# Persisting the AntidoteDB Cache: Design and Implementation of a Cache for a CRDT Datastore

Ayush Pandey, Annette Bieniusa, Marc Shapiro

# Persisting the AntidoteDB Cache: Design and Implementation of a Cache for a CRDT Datastore

**Ayush Pandey, Annette Bieniusa, Marc Shapiro**

**RESEARCH REPORT**
**N° 9470**
January 2022
Project-Teams DELYS

ISSN 0249-6399 ISRN INRIA/RR--9470--FR+ENG

# Persisting the AntidoteDB Cache: Design and Implementation of a Cache for a CRDT Datastore

Ayush Pandey*, Annette Bieniusa*, Marc Shapiro †

Project-Teams DELYS

Research Report n° 9470 — January 2022 — 65 pages

**Abstract:** Many services, today, rely on Geo-replicated databases. Geo-replication improves performance by moving a copy of the data closer to its usage site. High availability is achieved by maintaining copies of this data in several locations. Performance is gained by distributing the data and allowing multiple requests to be served at once. But, replicating data can lead to an inconsistent global state of the database when updates compete with each other.

In this work, we study how a cache is designed and implemented, for a database that prevents state inconsistencies by using CRDTs. Further, we study how this cache can be persisted into a checkpoint store and measure the performance of our design with several benchmarks. The implementation of the system is based on AntidoteDB. An additional library is implemented to realise the discussed design.

**Key-words:** Cache, Replication, Journal based database, Journal Indexing, Transactions

* TU Kaiserslautern
† Sorbonne Université

RESEARCH CENTRE
PARIS

2 rue Simone Iff - CS 42112
75589 Paris Cedex 12

# Persistance du cache d'AntidoteDB : Conception et mise en œuvre d'un cache pour un datastore de CRDT

**Résumé :** De nombreux services reposent aujourd'hui sur des bases de données géo-répliquées. La géo-réplication améliore les performances en rapprochant une copie des données de leur site d'utilisation. La haute disponibilité est obtenue en maintenant des copies de ces données à plusieurs endroits. Les performances sont améliorées en distribuant les données et en permettant à plusieurs requêtes d'être servies en même temps. Cependant, la réplication des données peut conduire à un état global incohérent de la base de données lorsque les mises à jour sont en concurrence les unes avec les autres.

Dans ce travail, nous étudions la conception et la mise en œuvre dâun cache, pour une base de données qui convergente utilisant les CRDTs. De plus, nous étudions comment persister le cache en en stockant des instantanés ; enfin, nous mesurons la performance du système ainsi conçu grâce à plusieurs bancs dâessai. La mise en œuvre est basée sur AntidoteDB, comme une bibliothèque.

**Mots-clés :** Cache, Réplication, Base de données basée sur le journal,Indexation du journal, Transactions

# List of Figures

# Contents

# 1. Introduction

In 2021, Cisco released a forecast report that presented the trends in internet usage and data production [14]. The key revelations were surprising.

- Globally, the internet traffic will reach 35 Gigabytes per capita in 2021, up from 13 Gigabytes per capita in 2016.
- In 2021, the gigabyte equivalent of all movies ever made will cross Global IP networks every 1 minutes.

This exponential increase is a side effect of technologies like smart devices, 5G, AI etc. These factors, compounded with the fact that more people prefer working remotely and virtually due to the trends set up by the pandemic, increase the demand for edge devices and services that support them.

Although edge computing holds relevance towards solving the problems of maintaining the quality of service across computational and geographic boundaries, this is only achievable if the necessary driving technologies are also present. The major aspect of improving performance at the edge is data management. This is where traditional RDBMS systems are not the most suitable candidates. This is due to several reasons.

- Conventional DBMS systems require central coordination for transaction management and replication.
- Synchronising updates requires precise time keeping and clock management.
- Converging the state requires consensus.

Unlike relational database systems, geo-replicated databases are designed with a globally replicated state in mind. Ideally, a geo-replicated database unit, which can be a single cluster or data-center, is independent and capable of functioning on its own without having to depend on redundant or remote replicas. Geo-Replicated databases base their design on a set of constraints that are very different from RDBMS solutions.

- Networks between replicas are unreliable.
- Precise timekeeping is not possible across boundaries.
- Network patterns and data store usage is unpredictable.
- Consensus at scale is highly expensive and also slow.
- Central coordination mechanisms have a limit on the maximum number of participants in the state management process.

## 1.1. Background

In database deployments at scale and in distributed systems generally, network partitions are inevitable. When a partition occurs, there are two options to keep maintain availability. We can respond to requests with an error message and maintain a consistent internal state of the system or we can respond to requests based on the available state and accept updates to the data which that lead to an inconsistent state later. This leads us to an impossibility result. The CAP theorem [17] states:

**Theorem 1.1.1.** *It is impossible, in the asynchronous network model, to implement a read/write data object that guarantees the following properties:*

- *Availability*
- *Atomic Consistency (see Definition 2)*

*In all fair executions (including those in which messages are lost).*

It also proceeds to say that, while it is impossible to have Consistency, Availability and Partition tolerance for an asynchronous network at the same time, guaranteeing any two of the three is possible. This is only true in scenarios when a network partition happens. If the communication channel supporting our system is stable, then both consistency and availability can be achieved.

Assuming a stable network between components, it is possible to receive updates for the same object that originate in different replicas. Such a scenario, in the absence of a central coordinator will lead to an inconsistent state and restoring consistency in the presence of conflicts will require dropping some updates. This is a problem that a lot of distributed computing algorithms deal with and try to solve.

There exists another approach. *Optimistic replication* [31] allows replicas to diverge by accepting all the updates, even the ones that can possibly lead to an inconsistent state. This strategy requires a change in the underlying data structures that store the data to ensure that inconsistencies can be resolved later when the replicas want to coordinate and synchronise their state. This special class of data structures is called CRDTs [28].

## 1.2. CRDTs

CRDT is an acronym for Conflict-Free Replicated Datatype. A CRDT mitigates the conflict created due to competing concurrent writes to the same data by making sure that the linearization (See Definition 2) of operations does not affect the final state of the data i.e. the order of execution of operations has no effect on the resulting data produced by those updates. Mathematically, the conflict resolution rules in CRDTs allow the operations to always merge or resolve. Based on the type of CRDT in use, operations can be *commutative* but may not be *idempotent* as it is the case with Operation Based CRDTs [11] or *commutative*, *associative* and *idempotent* as is the case with state based CRDTs [12].

A simple example of a CRDT is a *counter*. With a counter, the operations available are *increase* and *decrease*. Any order of execution of any combination of these operations on a counter object will always result in the same final value. In contrast, for a *set* where the operations are *add* and *remove*. Depending on the operation executed last on the set, an element might or might not be present in the set. CRDTs introduce additional semantics to resolve these problems. Some examples of these new data-types are *Last-Writer-Wins Set*, *Add-Wins Set*, *Multi-Valued-Register* etc.

## 1.3. Applications of CRDTs

The conflict resolution rules of CRDTs make them suitable for use in the design of complex distributed applications with replicated state. Some of which include

#### Collaborative editing [7]

- WOOT Framework[24] uses CRDTs in Peer to Peer collaborative Editing.
- Yjs [23] which is a framework supporting collaborative editing uses CRDTs as shared types.

#### Distributed file systems [6]

- ElmerFS [40] is a distributed file system based on CRDTs.
- Inter Planetary File System (IPFS) [19] uses CRDTs in the consensus component.

#### Distributed Processing Frameworks

- Logoot [42] is an algorithm that ensures CCI consistency for replicated linear structures and CRDTs, CRDTS in logoot framework are called logoots, and they are the main unit of data that is shared.

## 1.4. Related Work

Several DBMS products exist that use a wide variety of replication to facilitate consistency. Some products related to our work are the following.

**Amazon Aurora [41]** Aurora is a relational database provided as a part of the AWS suite for OLTP workloads. Aurora was designed for durability in the cloud at scale. It uses the concept of Availability Zones (AZs). AZs are regions that maintain connection to other AZs to coordinate replication. The design of AZs is such that the failures remain isolated. Aurora can tolerate a failure of a single AZ without losing the ability to write. An additional node can be lost from another AZ without impacting the read performance. The drawback of Aurora is the need for consensus. The nodes participating in consensus

are always located in different AZs by design which makes consensus expensive and in case of AZ failures, impossible. Another strategy that Aurora employs is the use of the log as the true data-store. The recovery and state generation is done through the log. This strategy is mirrored in our implementation.

**ArangoDB [9]** ArangoDB uses a leader-follower approach to achieve replication. The operations are written to a write ahead log and synchronised by the leader of the replica. The operations within a cluster are synchronously synchronised and asynchronous replication is used across replicas. In a single cluster of arangoDB, the followers are eventually consistent. In arangoDB, scalability can be achieved by horizontally scaling the system, but it increases the cost of replication. This is because the leader is the sole entity responsible for managing the log and the follows replicate the order of operations written by the leader. This approach sacrifices availability for consistency. AntidoteDB scales horizontally in a similar fashion by adding vnodes.

**DynamoDB [15]** DynamoDB is based on a dynamo style architecture which features consistent hashing for partitioning and replicating the data. Objects in dynamoDB are versioned and indexed by a primary key which is enough for a lot of Amazon's services like shopping carts and seller lists. Our implementation makes use of a lot of these strategies as well. The major difference between dynamoDB and AntidoteDB is that consistency for updates is maintained by a quorum-like technique in dynamoDB whereas AntidoteDB uses CRDTs for maintaining a consistent state.

**Riak KV [39]** Riak KV is an eventually consistent key-value data store that offers high availability and distribution. It is based on the riak core framework which, like dynamoDB, is built on a dynamo style architecture. Riak core lite, a simplification of riak core is used within AntidoteDB as well. This leads to a lot of similarities between the two products. Both riak KV and AntidoteDB have dynamic reconfiguration capabilities where nodes can enter and leave the system without an extra operational burden. A major difference between riak KV and AntidoteDB is that AntidoteDB supports transactions while riak KV does not.

**Google Spanner [10]** Spanner is a globally-distributed data management system that supports a strongly typed schema system. Spanner uses a shared write ahead log. The operations to this log are written after the replicas agree on a specific value by consensus through paxos [21], more specifically multi-paxos [20]. In spanner the replication is done by sharing the log. The state of the replicas is then constructed by replaying the log. This is similar to how replication in AntidoteDB functions. The operations to the CRDTs are written to the log which is then shared. The major difference stems from the non-use of consensus. AntidoteDB doesn't require consensus because CRDTs help in converging the replicated state.

**Redis [13]** Redis is a distributed key-value store that can be configured to run in-memory or utilise a persistent storage like a disk. The in-memory data in redis is persisted as a point-in-time snapshot which reflects the state of the memory at any given time. This process can be expensive if the memory occupancy is high. Unlike AntidoteDB, redis uses a leader follower approach for synchronising the state between the replicas.

## 1.5. AntidoteDB and Just Right Consistency

AntidoteDB is a highly available, Geo-replicated database that supports transactions. It makes use of CRDTs to store data objects and maintains strong consistency within a data-center by using highly available transactions. Strong consistency ensures that the operations within a data-center are seen by all the shards of that data-center in the same order.

AntidoteDB chooses a consistency model that compromises between strong and eventual consistency to manage the internal state. The consistency semantics are based on the requirements of the application. This is referred to as *Just Right Consistency (JRC)* [32]. The consistency guarantee provided by AntidoteDB is Transactional Causal Consistency (TCC). TCC is a combination of Atomicity and Causal consistency [8].

In an application programming model that guarantees TCC, we have the following properties:

- Transactions read from a causally consistent snapshot of an object which contains the effects of the transactions that causally precede it.
- Transactions updating the objects, abide by atomicity.

## 1.6. Overview and Problem Description

It is clear that CRDTs help the replicas in a distributed database converge; but there are strategies that need to be implemented such that this reconciliation happens without faults and leads to a consistent view across replicas. In AntidoteDB, CRDTs represent data objects but by design, the actual objects are never stored in a persistent storage. Rather, a sequence of operations, which when applied to a base version of the CRDT, create the object and this sequence of operations is stored in a persistent store as a journal.

This leads to an insidious problem because reading and writing from persistent storage is expensive. Also, objects can be made up of many operations which might not be consolidated in the log. The log contains operations issued by several transactions. The transactions that run concurrently will also write their operations to the journal concurrently and the operation sequences for these transactions will be interleaved. Building a single object requires filtering all the relevant operations and materialising them. This filtering operation is also expensive. We study in this work how an object cache for AntidoteDB is designed and populated. In this context, we present the internal design of calls within AntidoteDB and how they affect components like the cache.

To limit the size of a journal and reduce the need for reading the operations, a checkpoint store which contains snapshots of the objects can be created. We study how to design such a checkpoint store and how storing objects in the checkpoint store helps, in turn, towards reducing the size of the operations log. An indexing mechanism for the journal is also designed and studied for its impact on performance.

The design is supported by relevant benchmarks which are presented along with the hypotheses that motivated the design of the components of AntidoteDB.

# 2. System design

## 2.1. Recording Operations - AntidoteDB Journal

The journal or the operations log, forms the backbone of the database. It keeps track of the order and type of the operations performed on different objects in a DC as well as a record of the transactions. The journal serves the following major purposes in this regard.

- Record the updates performed on the objects along with the transactions that updated them.
- Record transaction coordination steps and the state of the 2 phase commit for a transaction i.e. *start, prepare, commit* etc.
- Facilitate materialization of the objects for responding to queries.

AntidoteDB's journal is made of log records. Each record consists of parameters that identify its characteristics like the type, name and number. The structure of a log record is shown in Figure 2.1. The journal grows as the number of objects and the operations performed on them increases.

## 2.2. Materialization - Using AntidoteDB Journal to create Objects

AntidoteDB keeps track of the operations performed on the objects in the journal. To create specific versions of them, based on the constrains of the query, the objects are materialised. Materialization starts with the base copy of the CRDT type requested and then recursively applies the operations from the journal until a compatible object version is created, this version is then returned to the client.

Materialisation is triggered by a read operation. For any given key (identifier for the object) and type (CRDT type), the materializer reads the journal and filters out the log records that correspond to the requested key. This list of operations is called the *payload*. The payload contains both, committed and uncommitted operations. The object is constructed by applying operations based on their order in the journal only if the following two criteria are satisfied:

1. The operation was performed under a transaction that committed and is a causal predecessor of the transaction that issued the operation. This satisfies the *Read Committed* isolation level (see Definition 1).

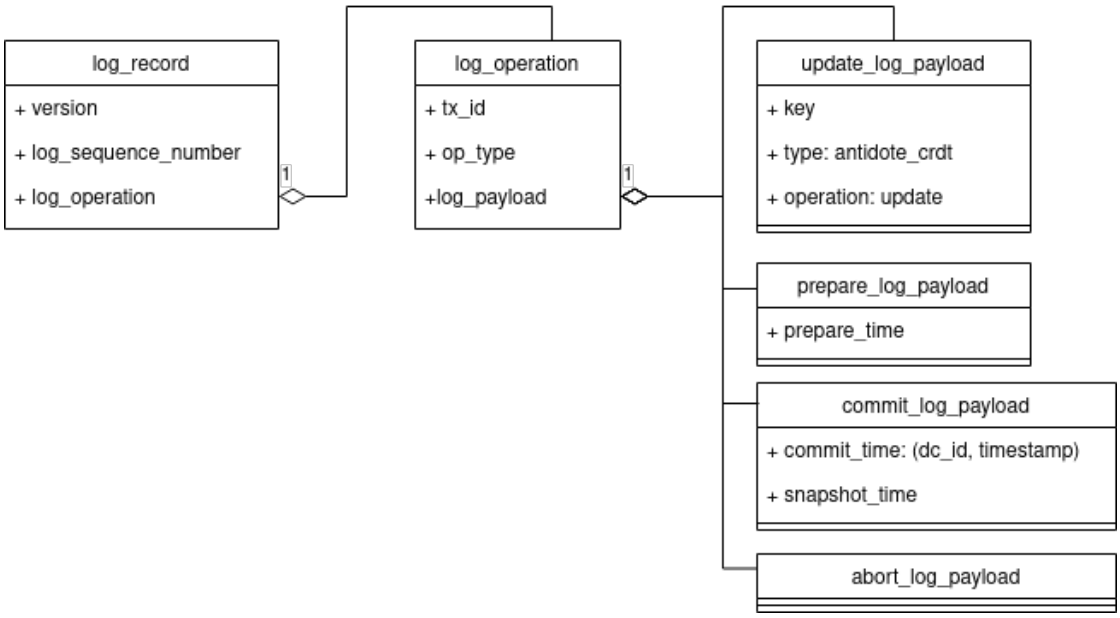


**Figure 2.1.:** *Structure of a single log record in AntidoteDB's Journal.*

2. The operation was performed within a transaction that has issued the read. This satisfies *Read Own Writes*.

Any operation satisfying these criteria is put into an ordered list called *committed payload*. The order of operations in this list is the order in which these operations were committed. The operations issued in a single transaction maintain their execution order.

After filtering the committed and uncommitted operations, the materializer creates two different objects:

- Interactive Materialisation: The snapshot containing committed and the transaction's own operations.
- Stable Materialisation: The snapshot containing only the committed operations.

These are used for different purposes and are explained in the later sections.

## 2.3. Making the Journal navigation faster - AntidoteDB Journal Index

In a high operation volume, distributed database like Antidote, the size of the journal is significant and increases very quickly. The journal is also the primary source of the object state for any given timestamp. However, the brute-force method of reading the journal and filtering out relevant records is highly inefficient.

To mitigate this problem, AntidoteDB maintains a journal index. This index contains pointers to strategic positions within the journal which help in quickly navigating through the journal. The entries in the index consist of the following data:

- Key: the identifier of the object being indexed.
- Timestamp: the snapshot timestamp of the position in the journal.
- Continuation: an object containing the pointer to a location in the journal.

Continuations are logical identifiers that point to chunks in the actual physical journal file. These are managed by the Erlang run-time and are a necessary parameter for the read operation on the journal file through the `disk_log` [4] module.

The index is updated in the following situations:

- Zero Version Index: When a new record is appended to the journal, we check if the object key in the record already has an entry in the index. If this key is being observed for the first time, an entry is added to the index, pointing to the current chunk where this entry is appended, along with a base vector clock. This entry serves an important purpose during the truncation of the journal (see Section 2.7)
- Stable Materialisation Index: Upon a successful read the pointers to the commits as well as their timestamps are added to the index. This helps with the partial materialisation of objects from the cache by starting a journal access at the timestamp of the last commit applied to the cached version and then sequentially reading the records.

## 2.4. Making Reads Faster - AntidoteDB Cache

Materialization is an expensive operation. It involves reading the entire log from an index position, filtering out specific operations and applying them to the base snapshot for a CRDT type. Even for a low volume load on the database, this leads to a significant slowdown as the reads to the journal file bottleneck at the operating system level. The solution is to keep the materialised objects for some time in a temporary storage and using these temporary materializations to reply to queries.

A cached version of the object contains the following parameters:

- Key: The identifier of the object.
- Type: CRDT type.
- Timestamp: The timestamp of the last commit applied to this materialisation.
- Materialisation: The stable materialisation created by the materializer (see section 2.2).

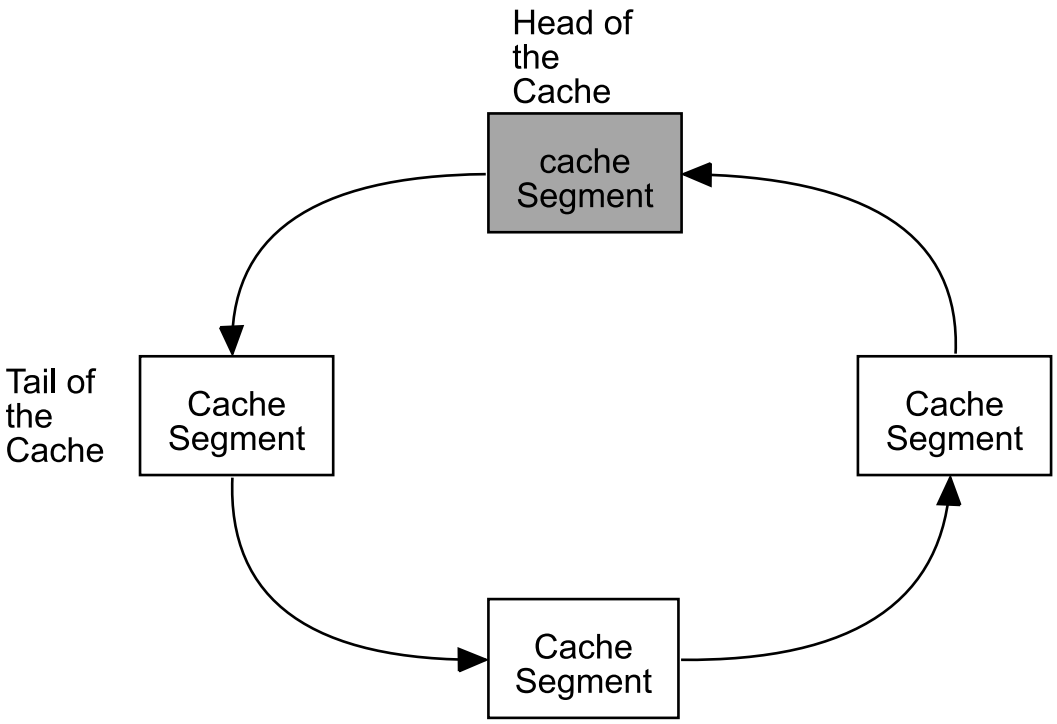


**Figure 2.2.:** *Example Structure of an AntidoteDB Cache with Four Segments.*

### 2.4.1. Structure of the Cache

The AntidoteDB cache is designed as a configurable multilevel segmented cache. The number of levels in the cache as well as the size of each level, (defined in the number of objects) is configurable at start. A level in the AntidoteDB cache is an *Erlang Term Storage* (ETS) table. This allows nested objects to be stored and looked up based on their key. The levels are organised in a circular list with the head of the list pointing to the latest garbage collected level. Recently read or materialised objects are always inserted in the segment at the head which ensures that recently read objects are not removed due to garbage collection.

### 2.4.2. Garbage Collecting the Cache

The design of the cache allows for an implementation of the *Segmented Least Recently Used* [18] garbage collection policy.

As the segment at the head of the cache reaches capacity, the segment at the tail is emptied, thereby replacing the least recently used objects. The head is then modified to point to the empty segment and the tail points to the next oldest segment.

Segmented LRU is not the most conservative of cache replacement policies. But the

design of the cache in Erlang based on ETS tables [5], Segmented LRU offers a balanced approach. Deleting individual elements from an ETS table to invalidate speciofic objects from the cache is more expensive since a selective delete requires a lookup operation. On the contrary, deleting the entire table is less expensive when compared to the cost of deleting all the elements in the table.

## 2.5. Persisting the Cache - AntidoteDB Checkpoint

Objects once written to the cache are vulnerable to garbage collection and system failures. For a well-configured system, the objects in cache are enough to facilitate a fast read throughput. Even routine maintenance, which might involve down-times, is enough to invalidate the cache which then needs to be rebuilt. Replaying the entire journal is an option to do this but it is vastly expensive and might leave the database unusable for the duration of the rebuild. Since the journal contains all the operations and with a routine use, the journal only grows, there will eventually be a point when the journal becomes too big to fit in the persistent storage, let alone the memory.

This necessitates the need of a persistent store which contains materialised versions of the objects that can be backed up and read on demand, based on the query requirements. This persistent store is the *checkpoint*.

The checkpoint contains the following information about the object:

- Key: Object identifier
- Snapshot Timestamp: The timestamp of the stable materialization of the object.
- Snapshot: Object materialisation at the specified timestamp.

### 2.5.1. Maintaining Checkpoints

The checkpoint store is notified when objects are updated. Whenever an update is issued, the checkpoint daemon adds the transaction that issued update to a set called the transaction set. This set is then used to find the truncation point of the journal based on a common causal predecessor of the currently running transactions. When and how the check-pointing happens is described in section 2.7.

## 2.6. Application Architecture

Riak Core Lite is used as a framework for designing the architecture of AntidoteDB. It is described as a framework that facilitates Dynamo-style [15] architectures for applications like Key-Value stores and messaging systems. One of the main advantages of the Dynamo-style design is that it allows for a leaderless architecture [29]. All the nodes in the system are homogeneous leaders with no single point of failure. This is achieved with a few strategic design choices.

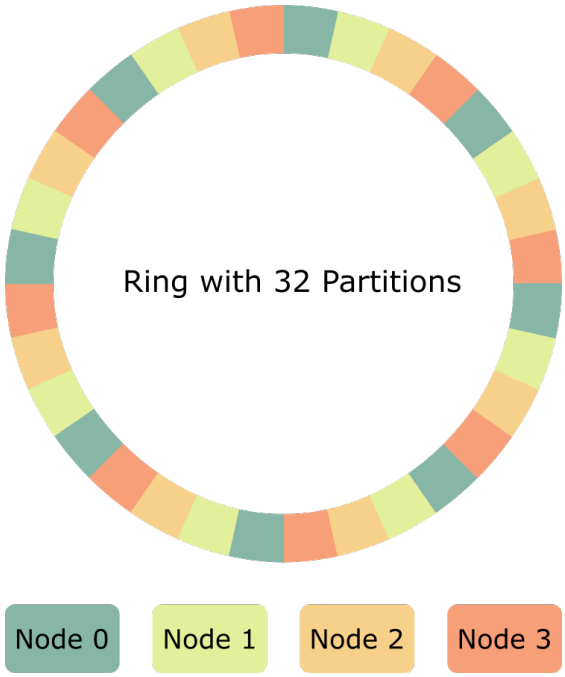


**Figure 2.3.:** *Structure of the Riak Core Ring*

A Riak ring structure with 32 partitions and 4 vnodes. Each node handles 8 segments which have a balanced position in the ring.

### 2.6.1. Riak Ring

The ring is a set of partitions which are addressed by a consistent hash space. This ring is used to manage the state of nodes and is shared between them to synchronise the entire cluster via the *gossip protocol* [35]. This way, each node knows the state of the cluster and failures of a single node are made visible to other vnodes. Failures trigger a hand-off [36] which transfers the ownership of the hash-set of the failed vnode to another vnode. The ownership is returned when the failed vnode comes back online.

The shared ring also allows nodes to send requests to the partition that manages a specific key and, themselves, act as a proxy for the client.

If a node is down or a new node enters the cluster, the nodes synchronise via the ring and the hash addressing is updated to include the modified number of nodes. Figure 2.3 shows the structure of the ring for a 4 node system.

### 2.6.2. Nodes and Virtual Nodes in a Riak Cluster

A *node* is a running instance of an application based on riak e.g. AntidoteDB. It can be a process within a system or a complete system deployed in a single location or even distributed across physical boundaries. Each node in this application consists of multiple *virtual nodes*, also called vnodes. These vnodes correspond to the ring architecture

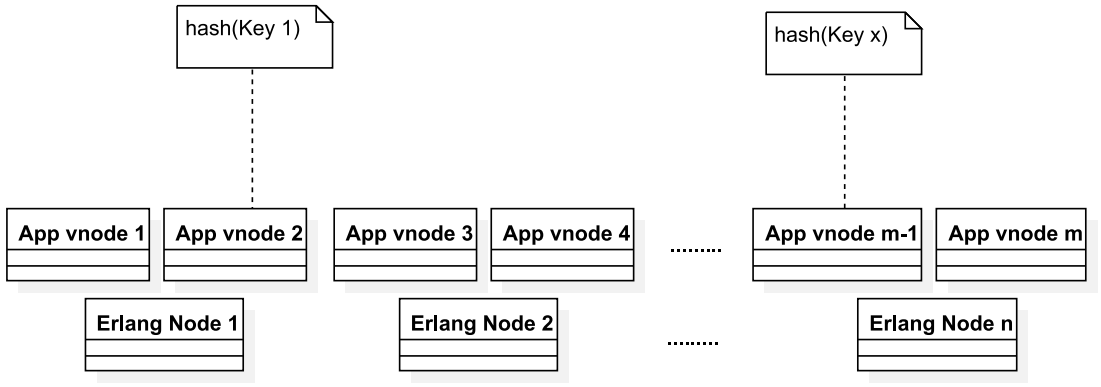


**Figure 2.4.:** *High level architecture of an application based on Riak Core*

This configuration has $n$ nodes and $m$ vnodes are distributed on these $n$ nodes.

defined in section 2.6.1.

A distributed application typically consists of multiple nodes and, consequently due to the design of riak, multiple vnodes too. Each deployment can be configured to manage a different number of vnodes. This number is dependent on the size of the ring.

For example, if we are running a cluster with 3 nodes and the ring size is 128, each node is responsible for 128/3, i.e. 43 partitions with the last node managing 42 partitions.

The decision on how many nodes, vnodes the system needs and can support is reached after cluster planning [34].

Vnodes essentially manage a subset of the cluster's hash space. Although nodes are running system processes, it is the vnodes that perform the actual operations in the application. The high-level architecture of an application based on riak core framework is shown in Figure 2.4.

The architecture of AntidoteDB is explained in detail in Chapter 3. The components of the system are shown in Figure 2.5. In AntidoteDB, each vnode has a journal and it's own cache. This way, the actual journal is split into however many vnodes the system is configured to use. This prevents contention when vnodes try to access the journal. The objects needed by a vnode are stored in a local cache which makes local reads faster.

## 2.7. Limiting the size of the Journal - Truncation

As described in Section 2.5 the journal grows with time as more operations are performed on the objects. With the checkpoint store containing materialised versions of objects at certain timestamps, we can begin to truncate the journal and remove old records from it.

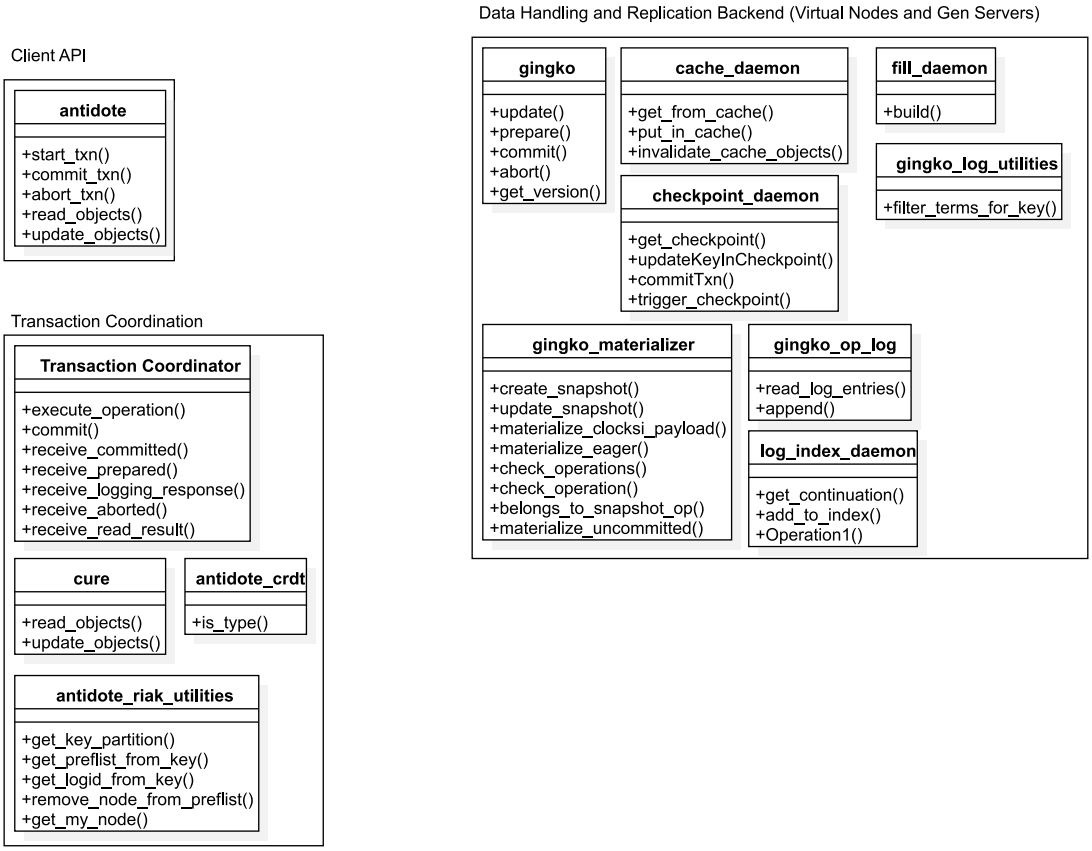


**Figure 2.5.:** *Application components of a single data center in AntidoteDB*

### 2.7.1. Recording operations for truncation

The checkpoint store consists of two ordered sets. They are called *truncation safe set* and *transaction set*. During routine operation, as updates are executed, the running transactions are added to the transaction set of a specific partition based on the key. This transaction record contains a pointer to the last commit that this transaction saw when it was started. This is the causal predecessor.

When a transaction commits all the checkpoint daemons are involved in the 2-PC process and add the committing transaction's commit time to the truncation set.

Based on a configured criteria, when the checkpoint is triggered, the transactions from the truncation safe set are removed from the transaction set. If the first element of the ordered transaction set changes, it indicates that the oldest transaction that started and was running in parallel with others has committed. The oldest transaction was also the top level causal dependency in the dependency graph containing the concurrent transactions.

The new head of the transaction set marks the point in the log until which, the operations can be check-pointed and the journal truncated without violating causal consistency or affecting the operation of the database. This is explained in more detail in Section 2.7.3

### 2.7.2. Guaranteeing a safe Truncation

Truncating the journal is a destructive process and without the correct semantics of truncation, it is possible to break the materialisation process, inducing causality issues at best and leading to complete failure due to corrupted state at the worst. To prevent this, there are invariants that need to be maintained during routine operation of the database. The invariants have been identified in the formal research on transactional and causally consistent databases [30]. They include certain clock times of interest which are:

- Low Watermark: The minimum timestamp that can be queried. This is the timestamp that marks the upper bound of the objects that should be persisted in the checkpoint store.
- Checkpoint Time: This is the minimum timestamp, such that, the updates issued by any transaction committed before this time have been check-pointed.
- DC-Wide Causal Safe Point (DCSf): This is the maximum timestamp until which all the updates, as well as their causal predecessors, have been synchronised across all the shards in a DC.
- Min Dependency: This is the oldest snapshot timestamp which is required by a running transaction as a causal dependency.
- Max Committed: This is the latest commit timestamp written to the journal.

Based on these timestamps, the invariants that AntidoteDB needs to maintain are the following:

- $LowWatermark \leq CheckpointTime$: This invariant ensures that the operations that have not been checkpointed should not be truncated from the journal. It is held by the mechanism of maintaining running transactions within the checkpoint daemon.
- $CheckpointTime \leq MinDependency$: This invariant ensures that the operations written to the journal by the earliest started and currently running transaction are not truncated by the transactions that started after and committed before it. The Min Dependency is maintained as the first element of the ordered *transaction set* and this element changes only when the transaction at this entry commits, thereby maintaining the invariant.
- $DCSf \leq MaxCommitted$: This invariant ensures that any operation that has not been synchronised is not checkpointed. This is held by design of the riak core ring. Since the ring structure is maintained across all the partitions and is modified based on the number of nodes available, the operations are guaranteed to be synchronised if the partition is not faulty.

### 2.7.3. Calculating the Truncation point in the Journal

Finding the latest entry in the journal that is required to maintain operational performance is non trivial. This involves keeping track of the multiple concurrently running transactions which might have invisible dependencies between each other. In general, it is safe to say that the journal can be truncated to a record that corresponds to a commit visible to the currently running transactions. Based on the invariants described in Section 2.7.2, any record in the journal that comes before the *Min Dependency* time is eligible for truncation. An example is shown in Figure 2.6.

When transactions *tx1* and *tx2* start, the tail of the journal is at `start` so nothing can be truncated from the beginning. At operation label 8, *tx2* commits, the operations until position 8 are now visible to *tx3*, but the journal cannot be truncated yet because *t1* still has the lowest visibility timestamp (Min Dependency). When *t1* commits at operation label 10, the lowest visible position becomes 8. At this point in time, operations from `start` to position 8 can be checkpointed and removed from the journal to reduce its size. The new Min dependency at this point is position 8

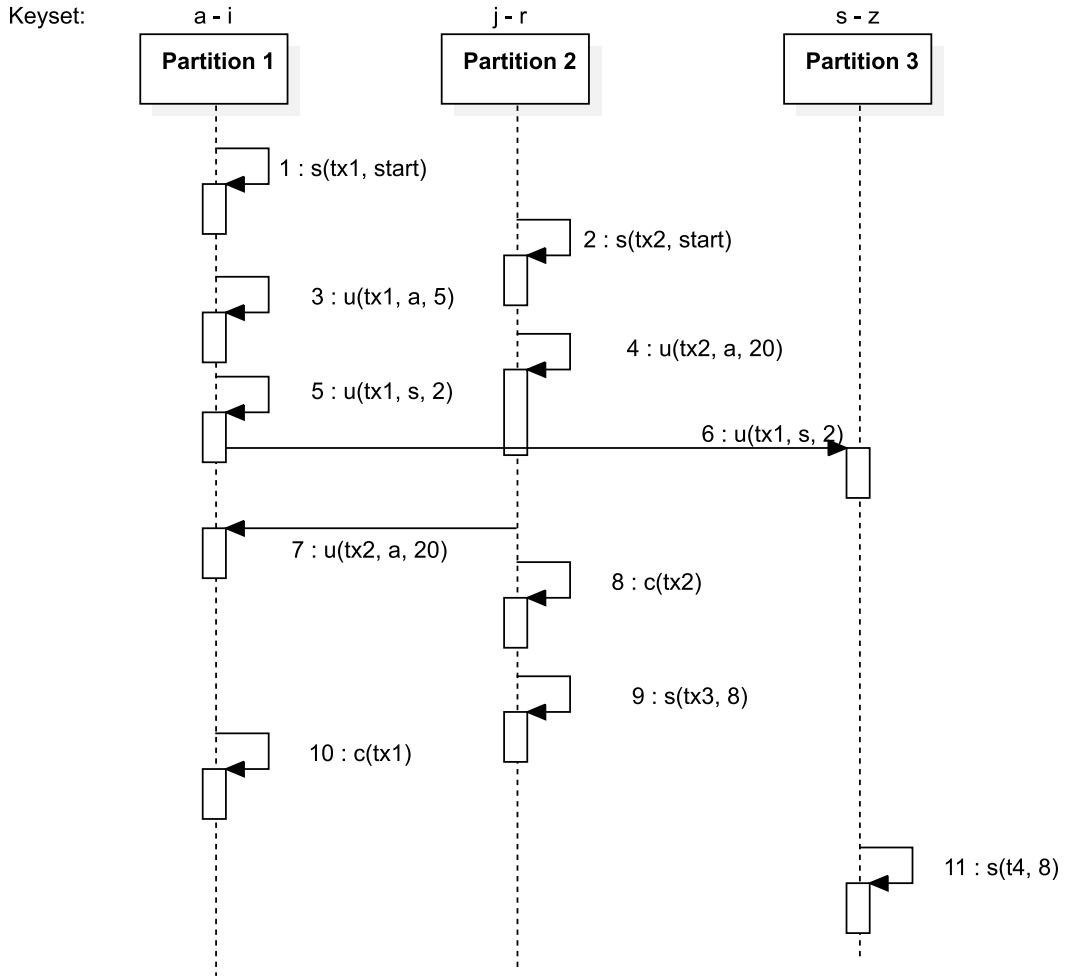


**Figure 2.6.:** *Schematic for finding the truncation point with multiple running transactions and 3 partitions.*

After the records for operation number 11 are appended to the journal, the records for operations 1 through 8 can be truncated since the minimum required journal entry is at position 8, belonging to transaction *t3*.

# 3. Implementation details

AntidoteDB is implemented in Erlang. The interest in Erlang is primarily driven by the capabilities of the language and the virtual machine at providing a very robust and natural concurrency model. It supports a process-based model of concurrency with the primary idea being that everything that runs within the Erlang VM is a process. Such a model, as seen in other languages like Java and Python, would be extremely resource intensive and also slow. Erlang allows for the creation and destruction on processes very seamlessly, without a large overhead and the VM can handle several thousand light weight processes running simultaneously. All this, combined with the capabilities of a soft-realtime system which allows for very low response times makes Erlang an excellent candidate for implementing distributed applications like AntidoteDB.

This chapter provides a detailed overview of the system components and their implementation details based on the design described in Chapter 2. the internal calls as well as the communication between components is explained through activity and sequence diagrams. Along with this, the supporting abstractions provided by the Erlang run-time are also explained when necessary.

## 3.1. System Implementation and Components

AntidoteDB, being a distributed application that is integrated into other use cases, remains a candidate for a client server model. With the ability to deploy individual components of the system on physically different locations, the internal communication also follows the same client server model where processes communicate with each other through messages and do not share a state.

### 3.1.1. Generic Server Processes

Implementation of distributed applications based on this model is made easier with a *Generic Server Process* [1] also called a `gen_server` in Erlang terminology. Generic servers have standard callbacks for handling synchronous and asynchronous messages that are relayed by the Erlang VM. These callbacks contain mechanisms for manipulating the state of the process as well as sending and receiving messages to other components. The key components in AntidoteDB implemented as a `gen_server` are:

- Cache Manager (`cache_daemon`),
- Journal Handler (`gingko_op_log`),
- Journal Index Manager (`log_index_daemon`) and

- Checkpoint Store Manager (`checkpoint_daemon`).

### 3.1.2. Generic State Machines

State machines are extremely useful when performing tasks such as transaction management. Erlang provides the state machine behaviour as the `gen_statem` abstraction. State machine behaviour allows processes to wait for a specific input in a particular state and then transition to another state based on that input. `gen_statem` eases the management of a transaction's phases as it starts, performs updates and also supports the implementation of a 2-phase commit protocol [22]. Because Erlang VM supports several concurrent processes, multiple transaction coordinators can be spawned to manage concurrent transactions.

## 3.2. High Level System Architecture

### 3.2.1. Erlang OTP and Supervision trees

One of the most basic application design paradigms in Erlang are supervision trees. A supervision tree categorises processes into two types:

- Supervisors: Processes that monitor other processes in the system. Supervisors can restart one or more processes based on a failure criteria. A supervisor can have multiple worker processes originating from the same or different process definitions.
- Workers: Processes that perform the actual work. Workers can start other supervisors or spawn processes.

Supervisors and Workers are defined as behaviors in Erlang code [2]. Supervisors and workers are organised in a tree structure which is called a *Supervision Tree*. The supervision tree for AntidoteDB components is shown in Figure 3.1

### 3.2.2. Partitioning the Data and Routing Calls

A riak cluster, as described in section 2.6.2, is made up of several virtual nodes that run on physical nodes. Based on the ring size, a number of partitions are created, each managed by a vnode. The keys for objects within AntidoteDB are, thus, bound to a specific partition and a vnode handles all the operations for that key. This partitioning is decided at the start of the system when the vnodes are initialised and can only change if the cluster is reconfigured.

This partitioning scheme requires calls to be routed based on the key. The updates for a key can be issued through any vnode. If a vnode is not responsible for handling the key it receives in a call, it becomes a proxy and the call is forwarded to the vnode responsible for that key. The proxy vnode can meanwhile continue handling other calls.

Call forwarding is made available through the `command`, `sync_command` and `sync_spawn_command` methods.

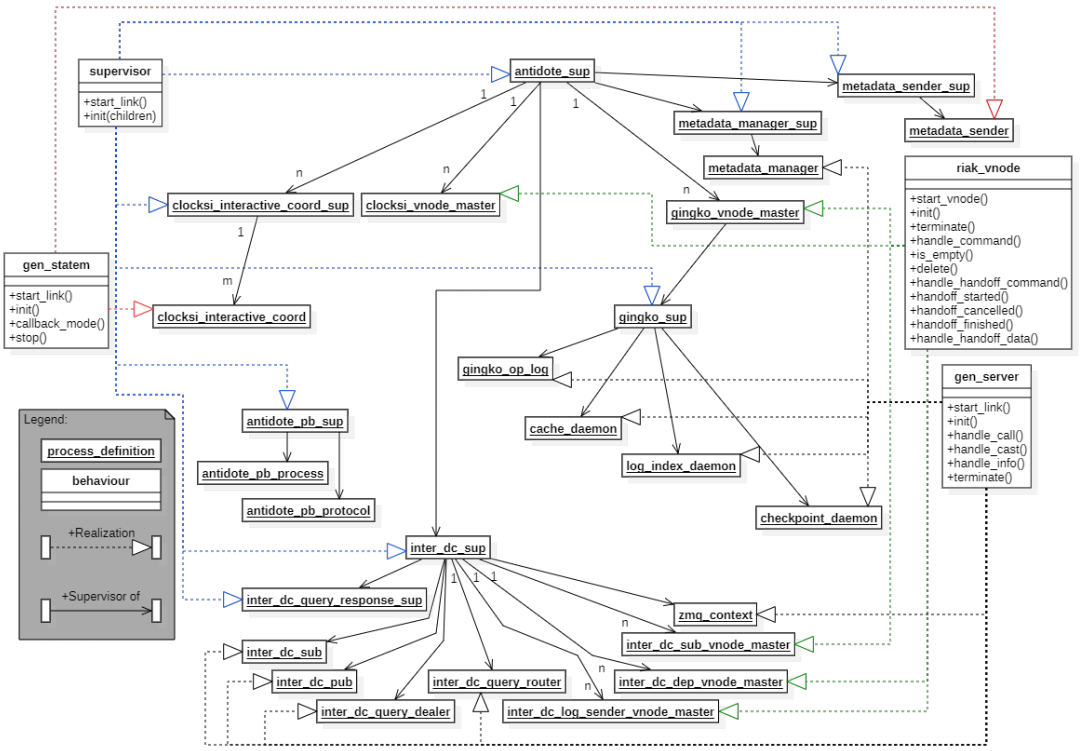


**Figure 3.1.:** *Erlang/OTP Supervision Tree for AntidoteDB*
The architecture shown in this Figure includes components which are involved with the InterDC communication and global state management as well as messaging and metadata management but these components are out of scope for this work. They are included to present a complete a complete system design.

### 3.2.3. Internal Operations within AntidoteDB libraries

Before discussing the process of reading and writing objects within AntidoteDB, there are several internal operations which help in the process and are used repeatedly. These operations are executed within the supporting libraries of AntidoteDB and can affect multiple components. The operations are:

- `get_version(Key, CRDTType, TransactionId, MaxReadSnapshotTime, MinReadSnapshotTime)`: The `get_version` call is the internal call sent to the materialization and logging library (Gingko) when an object needs to be read. This call is responsible for finding a version of the object which is compatible with the query parameters, mainly the read clock. To do this, it checks if the object is in the cache. If this leads to a miss, the closest version from the checkpoint store is retrieved. If the checkpointed version satisfies our clock, the object is returned. Otherwise, the object is partially or completely built based on the clock of the checkpointed version. This process is shown in Figure 3.2 in detail.

- `build(TransactionId, Key, Type, BaseSnapshot, MinSnapshotTime, MaxSnapshotTime)`: Once the correct build parameters for an object version are identified, a build call is initiated. This call fetches the operations from the journal (see section 2.2) and applies them onto the base version of the CRDT or updates a snapshot by applying operations on it. The build process also indexes the journal by adding commit points to the journal index. This index helps future reads navigate the journal quickly. The detailed sequence of the build process is shown in Figure 3.3.

- `append(log_record)`: This call is used to create an entry in the journal. The record is appended to the tail of the journal. To make reads faster and preventing unnecessary reads, a zero version index is created when appending to the journal. If the key for an operation in the journal record does not exist in the index, a zero entry with an empty vector clock is added. This zero index will be the starting point of a read operation when this key is queried to build a new snapshot from the journal. The sequence of calls for appending an entry is shown in Figure 3.4.

## 3.3. Reading and Writing Objects in AntidoteDB

The main operations exposed by the AntidoteDB client API are the following:

- `read_objects(Objects, TxId)` : This read is executed in the context of an existing transaction.

- `read_objects(Clock, Properties, Objects)` : This read can be issued without explicitly starting a transaction. A transaction is started if the read operation refers to multiple objects. This is to ensure that the read values are compatible

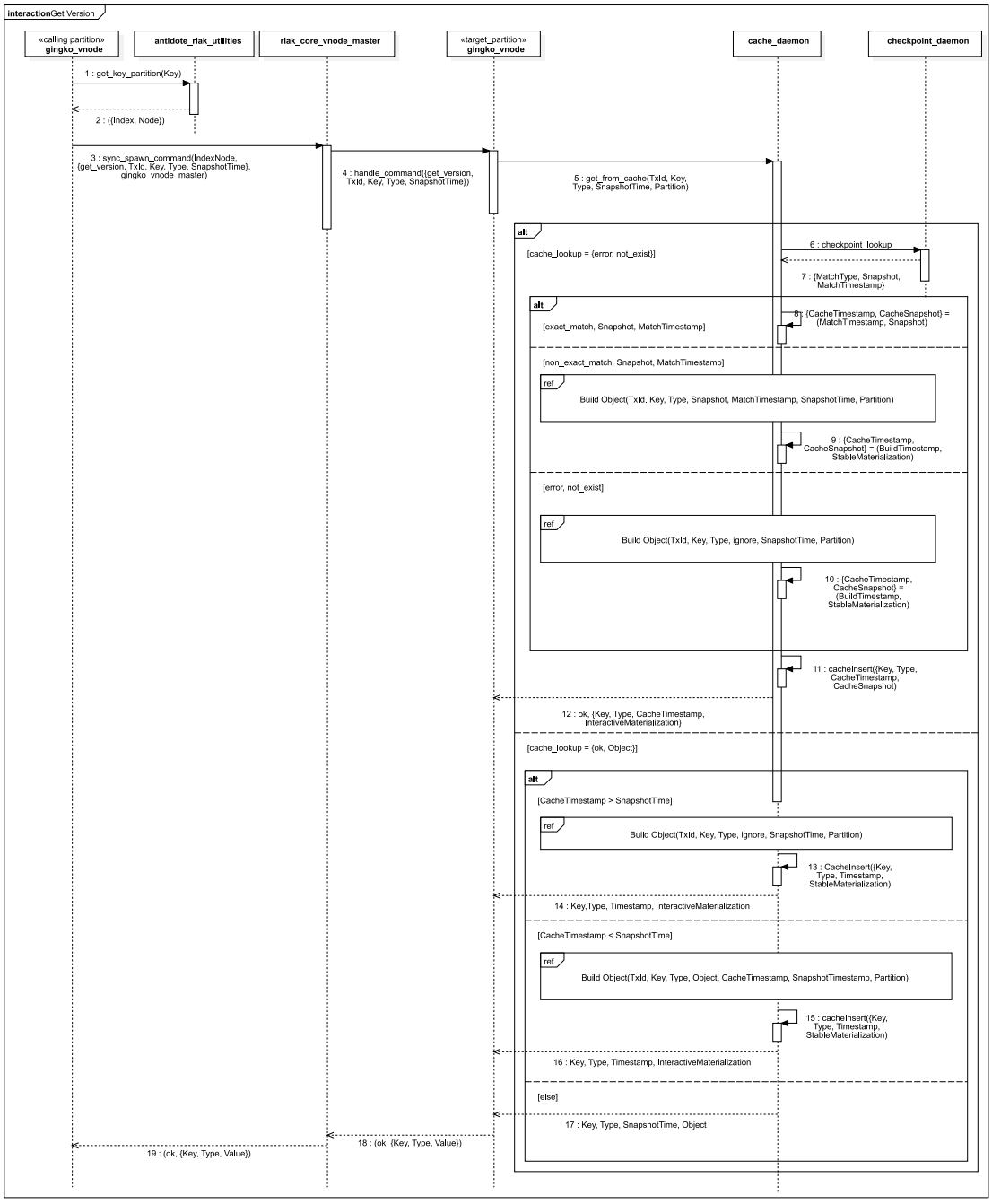


**Figure 3.2.:** *Sequence diagram for a* `get_version` *call within Gingko*

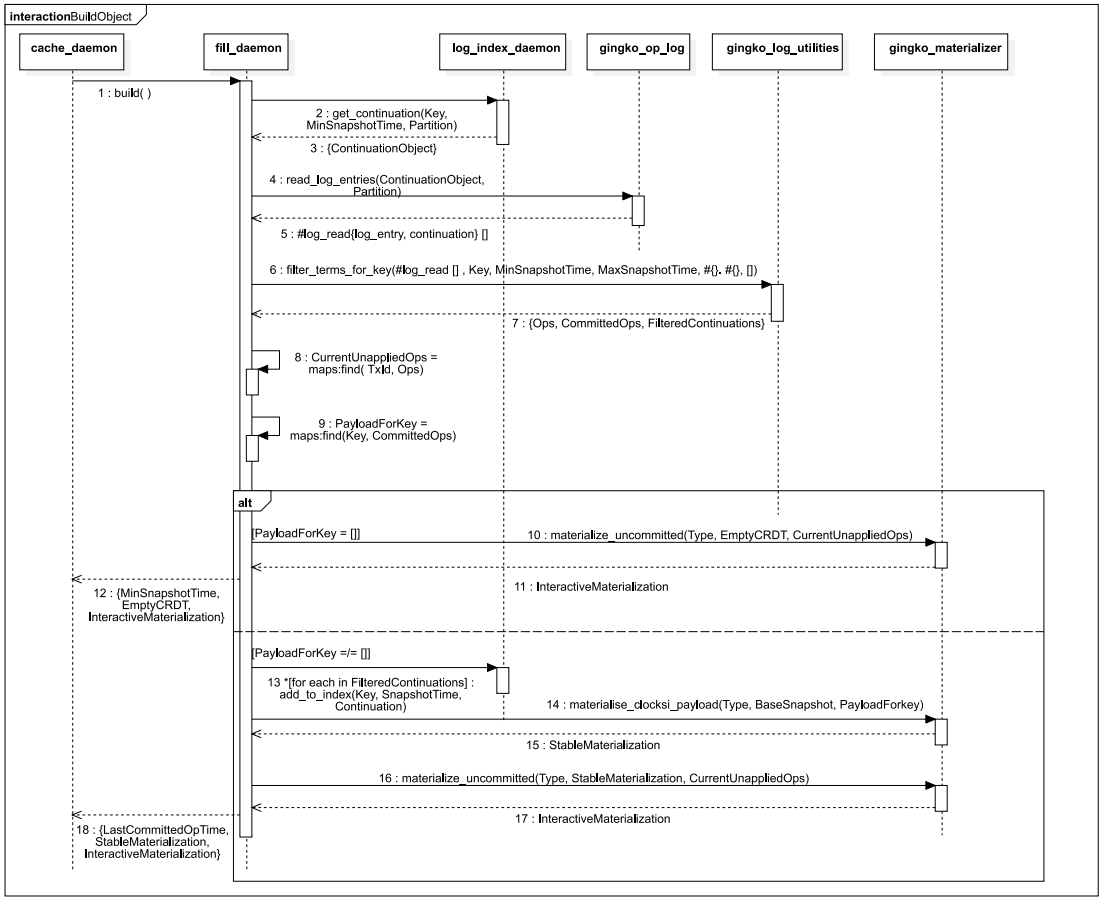


**Figure 3.3.:** *Sequence diagram for a* `build` *call within Gingko*

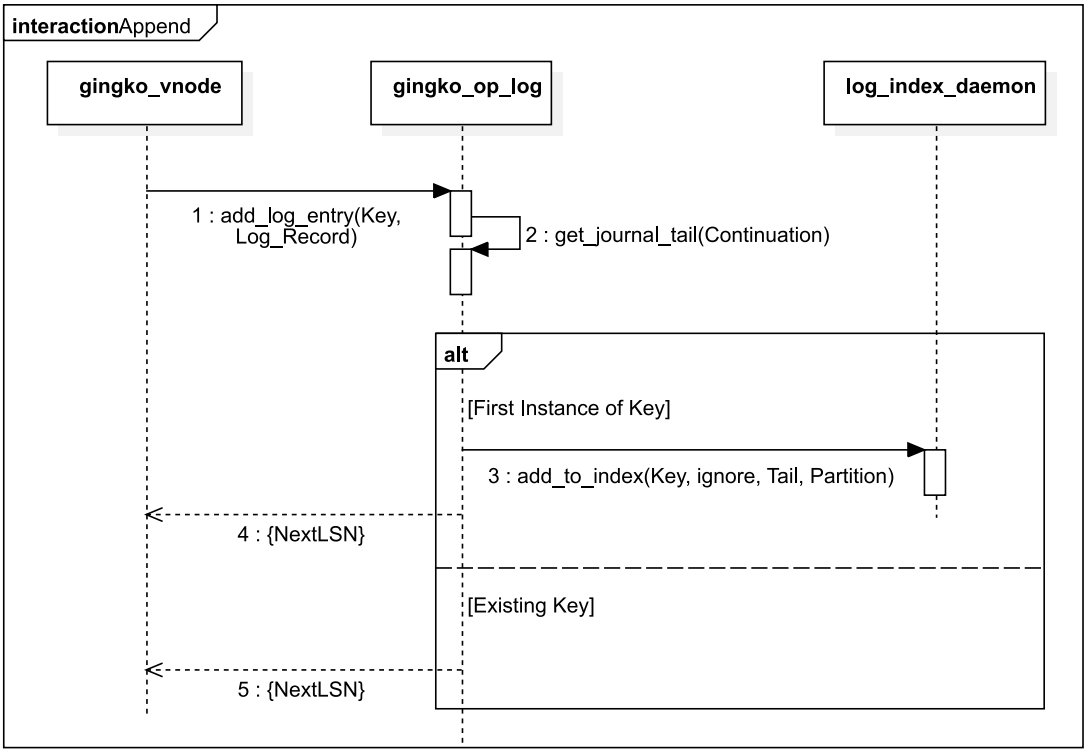


**Figure 3.4.:** *Sequence diagram for an* `append` *call within Gingko*

with the same clock and values concurrently updated by other transactions are not read.

- `update_objects(Updates, TxId)` : This update is executed in the context of an existing transaction. This operation is isolated and invisible to other transactions until the transaction is committed.
- `update_objects(Clock, Properties, Objects)` : This update can be issued without explicitly starting a transaction. For this operation, a transaction is started and the updates are internally performed within the context of a transaction.
- `start_transaction(Clock, Properties)` : This operation starts a transaction. A corresponding transaction coordinator state machine is started and waits for operations to be issued. A transaction ID id returned by this operations used to identify the transaction coordinator.
- `abort_transaction(TxId)` : An already running transaction is aborted using this operation.
- `commit_transaction(TxId)` : An already running transaction is committed by this operation. Once a commit is issued, the transaction coordinator initiates the two phase commit protocol [22].

### 3.3.1. Essential Notation

The following sections explain the execution details of the different possible combinations of reads and writes. For this, there are certain notation details that need to be considered.

- $o(k, op, v) \in t$ is an operation for key $k$ of type $op$ with effect $v$ issued within a transaction $t$.
- $\tau$ is a set of transactions running in the system.
- $\zeta$ is a set of committed transactions.
- $\kappa$ is a set of keys.
- $\nu$ is a set of CRDT values.
- $vc[\alpha, \beta, ...]$ is a vector clock which is used to denote snapshot timestamps, commit timestamps and for the read operations, the snapshots required.
- $vc[\alpha_1, \beta_1, ...] \leq vc[\alpha_2, \beta_2, ...]$ is the strictly greater than comparison means that $\alpha_1 \leq \alpha_2$ and $\beta_1 \leq \beta_2$ and so on.
- $r(\kappa, vc[\alpha, \beta, ...])$ is a read operation without a transaction context provided.
- $r(\kappa, t, vc[\alpha, \beta, ....])$ is a read issued within a transaction.
- $u(\kappa, \nu, vc[\alpha, \beta, ....])$ is an update executed without a transaction context provided.
- $u(\kappa, t, \nu)$ is an update issued within a transaction.

### 3.3.2. Single Object Read (Static)

Reading a single object statically means that the read operations was issued outside a transaction context. This involves reading the last committed value for a specific key. Using the notation, the result of this operation will be a materialisation of the set of operations ordered by their commit timestamps based on the filter described in Equation 3.1

$$r(\kappa, vc_r) = \{\forall o(k, op, v) \in t, \forall t \in \zeta : vc_t \leq vc_r, k \in \kappa, |\kappa| = 1\} \tag{3.1}$$

This set contains all the operations for they key in the set $\kappa$ belonging to the transactions that were committed such that the commit timestamp $vc_t$ of the transaction is less than or equal to the timestamp requested for the read $vc_r$. For a single read, $\kappa$ contains only one key.

The query of this type does not require a transaction coordinator and is answered as shown in Figure 3.5. The call sequence with parameters is shown in detail in the Figures 3.6.

### 3.3.3. Multi Object Read (Static)

Reading multiple objects in a single query can lead to inconsistent reads if a concurrent transaction updates the values. To prevent this, static multi object read queries start transaction implicitly which is committed when the objects are read. This implicit transaction cannot receive any update queries since the transaction id is never exposed to the client. Unlike single object reads, a multi object read call is distributed and routed to the responsible vnodes. These vnodes are then included in the 2-PC protocol when the transaction commits. The result is the materialization of the set of operations ordered by their commit timestamps based on the filter in Equation 3.2

$$r(\kappa, vc_r) = \{\forall o(k, op, v) \in t, \forall t \in \zeta : vc_t \leq vc_r, k \in \kappa\} \tag{3.2}$$

This set contains all the operations for they key in the set $\kappa$ belonging to the transactions that were committed such that the commit timestamp $vc_t$ of the transaction is less than or equal to the timestamp requested for the read $vc_r$. For a single read, $\kappa$ can contain more than one key.

The query is answered as shown in Figure 3.7. The sequence of calls is shown in Figure 3.8

### 3.3.4. Single/Multi Object Read (Interactive)

An interactive read is issued after a transaction was already started and contains the ID of the transaction. The read semantics for an interactive include, in addition to the committed updates, the updates performed by the transaction itself. Using the notation, the result is the materialization of the set of operations ordered by their commit timestamps based on the filter described in Equation 3.3

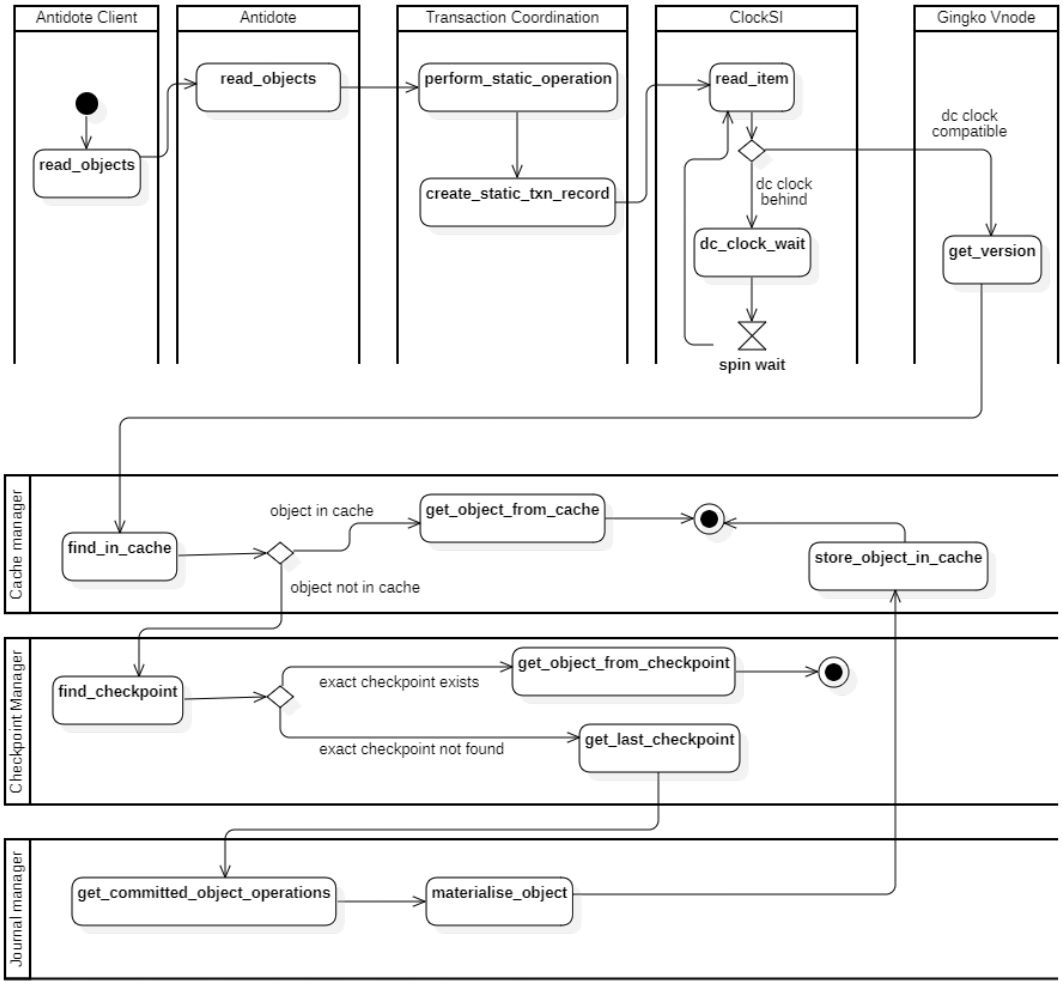


**Figure 3.5.:** *Activity diagram for reading a single object in AntidoteDB*

A single object read is a static operation and is answered without a transaction coordinator. A static transaction record is created which is included with the internal calls in AntidoteDB.

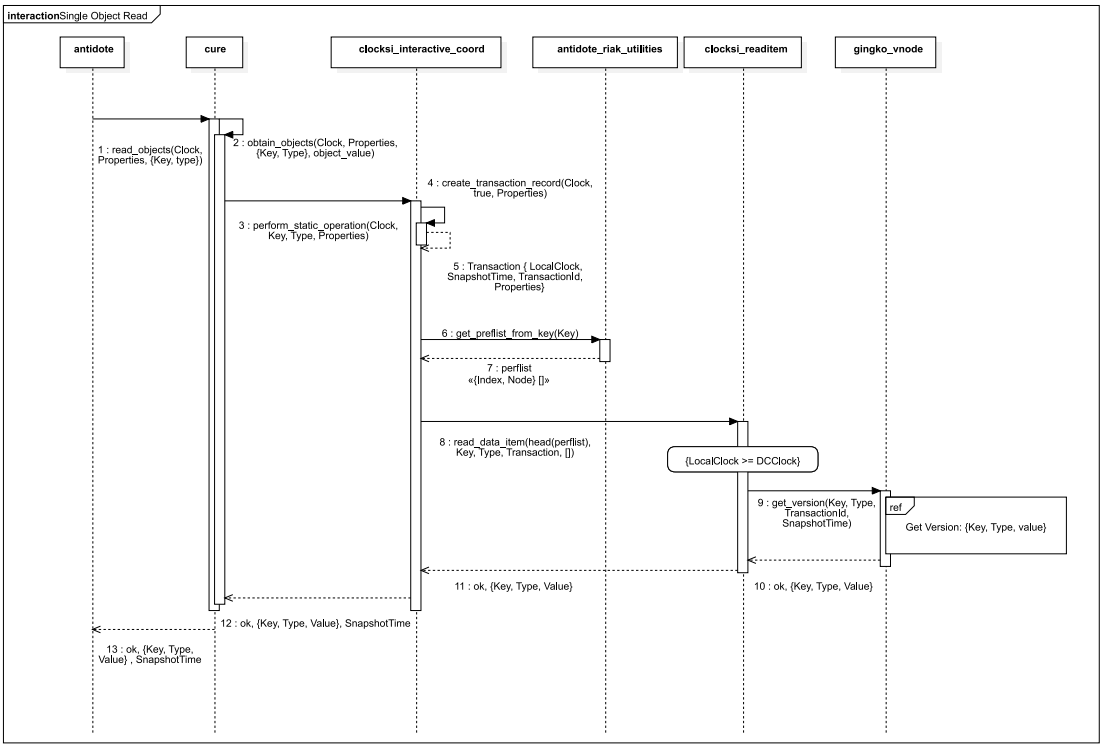


**Figure 3.6.:** *Sequence diagram for a call, reading a single object statically, sent to Gingko*

The call for a single object read triggers a materialisation in Gingko. This is done through the `get_version` call. `get_version` is only called if the DC's clock is at least equal to the required read clock.

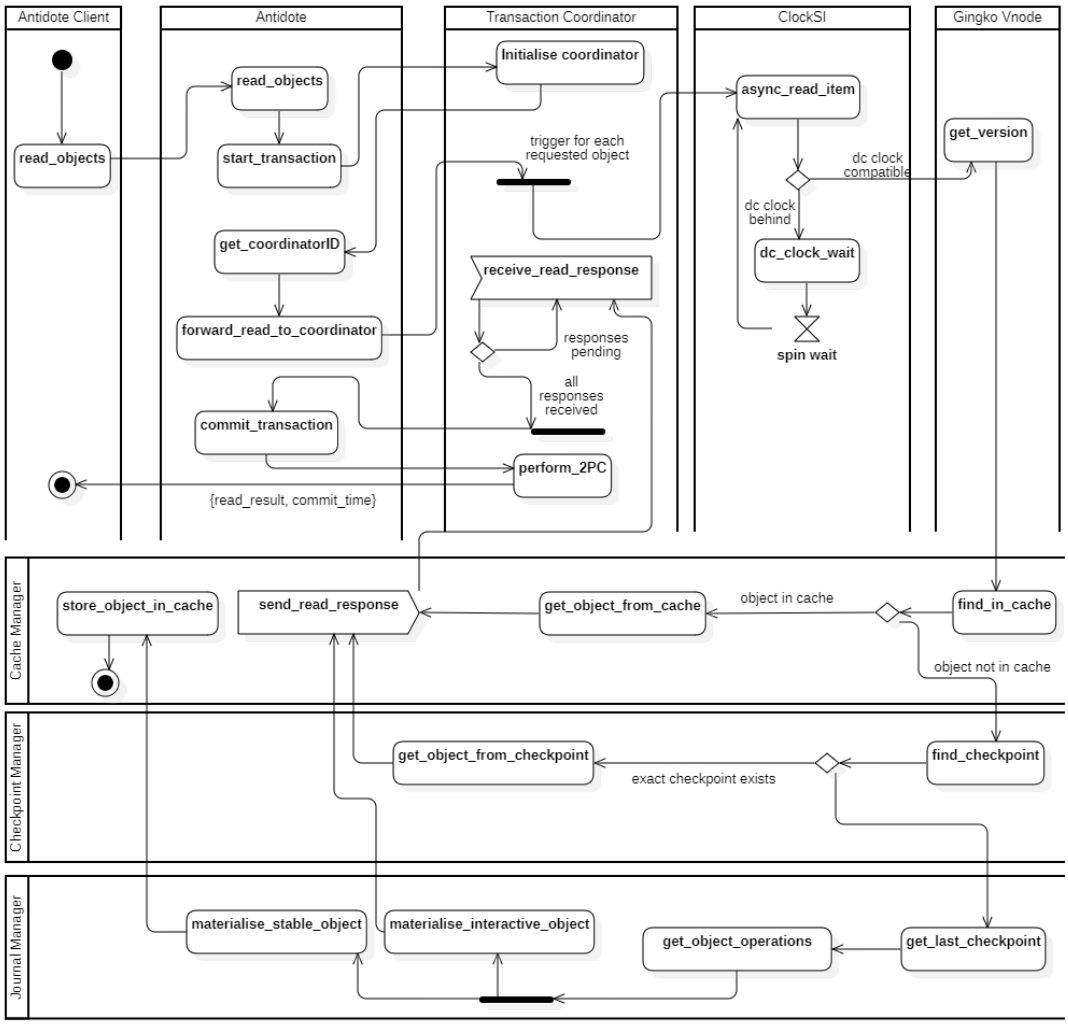


**Figure 3.7.:** *Activity diagram for reading a multiple objects within a single query in AntidoteDB*

In a multi object read, a transaction coordinator is started that manages the read calls and their results. After the read responses are received by the coordinator, 2-PC is initiated.

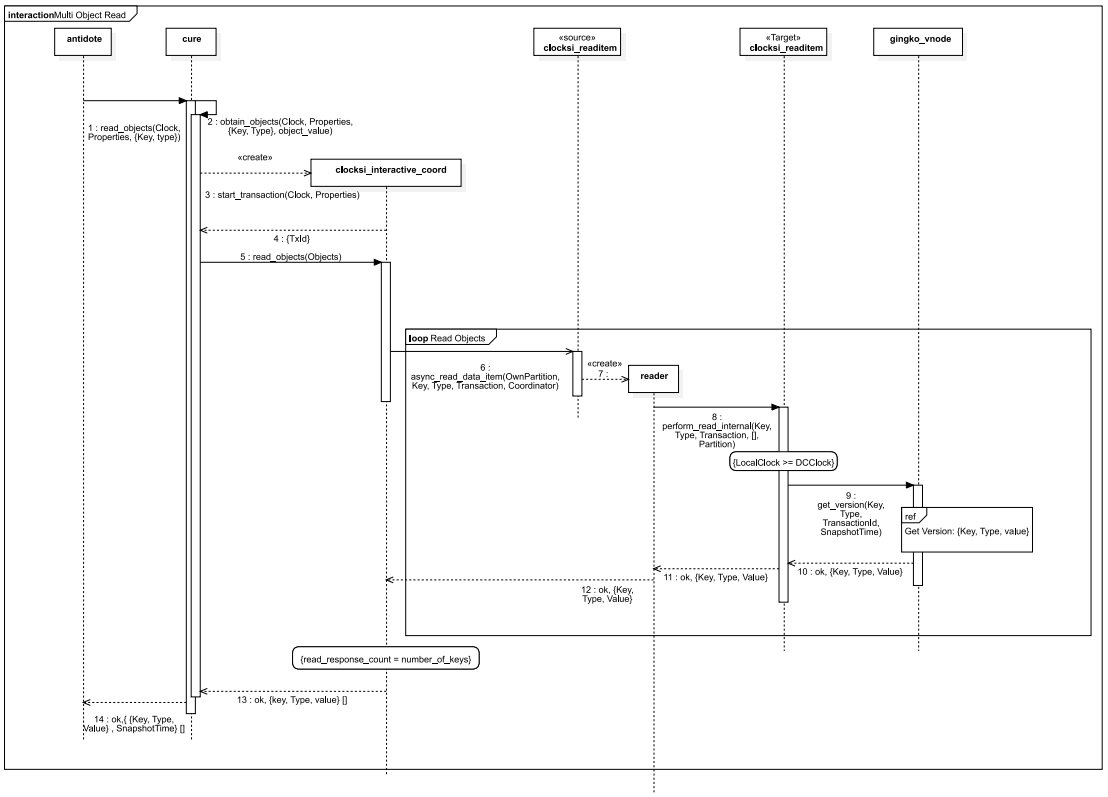


**Figure 3.8.:** *Sequence diagram for a call, reading multiple objects within a single query, sent to Gingko*

Multi object reads spawn processes that perform the reads. The `reader` process is created via tha `async_read_data_item` call. This `reader` send the read result back to the coordinator.

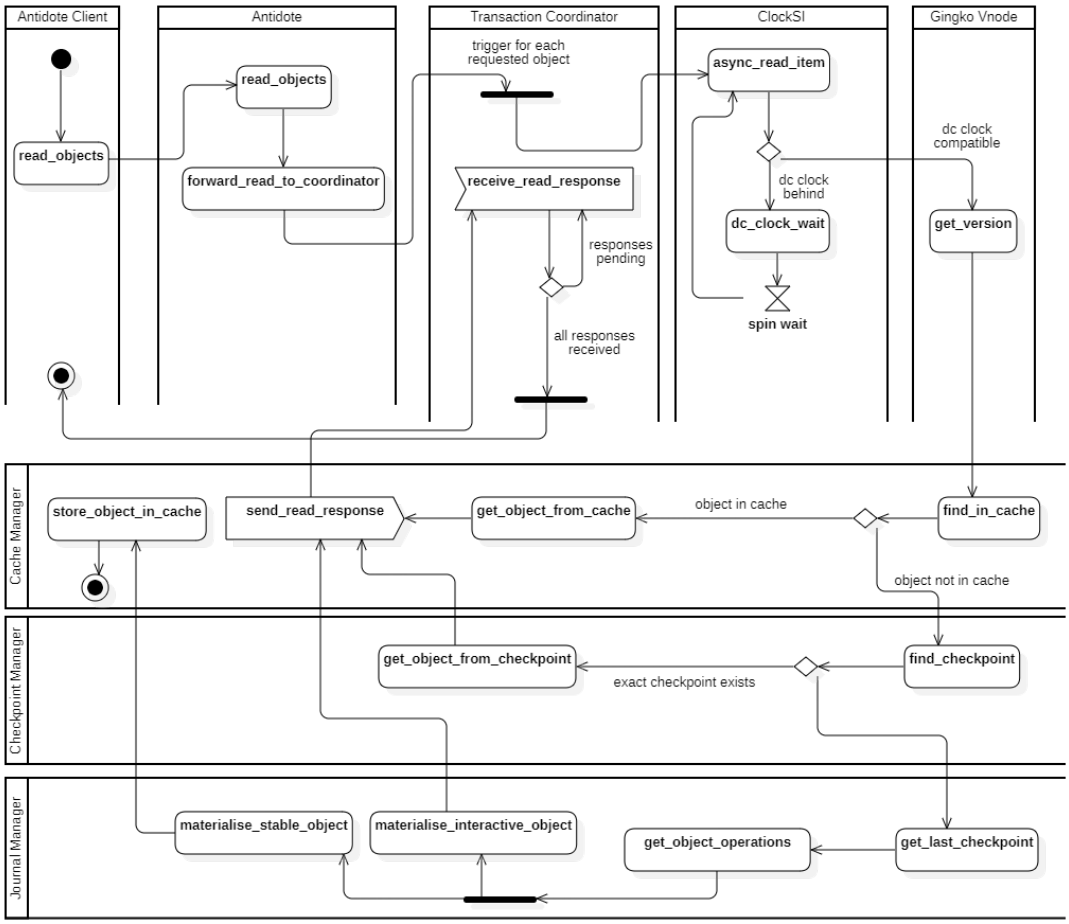


**Figure 3.9.:** *Activity diagram for reading a multiple objects interactively in AntidoteDB*

$$r(\kappa, T) = \{\forall o(k, op, v) \in t, \forall t \in \zeta : k \in \kappa\} \cup \{\forall o(k, op, v) \in T : k \in \kappa\} \tag{3.3}$$

This set contains all the operations belonging to the transactions that have been committed as well as the uncommitted operations issued by the current transaction. This ordered set corresponds to the interactive materialisation of the objects for keys in $\kappa$

The query of this type is handled by a transaction coordinator and is answered as shown in Figure 3.9. The Sequence of calls issued for an interactive read is shown in Figure 3.10

### 3.3.5. Update Objects (Static)

Updating the objects within AntidoteDB involves adding an update operation record in the journal. The updates for a key are sent to the partition responsible for handling that

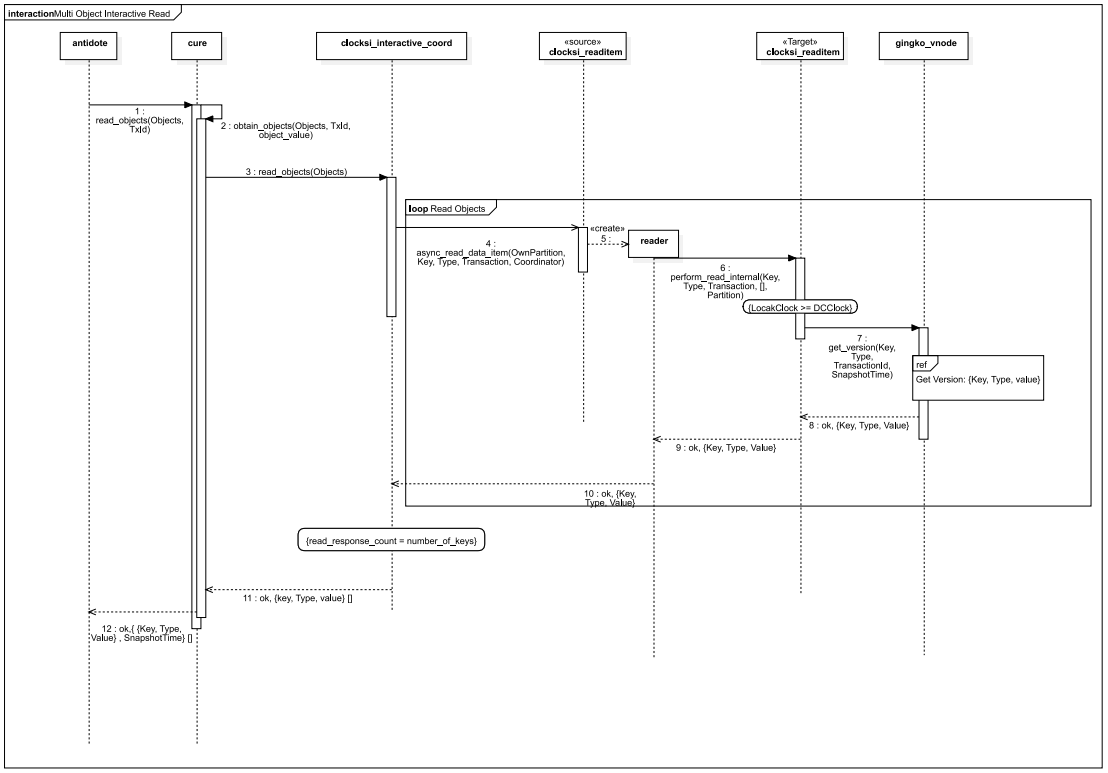


**Figure 3.10.:** *Sequence diagram for a call, reading multiple objects within a single query issued in a transaction, sent to Gingko*

key as described in section 3.2.2. The effect of executing an update is represented by the set described in Equation 3.4.

$$u(\kappa, \nu, vc[\alpha, \beta, ....]) = \{\forall o(k, op, v) \in t, \forall t \in \zeta : vc_t \leq vc_u, k \in \kappa\} \cup \{\forall o(k, op, v) : k \in \kappa, v \in \nu\} \quad (3.4)$$

The flow of operations for an update is shown in Figure 3.11. The call sequence is detailed in 3.12

### 3.3.6. Update Objects (Interactive)

An interactive update is issued within a transaction and adds a record to the journal. For this update, the transaction coordinator handling the transaction is identified through the transaction id. 2 phase commit is performed only when an explicit commit is issued.The effect of executing an update is represented by the set described in Equation 3.5.

$$u(\kappa, t, \nu) = \{\forall o(k, op, v) \in T, \forall T \in \zeta : k \in \kappa\} \cup \{\forall o(k, op, v) \in t : k \in \kappa, v \in \nu\} \quad (3.5)$$

The flow of operations is shown in Figure 3.13. The call sequence is shown in Figure 3.14

## 3.4. Failure scenarios and handling errors during normal operation

Faults, high latency and network partitions are only some of the things that can go wrong and lead to failures within a distributed application. There are certain reliability parameters that are guaranteed by any distributed application and those parameters serve as the basis of how and where to deploy that application. AntidoteDB guarantees TCC which can be violated in several ways without the presence of mitigation measures. A lot of these measures are based on the ability of the Erlang VM to handle several concurrent processes without an overhead. Additional guarantees for maintaining partitions, data replication and vnode failures are implemented through riak. Some common failure scenarios are explained in the following sections.

### 3.4.1. Heavy loads and multiple concurrent transactions leading to unintended aborts

Erlang VM can handle several thousand concurrent processes, so, it is possible to have several thousand concurrent transaction coordinators running. If due to an unexpected fault, a transaction coordinator fails, the transaction id becomes invalid because the

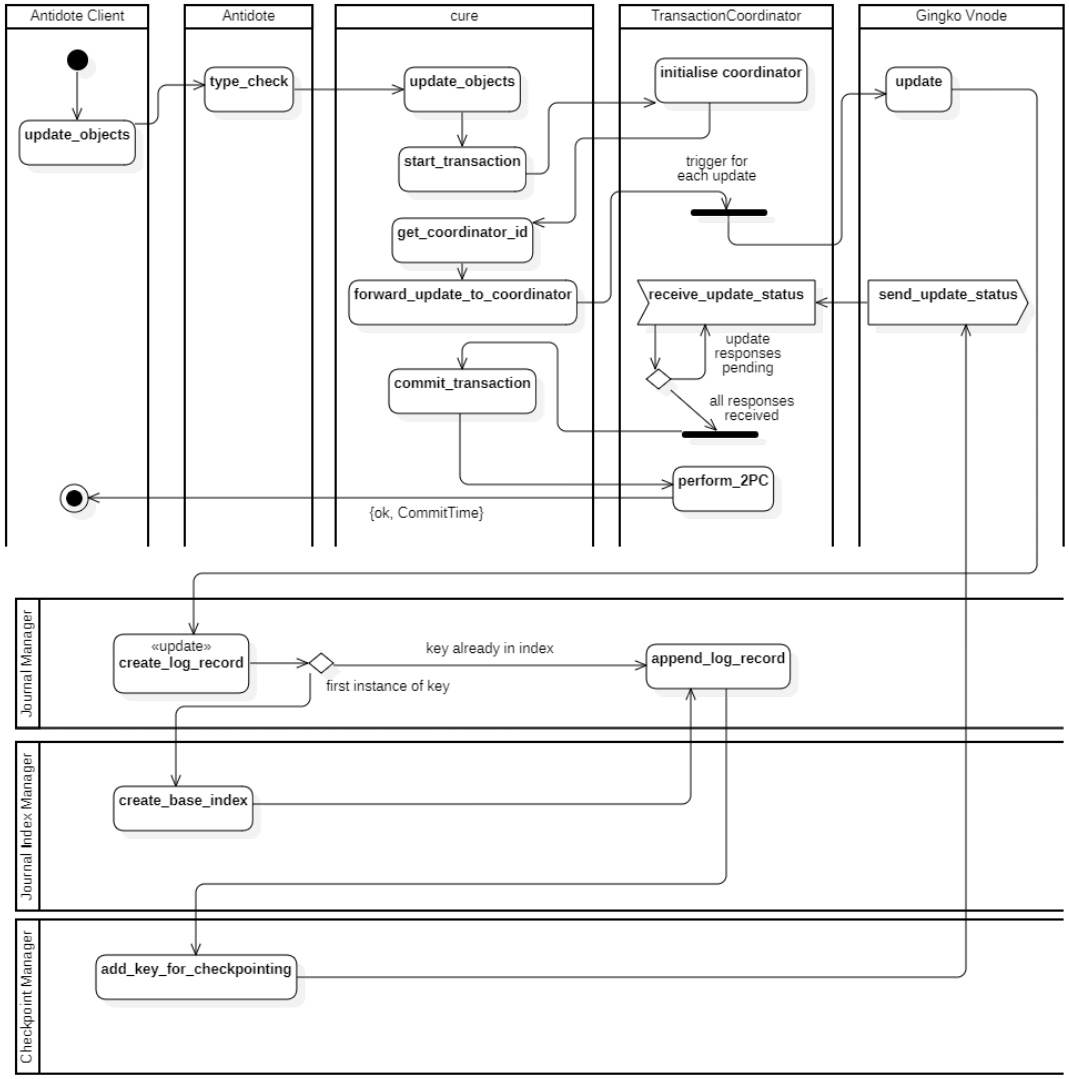


**Figure 3.11.:** *Activity diagram for updating objects statically in AntidoteDB*

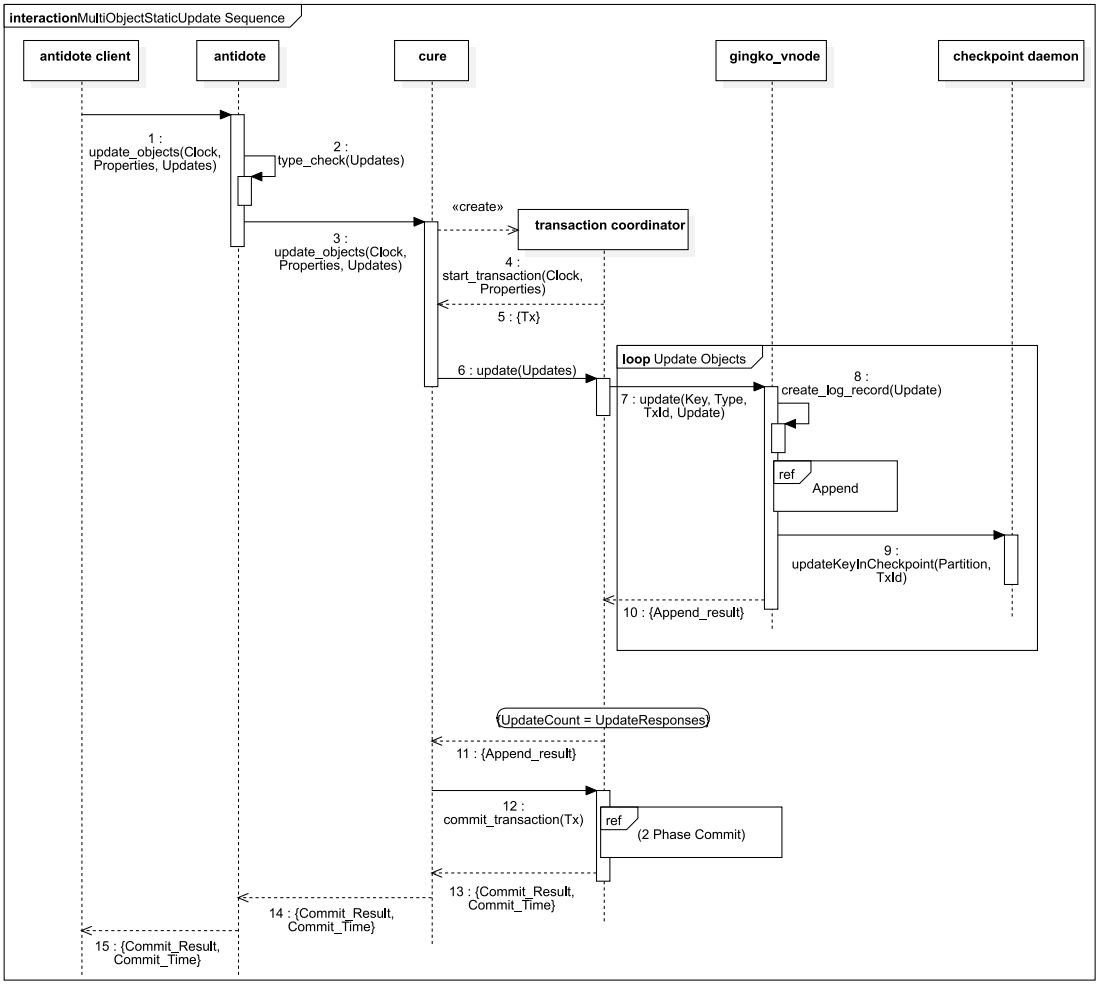


**Figure 3.12.:** *Sequence diagram for a call, updating objects within a query statically, sent to Gingko*

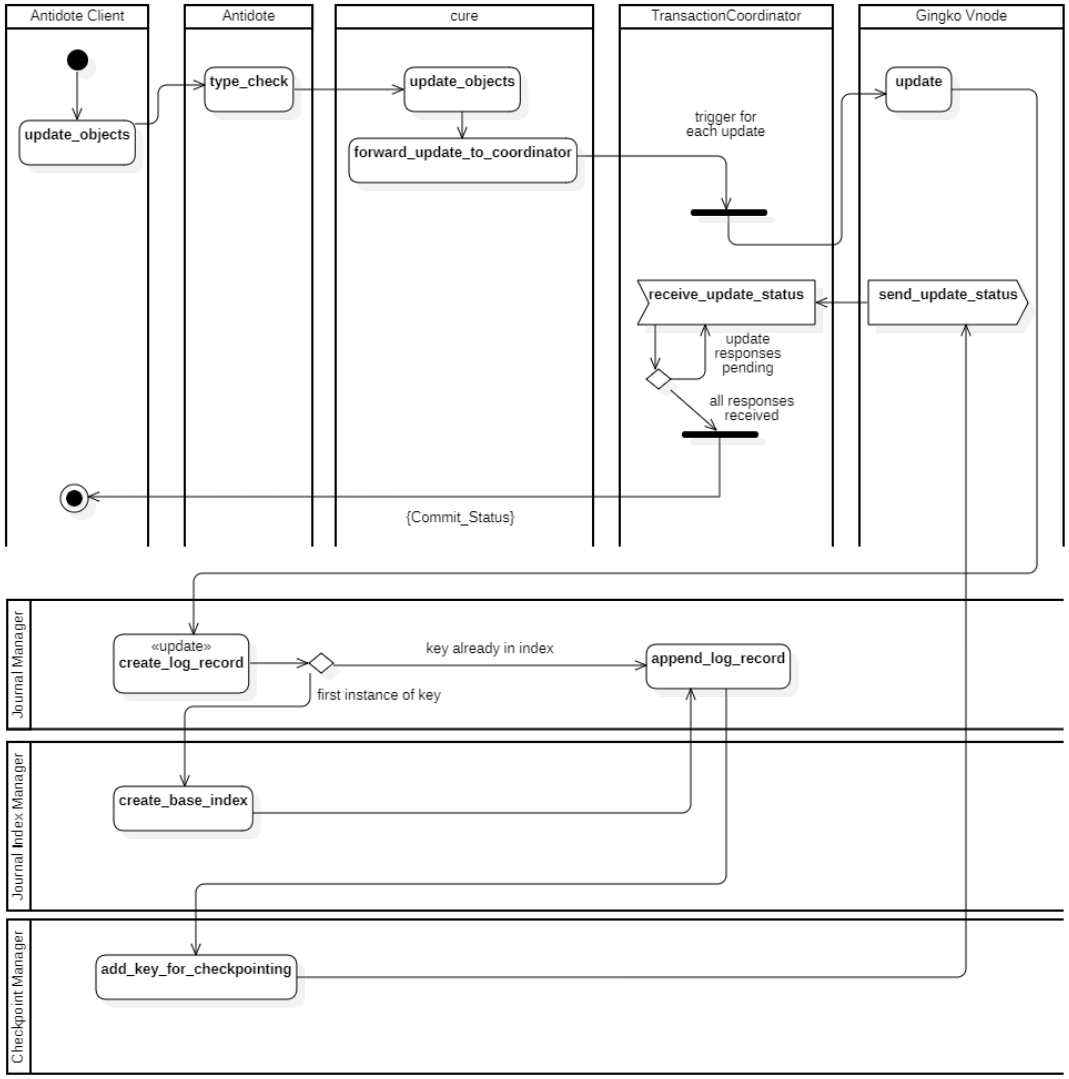


**Figure 3.13.:** *Activity diagram for updating objects within a transaction in AntidoteDB*

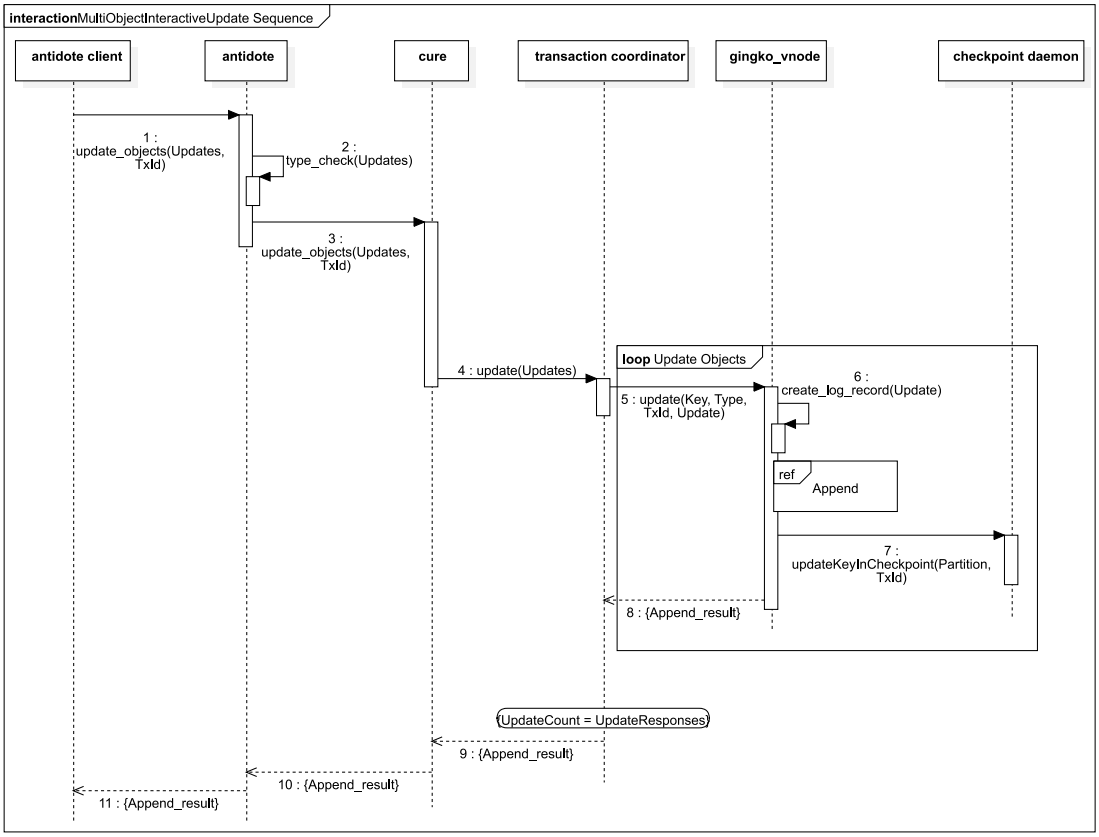


**Figure 3.14.:** *Sequence diagram for a call, updating objects within a query statically, sent to Gingko*

process id for the coordinator is not valid anymore. The termination is noticed by the transaction coordinator supervisor [3].

Any updates issued by this transaction that have been added to the journal will be irrelevant since a commit is required for the operations to materialise. Thus, a faulty transaction has no semantic side-effects.

### 3.4.2. Unresponsive or Unreachable Vnodes

If a vnode is busy or unreachable, which can be due to network faults or high latency that leads to long response times, the configurable replication strategy [37] within riak can rescue performance. As a default configuration, a single vnode is responsible for a set of keys. This can be changed to a value upto N, where N is the number of partitions. In such a configuration, a value is stored in N-1 partitions following the claimant partition for the key in the ring. In case the claimant is unresponsive, another vnode can respond to the query, maintaining the performance.

This does come at a cost. An update in an N replicated ring is considered successful if at-least N/2 nodes successfully write it to their partitions. Similarly, a read needs to be answered by at-least N/2 partitions to be considered consistent and correct. Both these methods require a quorum which sacrifices performance [38]. To avoid running expensive quorums, in our implementation the replication parameter is kept to 1 and other persistence mechanisms like the checkpoint store and the journal are used to avoid missing updates in case of failures.

### 3.4.3. Failure of AntidoteDB components

Within a single data-center, there are several processes that handle operations: Cache manager, Journal index manager, Checkpoint store etc. These processes help improve the performance of the reads, but are not vital to the correctness of the operations and do not improve on the guarantees that AntidoteDB provides. The guarantees are ensured by the materializer, the journal and the AntidoteDB client API, all of which are static modules and not processes so if everything fails, the read and update operations will still be successful albeit at a very slow rate. Some indications of the impact of the helper processes are revealed in the benchmarks (see chapter 4).

# 4. System Performance and Evaluation

## 4.1. Introduction to the Benchmark system (RCL Bench)

The benchmarking of distributed application often requires special deployments to generate loads and keeping record of throughputs and other metrics. To benchmark AntidoteDB, Riak Core Lite Benchmark (RCL bench), which is a simplification of Riak Bench, is used [33]. RCL bench has an extendable interface that can use different drivers that generate loads and call the AntidoteDB APIs. The drivers are implemented in Erlang and the metrics are collected and presented by the benchmark tool.

## 4.2. Benchmark setup

For our benchmarking setup, AntidoteDB was deployed on a cluster with a local load generator machine. The database is on a machine with a 3.6 GHz 8 core CPU and 16GB of RAM. The load generator is on a machine with a 1.8 GHz, 4 core CPU with 8GB of RAM. The Machines are on the same local network and connected to a single access point so the network latency is negligible. The connection between the two machines is stable and wired to prevent network faults.

AntidoteDB is started with a base configuration of 16 vnodes in the riak ring. The cache for the base benchmark contains 2 partitions with a segment size of 2000 objects each. This parameter is varied throughout benchmarks to compare relative performance of the cache in different setups. The benchmark is started with 32 concurrent workers unless specified otherwise.

The operations issued by the benchmark are reads and writes. Every operation is performed on a counter CRDT. The write operations implicitly start a transaction whereas the reads can be static. For most benchmarks presented ahead a read to write ratio of 8:2 was used. This is representative of a typical real life usage.

## 4.3. Benchmark Hypotheses and Results

Before varying the different system component parameters and measuring their impact on performance, it is essential to set up a baseline to gain a perspective towards how the performance is influenced.

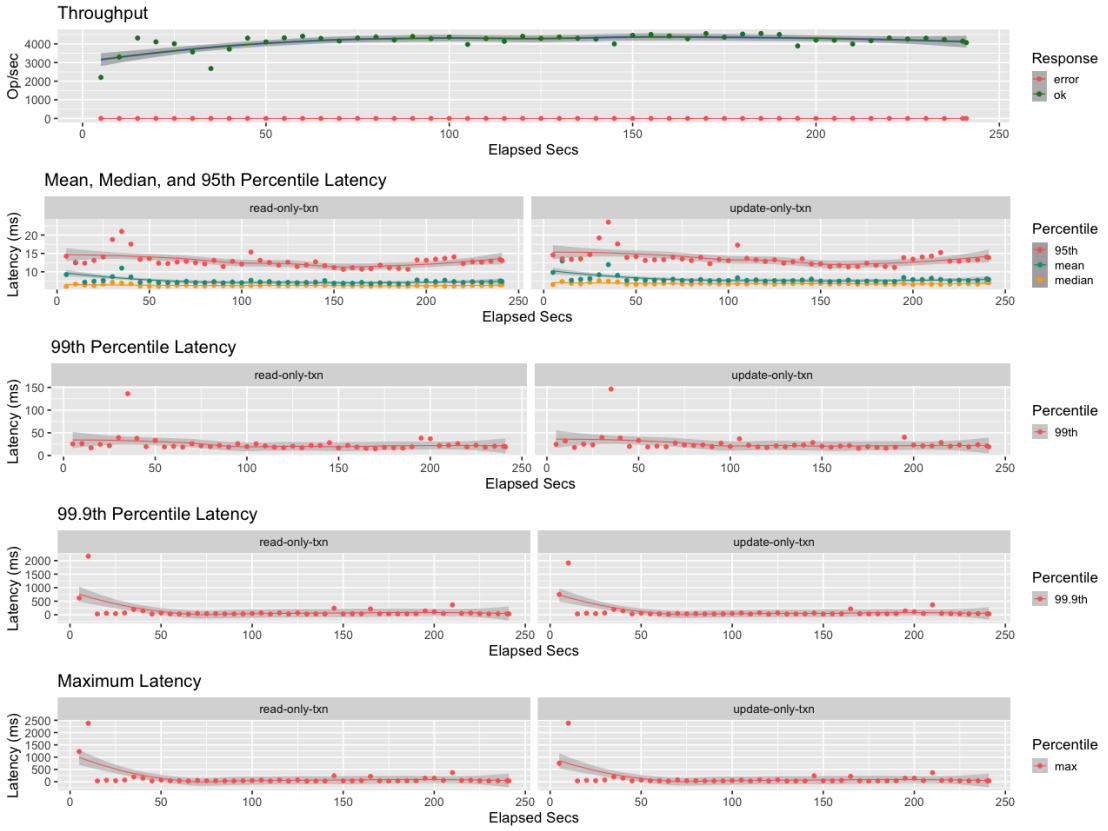


**Figure 4.1.:** *Baseline performance of AntidoteDB*

### 4.3.1. Setting up a Baseline

The original AntidoteDB implementation served as the starting point for our development. The components within AntidoteDB were then replaced or restructured into Gingko which was then re-integrated as a library. AntidoteDB's performance was measured with 32 concurrent workers for a set of 1000 keys distributed uniformly. For this, AntidoteDB was started with the default parameters of our benchmark configuration. The peak performance averages around 4000 op/s. The benchmark run is shown in figure 4.1

In a similar fashion, a benchmark was run for Gingko to measure the baseline. The benchmark and system configuration was kept the same. In this benchmark, the new implementation gives about 3500 op/s. The benchmark run is shown in figure 4.2

### 4.3.2. Impact of Workload Distribution on Performance

**Hypothesis 1.** *AntidoteDB with Gingko is more performant for read heavy loads.*

To study the effect of workloads on performance, AntidoteDB was deployed with 16 Vnodes in the riak ring. The load generation was done by 64 workers over a set of

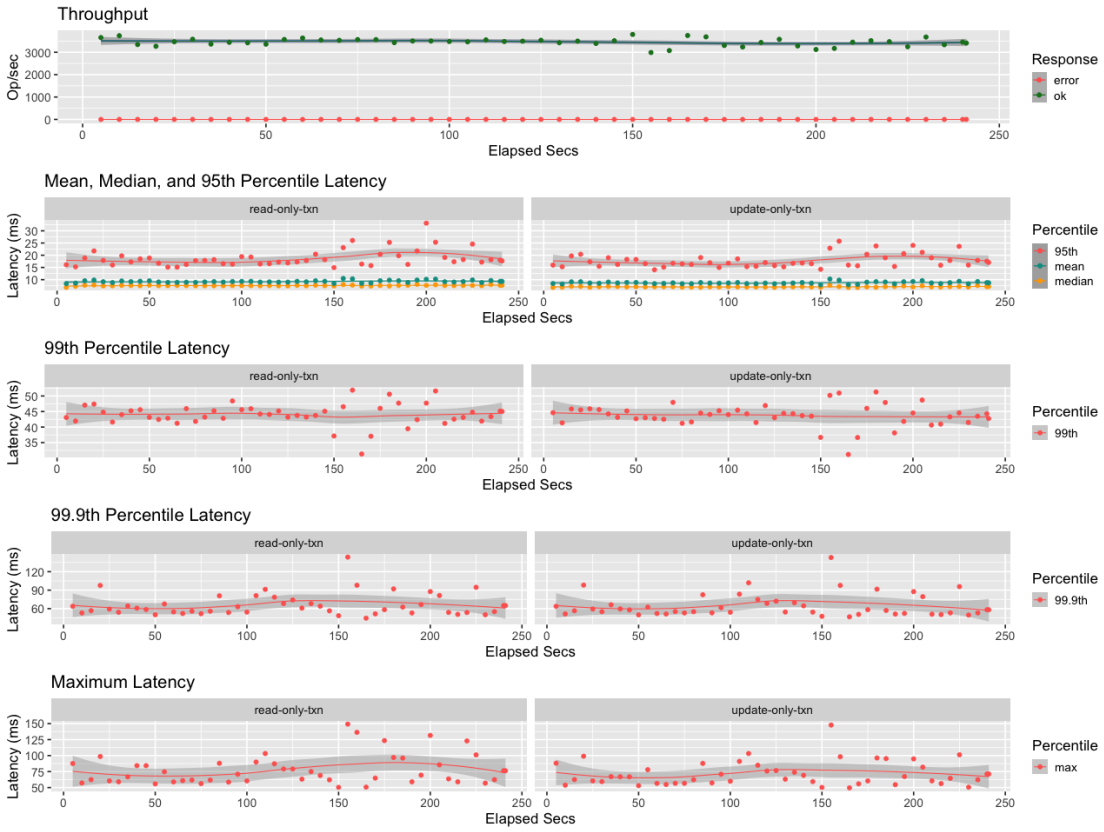


**Figure 4.2.:** *Baseline performance of AntidoteDB with Gingko*

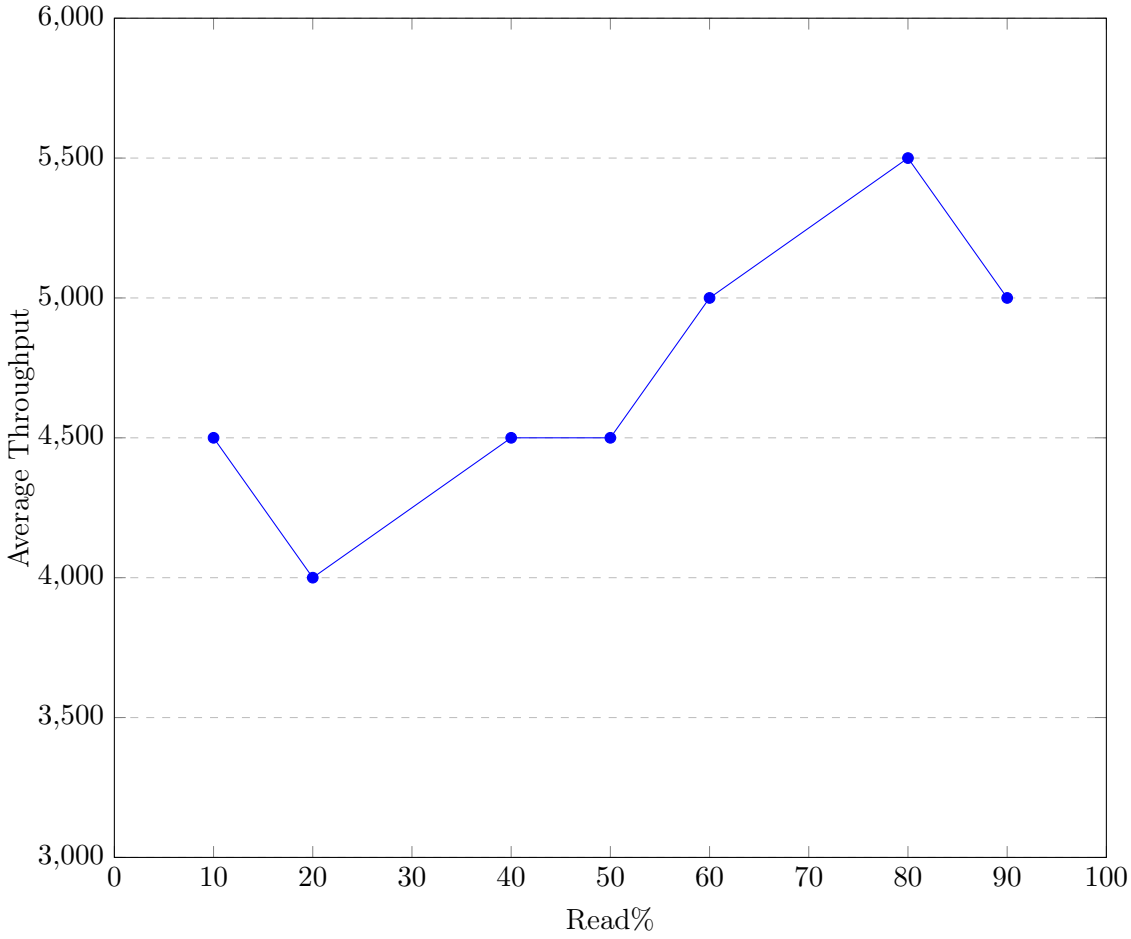


**Figure 4.3.:** *Observing the variance of AntidoteDB performance for different workloads*

1000 uniformly distributed keys [25]. The plot shows a general trend that indicates that benchmark loads with a higher percentage of read operations than write operations are more performant.. The trend of performance is shown in figure 4.3.

### 4.3.3. Impact of Key Distribution on Performance

**Hypothesis 2.** *Skewed Key distributions under-utilise the available vnodes.*

The assumption for this hypothesis is based on the fact that calls are routed to specific vnodes within AntidoteDB. A smaller distribution concentrates the load on a few vnodes but for a larger distribution, the load is spread across several vnodes. The load generation was done by 32 workers and consists of 80% reads [26].

The trend of performance is shown in figure 4.4. The trend presents a supporting conclusion. Larger key distributions favor performance but this performance plateaus

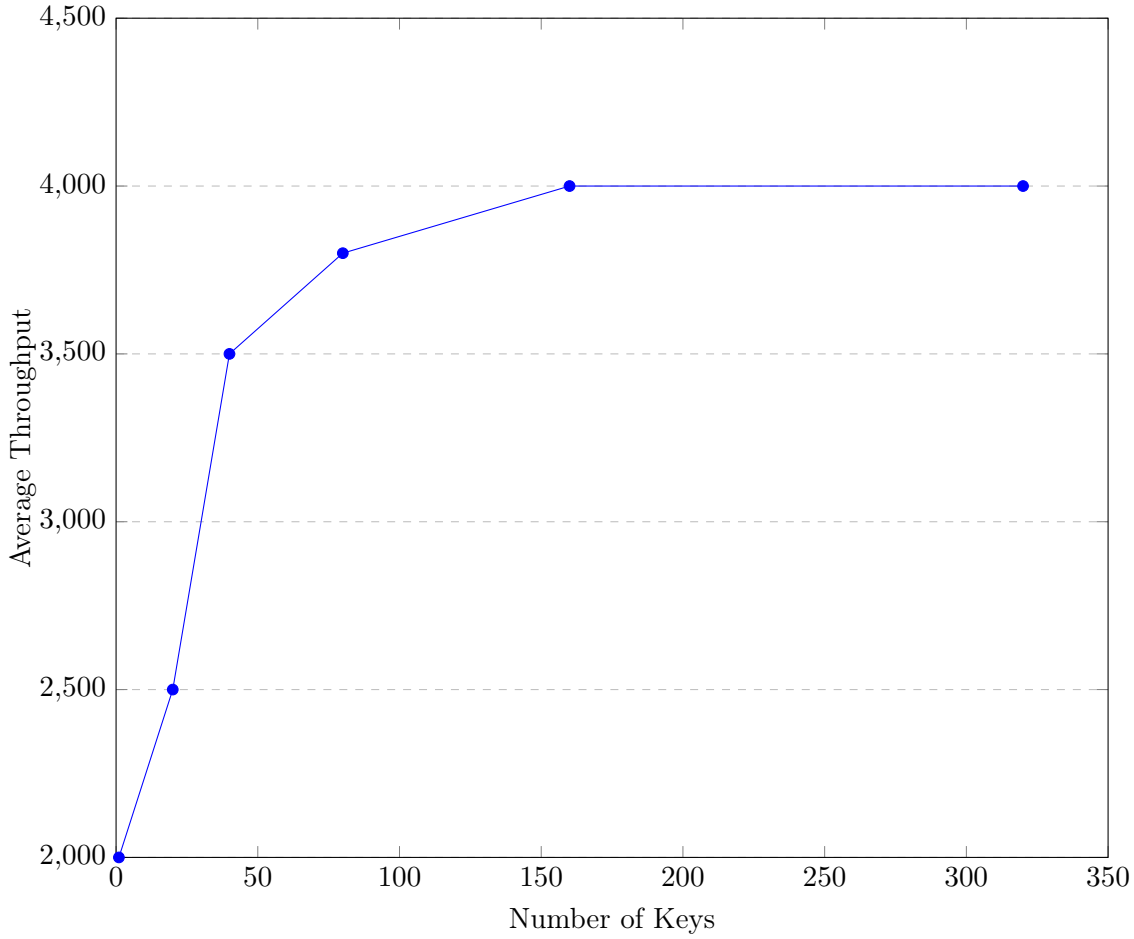


**Figure 4.4.:** *Observing the variance of AntidoteDB performance against distribution of calls to vnodes*

because after a while, the vnodes reach their peak performance which can only be improved by tuning other aspects of the database setup.

### 4.3.4. Impact of Vnodes on Performance

**Hypothesis 3.** *More vnodes facilitate higher performance.*

The number of vnodes, configured with the system, decides the scalability of the system. Higher the number of vnodes the more workers are available to answer to queries and the smaller the journal files are because a single vnode is responsible for a smaller keyset.

To study this aspect of system performance, AntidoteDB was deployed with multiple different vnode configurations starting from 2 upto 256. The number of vnodes is always an exponent of 2 due to the design of riak. Since a fixed number of load generators

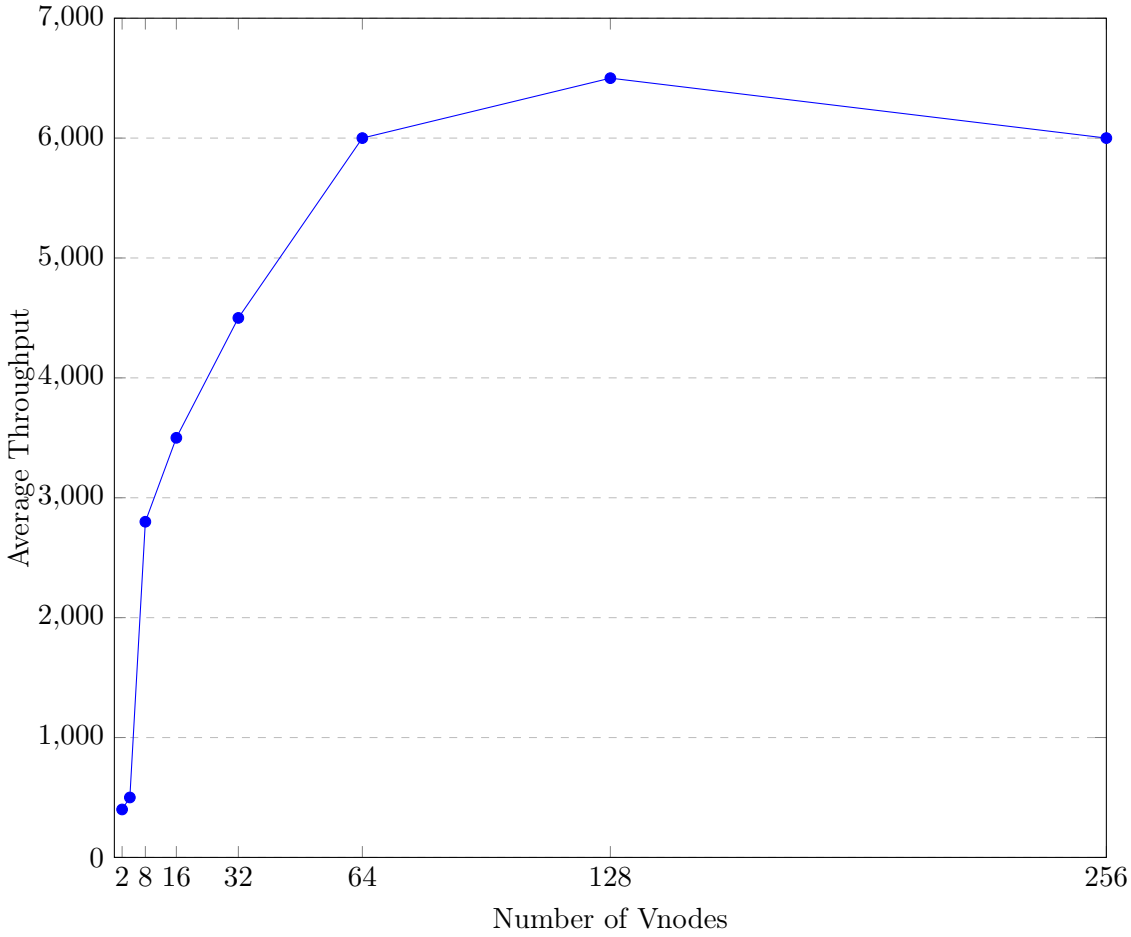


**Figure 4.5.:** *Observing the variance of AntidoteDB performance for different vnode configurations*

cannot lead to peak performance, the number of workers was also changed according to the vnode configuration[27]. The result of this analysis is shown in figure 4.5

The trend graph indicates the scalability that is achieved by increasing the number of vnodes. As we reach higher number of vnodes, the performance starts to decline because we are reaching the limit of the computational capacity of a single machine. A deployment over multiple physical machines would offer better results.

### 4.3.5. Impact of the Cache on Performance

**Hypothesis 4.** *Caching objects for reads leads to higher performance.*

For this benchmark, the journal was populated with 1804800 records for 320 keys. The database was then restarted with an empty cache. For the first test, the reads were performed in the same order as the writes and performance is measured. During this

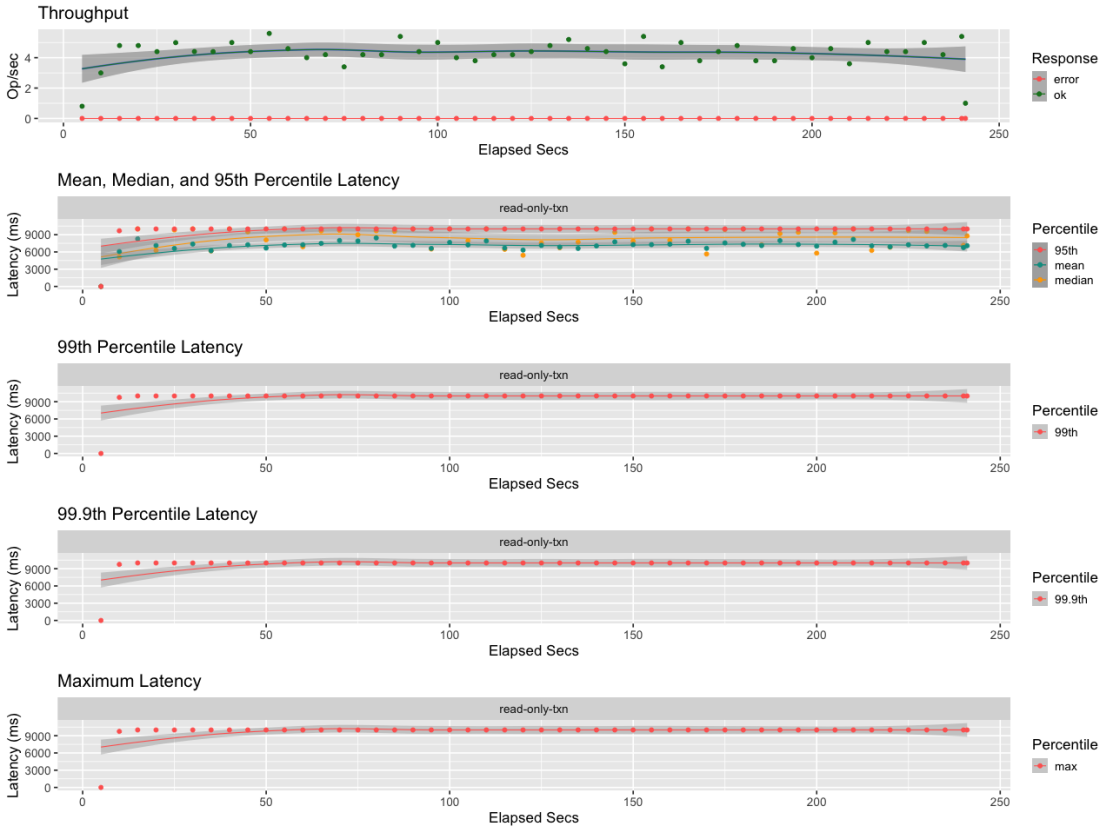


**Figure 4.6.:** *Performance of reads in AntidoteDB without a cache*

test, writes to and reads from the cache were disabled so the objects needed to be built from the journal and the performance is determined by disk read performance and how quickly the operations can be filtered for a key. The result is shown in figure 4.6

After the benchmark without a cache, the database was restarted and the cache reads and writes were enabled. During the first few seconds, while the cache is being populated as read requests come in, the performance is low. As soon as the cache is able to provide relevant snapshots for requests, the performance increases drastically. The performance of the system is shown in figure 4.7

### 4.3.6. Impact of Indexing the Journal on Performance

**Hypothesis 5.** *Indexing the journal allows shorter read times and improves performance.*

Index in AntidoteDB allows jumping to strategic positions within the journal. Without the index, for filtering a specific operation, the journal needs to be read as a whole. This should lead to a performance degradation as the journal grows.

To analyse this, the database is subjected to a normal read and write load of 80:20 ratio with the indexing turned off. The result is shown in figure 4.8

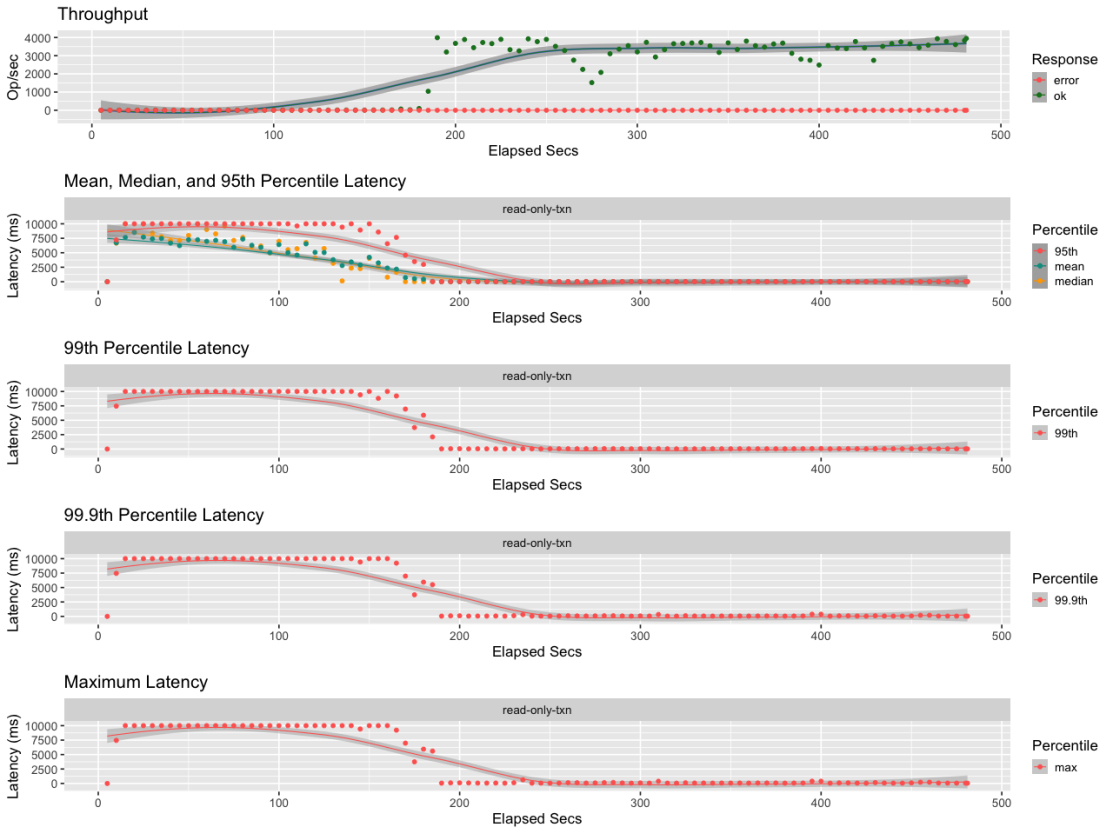


**Figure 4.7.:** *Observed change in performance after the cache is populated in AntidoteDB*

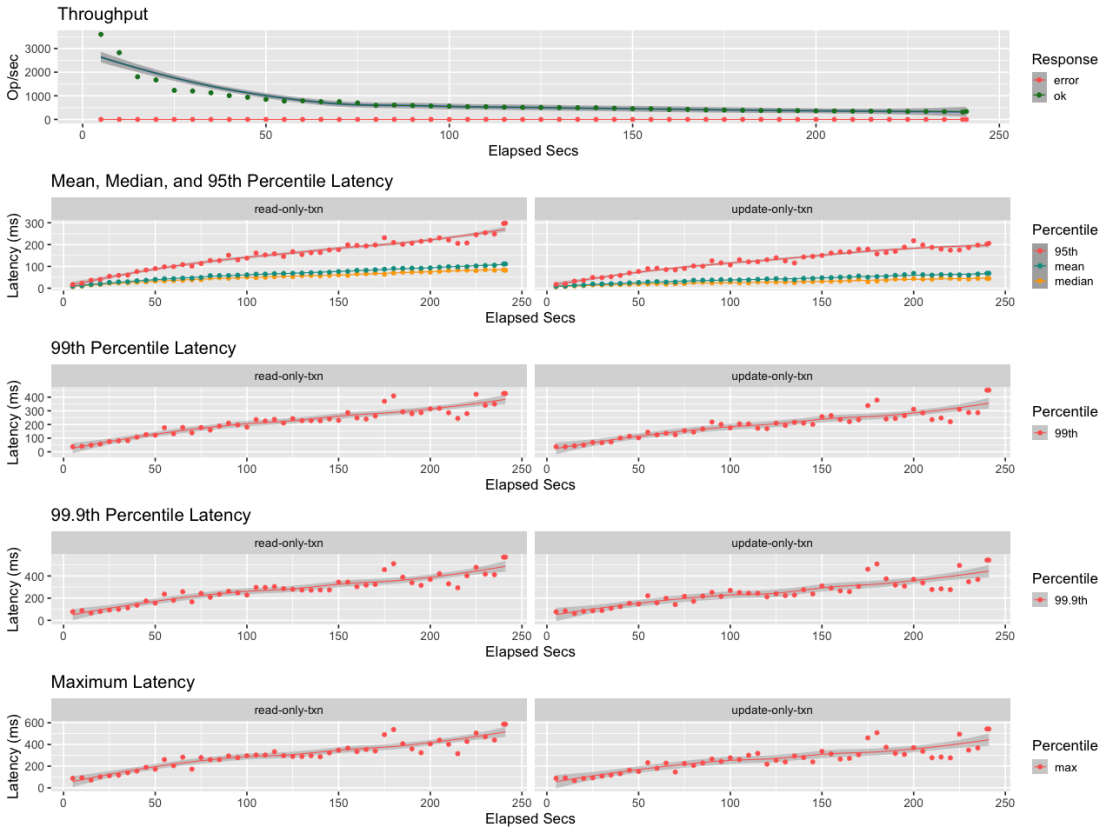


**Figure 4.8.:** *Observed degradation in performance without the journal index*

When the indexing is enabled, the usual baseline database performance can be achieved again.

# 5. Conclusion and Future

In this thesis, we discussed the design and implementation of a cache and a persistent checkpoint store for AntidoteDB, which is a distributed CRDT data-store. The motivation for the design of these components was outlined in Chapter 1 along with other products that are related to AntidoteDB. The design of the system along with the constraints on the design and the thoughts were explained in Chapter 2. The detailed design of the system and supporting architecture was presented in Chapter 3. This implementation was benchmarked for performance and compared to the original implementation of AntidoteDB. The results and influence of design components on performance was discussed in Chapter 4.

This design and implementation, however, is not complete. There are several directions of improvement.

**Inter Data-Center Synchronisation:** The design of the cache is confined to a single data-center with several partitions. In Geo-replicated databases,it is important to consider Inter-DC communication and synchronization. In this regard, The complexity of synchronising the updates requires additional design decisions which were not a part of our work.

**Inter Data-Center Communication:** The components within a data-center can reliably use Erlang message passing but remote data-centers need messaging guarantees. The design and use of message queues for example ZeroMQ or RabbitMQ [16] needs to be considered. If delivery guarantees can be provided by the message queues then it is also possible to simply the design of the data-center based on these guarantees.

**Maintaining an Operations Cache:** The cache in our design contains materialised objects. The operations needed to materialise those objects need to be read from the journal which involves a disk I/O. The design can include an operations cache which stores the operations in memory. This design would then require thinking about the cache replacement policy for the operations cache to make sure, that operations required for materialising the objects are always available and are only written to the journal when absolutely necessary.

**Eager Materialization:** In the current implementation, the objects are materialised only when they are read. It would be interesting to see the impact of eager materialisation on the performance where materialised versions of the objects are stored as the operations come in and the committing transactions leave a copy of the materialised objects in the

cache which can be easily read by other transactions. Eager materialisation would be supported by the operations cache and would this help with the garbage collection of the operations cache as well.

# Bibliography


[1] Ericsson AB. 2021. URL: https://www.erlang.org/doc/design_principles/gen_server_concepts.html.

[2] Ericsson AB. 2021. URL: https://www.erlang.org/doc/design_principles/des_princ.html#behaviours.

[3] Ericsson AB. 2022. URL: https://www.erlang.org/doc/reference_manual/processes.html#errors.

[4] Ericsson AB. *Erlang - Disk Log*. Dec. 2021. URL: https://www.erlang.org/doc/man/disk_log.html.

[5] Ericsson AB. *Erlang Efficiency Guide - Tables and Databases*. Dec. 2021. URL: https://www.erlang.org/doc/efficiency_guide/tablesdatabases.

[6] Mehdi Ahmed-Nacer, Stéphane Martin, and Pascal Urso. "File system on CRDT". In: *CoRR* abs/1207.5990 (2012). arXiv: 1207.5990. URL: http://arxiv.org/abs/1207.5990.

[7] Mehdi Ahmed-Nacer et al. "Evaluating CRDTs for real-time document editing". In: *Proceedings of the 2011 ACM Symposium on Document Engineering, Mountain View, CA, USA, September 19-22, 2011*. Ed. by Matthew R. B. Hardy and Frank Wm. Tompa. ACM, 2011, pp. 103–112. DOI: 10.1145/2034691.2034717. URL: https://doi.org/10.1145/2034691.2034717.

[8] Deepthi Devaki Akkoorath et al. "Cure: Strong Semantics Meets High Availability and Low Latency". In: *36th IEEE International Conference on Distributed Computing Systems, ICDCS 2016, Nara, Japan, June 27-30, 2016*. IEEE Computer Society, 2016, pp. 405–414. DOI: 10.1109/ICDCS.2016.98. URL: https://doi.org/10.1109/ICDCS.2016.98.

[9] ArangoDB. *Introduction to ArangoDB*. Jan. 2022. URL: https://www.arangodb.com/docs/stable/index.html.

[10] David F. Bacon et al. "Spanner: Becoming a SQL System". In: *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. Ed. by Semih Salihoglu et al. ACM, 2017, pp. 331–343. DOI: 10.1145/3035918.3056103. URL: https://doi.org/10.1145/3035918.3056103.

[11] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. "Making operation-based CRDTs operation-based". In: *Proceedings of the First Workshop on the Principles and Practice of Eventual Consistency, PaPEC@EuroSys 2014, April 13, 2014, Amsterdam, The Netherlands*. Ed. by Marc Shapiro. ACM, 2014, 7:1–7:2. DOI: 10.1145/2596631.2596632. URL: https://doi.org/10.1145/2596631.2596632.

[12] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. "Pure Operation-Based Replicated Data Types". In: *CoRR* abs/1710.04469 (2017). arXiv: 1710.04469. URL: http://arxiv.org/abs/1710.04469.

[13] Shanshan Chen et al. "Towards Scalable and Reliable In-Memory Storage System: A Case Study with Redis". In: *2016 IEEE Trustcom/BigDataSE/ISPA, Tianjin, China, August 23-26, 2016*. IEEE, 2016, pp. 1660–1667. DOI: 10.1109/TrustCom.2016.0255. URL: https://doi.org/10.1109/TrustCom.2016.0255.

[14] Cisco. *VNI Complete Forecast Highlights*. Jan. 2022. URL: https://www.cisco.com/c/dam/m/en_us/solutions/service-provider/vni-forecast-highlights/pdf/Global_2021_Forecast_Highlights.pdf.

[15] Giuseppe DeCandia et al. "Dynamo: amazon's highly available key-value store". In: *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*. Ed. by Thomas C. Bressoud and M. Frans Kaashoek. ACM, 2007, pp. 205–220. DOI: 10.1145/1294261.1294281. URL: https://doi.org/10.1145/1294261.1294281.

[16] Nicolas Estrada and Hernán Astudillo. "Comparing scalability of message queue system: ZeroMQ vs RabbitMQ". In: *2015 Latin American Computing Conference, CLEI 2015, Arequipa, Peru, October 19-23, 2015*. IEEE, 2015, pp. 1–6. DOI: 10.1109/CLEI.2015.7360036. URL: https://doi.org/10.1109/CLEI.2015.7360036.

[17] Seth Gilbert and Nancy Lynch. "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services". In: *SIGACT News* 33.2 (June 2002), 51â59. ISSN: 0163-5700. DOI: 10.1145/564585.564601. URL: https://doi.org/10.1145/564585.564601.

[18] Ramakrishna Karedla, J. Spencer Love, and Bradley G. Wherry. "Caching Strategies to Improve Disk System Performance". In: *Computer* 27.3 (1994), pp. 38–46. DOI: 10.1109/2.268884. URL: https://doi.org/10.1109/2.268884.

[19] Protocol Labs. *IPFS Cluster: Consensus Component*. Jan. 2022. URL: https://cluster.ipfs.io/documentation/guides/consensus/.

[20] Leslie Lamport. "Paxos Made Simple, Fast, and Byzantine". In: *Procedings of the 6th International Conference on Principles of Distributed Systems. OPODIS 2002, Reims, France, December 11-13, 2002*. Ed. by Alain Bui and Hacène Fouchal. Vol. 3. Studia Informatica Universalis. Suger, Saint-Denis, rue Catulienne, France, 2002, pp. 7–9.

[21] Leslie Lamport. "The Part-Time Parliament". In: *ACM Trans. Comput. Syst.* 16.2 (1998), pp. 133–169. DOI: 10.1145/279227.279229. URL: https://doi.org/10.1145/279227.279229.

[22] C. Mohan and B. Lindsay. “Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions”. In: *SIGOPS Oper. Syst. Rev.* 19.2 (Apr. 1985), 40â52. ISSN: 0163-5980. DOI: 10.1145/850770.850772. URL: https://doi.org/10.1145/850770.850772.

[23] Petru Nicolaescu et al. “Yjs: A Framework for Near Real-Time P2P Shared Editing on Arbitrary Data Types”. In: *Engineering the Web in the Big Data Era - 15th International Conference, ICWE 2015, Rotterdam, The Netherlands, June 23-26, 2015, Proceedings*. Ed. by Philipp Cimiano et al. Vol. 9114. Lecture Notes in Computer Science. Springer, 2015, pp. 675–678. DOI: 10.1007/978-3-319-19890-3_55. URL: https://doi.org/10.1007/978-3-319-19890-3%5C_55.

[24] Gérald Oster et al. “Data consistency for P2P collaborative editing”. In: *Proceedings of the 2006 ACM Conference on Computer Supported Cooperative Work, CSCW 2006, Banff, Alberta, Canada, November 4-8, 2006*. Ed. by Pamela J. Hinds and David Martin. ACM, 2006, pp. 259–268. DOI: 10.1145/1180875.1180916. URL: https://doi.org/10.1145/1180875.1180916.

[25] Ayush Pandey. *Comparing workload performance in AntidoteDB*. Jan. 2022. URL: https://github.com/ayushpandey8439/antidote_bench_results/tree/main/Different%20read%20write%20ratios%201000%20keys%2064%20workers.

[26] Ayush Pandey. *Skewed Key Distribution performance*. Jan. 2022. URL: https://github.com/ayushpandey8439/antidote_bench_results/tree/main/Skewed%20Load%20Performance%20Benchmark%2032%20workers.

[27] Ayush Pandey. *Variable vnode configuration performance*. Jan. 2022. URL: https://github.com/ayushpandey8439/antidote_bench_results/tree/main/Variable%20Vnode%20Configurations%202:8%20W:R.

[28] Nuno M. Preguiça, Carlos Baquero, and Marc Shapiro. “Conflict-Free Replicated Data Types CRDTs”. In: *Encyclopedia of Big Data Technologies*. Ed. by Sherif Sakr and Albert Y. Zomaya. Springer, 2019. DOI: 10.1007/978-3-319-63962-8_185-1. URL: https://doi.org/10.1007/978-3-319-63962-8%5C_185-1.

[29] Tuanir França Rezende and Pierre Sutra. “Leaderless State-Machine Replication: Specification, Properties, Limits”. In: *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*. Ed. by Hagit Attiya. Vol. 179. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 24:1–24:17. DOI: 10.4230/LIPIcs.DISC.2020.24. URL: https://doi.org/10.4230/LIPIcs.DISC.2020.24.

[30] Marc Shapiro Saalik Hatia. *Specification of a Transactionally and Causally-Consistent (TCC) database*. Tech. rep. DELYS; LIP6, Sorbonne UniversitÃ©, Inria, Paris, France., 2020.

[31] Marc Shapiro and Yasushi Saito. "Scaling Optimistic Replication". In: *Future Directions in Distributed Computing, Research and Position Papers*. Ed. by André Schiper et al. Vol. 2584. Lecture Notes in Computer Science. Springer, 2003, pp. 164–172. DOI: 10.1007/3-540-37795-6_33. URL: https://doi.org/10.1007/3-540-37795-6%5C_33.

[32] Marc Shapiro et al. *Just-Right Consistency: reconciling availability and safety*. 2018. arXiv: 1801.06340 [cs.DC].

[33] Basho Technologies. *Benchmarking*. Jan. 2022. URL: https://docs.riak.com/riak/kv/2.2.3/using/performance/benchmarking/index.html.

[34] Basho Technologies. *Cluster Capacity Planning*. Dec. 2021. URL: https://docs.riak.com/riak/kv/2.2.3/setup/planning/cluster-capacity.1.html.

[35] Basho Technologies. *Cluster Gossiping*. Jan. 2022. URL: https://docs.riak.com/riak/kv/latest/learn/concepts/clusters/index.html#gossiping.

[36] Basho Technologies. *Handoff Reference*. Jan. 2022. URL: https://docs.riak.com/riak/kv/latest/using/reference/handoff/index.html.

[37] Basho Technologies. *Intelligent replication*. Jan. 2022. URL: https://docs.riak.com/riak/kv/latest/learn/concepts/clusters/index.html#intelligent-replication.

[38] Basho Technologies. *Primary Reads and Writes*. Jan. 2022. URL: https://docs.riak.com/riak/kv/2.2.3/developing/usage/replication.1.html#primary-reads-and-writes-with-pr-and-pw.

[39] Basho Technologies. *Riak Key-Value Store*. Jan. 2022. URL: https://docs.riak.com/riak/kv/2.2.3/learn/why-riak-kv/index.html.

[40] Romain Vaillant et al. "CRDTs for truly concurrent file systems". In: *HotStorage '21: 13th ACM Workshop on Hot Topics in Storage and File Systems, Virtual Event, USA, July 27-28, 2021*. Ed. by Philip Shilane and Youjip Won. ACM / USENIX Association, 2021, pp. 35–41. DOI: 10.1145/3465332.3470872. URL: https://doi.org/10.1145/3465332.3470872.

[41] Alexandre Verbitski et al. "Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases". In: *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. Ed. by Semih Salihoglu et al. ACM, 2017, pp. 1041–1052. DOI: 10.1145/3035918.3056101. URL: https://doi.org/10.1145/3035918.3056101.

[42] Stéphane Weiss, Pascal Urso, and Pascal Molli. "Logoot: A Scalable Optimistic Replication Algorithm for Collaborative Editing on P2P Networks". In: *29th IEEE International Conference on Distributed Computing Systems (ICDCS 2009), 22-26 June 2009, Montreal, Québec, Canada*. IEEE Computer Society, 2009, pp. 404–412. DOI: 10.1109/ICDCS.2009.75. URL: https://doi.org/10.1109/ICDCS.2009.75.

# A. Definitions

**Definition 1.** ***Read Committed Isolation****: Read committed isolation is concurrency control technique which prevents dirty reads. A dirty read happens when a transaction is allowed to read a value or object which was modified by a transaction that ha not committed yet. Read committed isolation does not however ensure that two reads to the same key will return the same value. This is because another transaction can update the value and commit between when the two reads are performed.*

**Definition 2.** ***Linearizability and Atomic Consistency****: In a concurrent setting where multiple processes can share the data, a set of operations, issued by concurrent processes, is linearizable if for a sequence of reads and writes as well as their responses satisfies the following criteria.*

- *The operations and their responses can be rearranged such that they are not concurrent anymore but the execution of this new sequence does not change the result of any reads or writes in the original sequence. This property is called Serializability.*
- *The rearranged sequence of operations is a subset of the original sequence i.e. we cannot add more operations but rather only reorder them.*

Linearizability is also called as *Atomic Consistency*.

# B. Supporting Material

- The original AntidoteDB implementation is available on github here: `https://github.com/AntidoteDB/antidote.git`
- The modifications in AntidoteDB made as a part of this work are available here: `https://github.com/ayushpandey8439/antidote/tree/complete_refractor`
- The logging, materialization and checkpoint library (Gingko) is available here: `https://github.com/ayushpandey8439/gingko/tree/gingko-cache`

**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