify it. And finally implement and validate through experimentation.

Once all this work is done, there are a few enhancements the design would benefit from. In the current design all the DCs are sharded identically. If one DC was to add another shard, this change should be reflected on other DCs, considering communication is shard-to-shard and the protocol is designed to work between identical shards. Allowing DCs to scale asymmetrically would be a welcomed improvement as DCs might not have the same workload.

Another objective would be to be able to use multiple storage backends, including legacy databases or file systems without changing their native format. Antidote would become the transactional and replication layer adding support for TCC on top.

8 Conclusion

This report presents the work on a design of a database that enables safe truncation of the Journal without any observable loss information. It combines operations and checkpointed states to reduce the response time associated with having only operations. Truncating the Journal while checkpointing during normal operation of the database is tricky, and if not properly specified, can lead to data loss.

A secondary goal of this design is to create a blueprint for a TCC database that supports storing states alongside the Journal of operation.

References

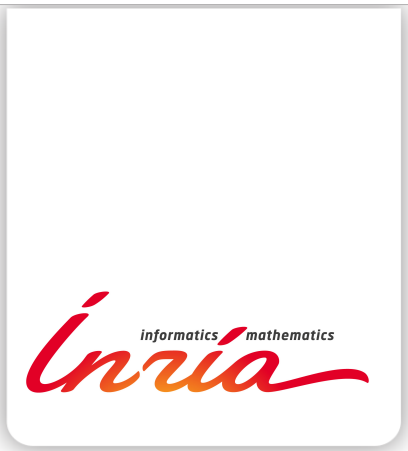
[1] Deepthi Devaki Akkoorath, Alejandro Z. Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno Preguiça, and Marc Shapiro. Cure: Strong semantics meets high availability and low latency. In *Int. Conf. on Distributed Comp. Sys. (ICDCS)*, pages 405–414, Nara, Japan, June 2016.

[2] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. Pure operation-based replicated data types. ArXiv e-print 1710.04469, arXiv Computing Research Repository (CoRR), October 2017.

[3] Jiaqing Du, Sameh Elnikety, and Willy Zwaenepoel. Clock-SI: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *Symp. on Reliable Dist. Sys. (SRDS)*, pages 173–184, Braga, Portugal, October 2013. IEEE Comp. Society.

[4] Mathias Weber, Annette Bieniusa, and Arnd Poetzsch-Heffter. Access control for weakly consistent replicated information systems. In Gilles Barthe, Evangelos P. Markatos, and

Pierangela Samarati, editors, *Int. W. on Security and Trust Management*, volume 9871 of *Lecture Notes in Comp. Sc.*, pages 82–97, Heraklion, Crete, Greece, September 2016. Springer-Verlag. ACGreGate.



RESEARCH CENTRE
PARIS

2 rue Simone Iff - CS 42112
75589 Paris Cedex 12

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399