Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Scalable multiversion handling for geo-replicated storage

Tesis presentada para optar al título de
Licenciado en Ciencias de la Computación

Santiago Javier Alvarez Colombo

Advisors: Msc. Alejandro Tomsic and Dr. Marc Shapiro (LIP6-INRIA)
Co-advisor: Dr. Sergio Yovine (DC-UBA)
Paris - Buenos Aires, 2017

MANEJO DE MULTIVERSIÓN ESCALABLE PARA BASES DE DATOS GEO-REPLICADAS

Antidote, una base de datos Geo-Replicada, implementa estructuras de datos replicadas multiversión. Este tipo de bases de datos, proporciona un módulo que da soporte a la multiversión, manejando todas las lecturas y escrituras al almacenar en caché y persistir las distintas versiones de los objetos. Este módulo es crítico para la performance y tolerancia a fallas de Antidote.

La implementación actual de este módulo en Antidote tiene algunas desventajas. Por ejemplo, no es tolerante a fallas, se realiza garbage collection de forma sincrónica (afectando el throughput) y la base de código está fuertemente acoplada, por lo que introducir un cambio en este módulo conlleva una gran complejidad.

Para solucionar estos problemas, diseñamos un nuevo módulo de cache y persistencia basado en LevelDB, llamado Antidote-LevelDB. LevelDB está diseñado para datos no versionados, y solo admite operaciones de lectura y escritura. Adaptarlo para soportar multiples versiones de objetos, actualizaciones concurrentes y estructuras de datos replicadas multiversión de alto nivel, como las que se soportan en Antidote, plantea muchos desafíos nuevos. Esta tesis discute estos problemas, nuestro diseño e implementación. Nuestros experimentos muestran que Antidote-LevelDB corrige muchos de los problemas del modelo anterior, aumentando el throughput en la mayoría de los escenarios, al mismo tiempo que proporciona tolerancia a fallas y modularidad.

Antidote-LevelDB es un primer paso hacía proveer una solución completa para el manejo de multiversión de las estructuras de datos replicadas provistas por Antidote. Este es un enfoque nuevo e innovador para resolver este tipo de problemas, por lo que puede ser considerado el primero en su tipo.

Palabras claves: CRDT, replication, Antidote, LevelDB, caching, logging.

SCALABLE MULTIVERSION HANDLING FOR GEO-REPLICATED STORAGE

Antidote, a cutting edge Geo-Replicated Data Store, implements multi-versioned commutative replicated data types. This kind of databases, provide a module that gives support to multiversion, handling all reads and writes by caching and storing versions of data objects. This module is critical to the database since it is in charge of providing performance and reliability.

Antidote's current implementation of this module has some disadvantages. For example, there is no fault tolerance, garbage collection is done synchronously (impacting throughput), and the code base is strongly coupled.

In order to address these issues, we designed a new storage/caching module based on the LevelDB backend, called Antidote-LevelDB. LevelDB provides very efficient storage and caching, but is designed for non-versioned data, supporting only read an write operations. Adapting it to support different versions, concurrent updates, and high-level commutative replicated data types as the ones supported in Antidote, raises many new challenges. This thesis discusses this issues, our design and implementation. Our experiments show that Antidote-LevelDB fixes many of the issues with the previous model, increasing performance in most scenarios, while providing better fault tolerance.

This new backend module is a first approach to caching and storing persistently the multiversion objects of Antidote. This is a new and innovative approach to solve this type of problems, making it the first of its kind in the field.

Keywords: CRDT, replication, Antidote, LevelDB, caching, logging.

AGRADECIMIENTOS / ACKNOWLEDGEMENTS

To Ale and Marc, for selecting me to work with them for 6 months in Paris, accepting to be the directors of this work, and most importantly for everything they taught me in this years working together. Merci pour tout!

A Sergio, por haber co-dirigido este trabajo, guiandome en estos últimos meses para llegar a tener una tesis como la que hoy estamos presentando.

A Hernán Melgratti y Gervasio Perez por haber aceptado ser jurados de mi tesis y por haberse tomado el tiempo para leer y evaluar el trabajo que realizamos.

To Tyler Crain, the missing co-director in this work. Thanks for everything man, every technical discussion, every talk, everything!

To everyone else in the SyncFree project who I got the honor to work with, specially to Mahsa, Annette, Deepthi, Peter, Vinh and Michal.

A Rudy. Llegar a la UPMC en medio de las vacaciones de verano y casi sin hablar francés, fue muy duro. No podría haber tenido mejor suerte que trabajar enfrente de alguien que me recibió de manera excelente desde el día uno. Gracias por los consejos, los almuerzos y los cafes compartidos.

A todos los docentes y no docentes del Departamento de Computación de la Facultad de Ciencias Exactas de la Universidad de Buenos Aires, por brindarle a todos sus alumnos, educación de primer nivel mundial.

A los amigos que me dio la facultad. Empezando por el eterno Algo2 con los que compartí la carrera casi desde el primer día: Maurito, Juanma, Tincho, Bender, Nico y Pablito. A los que conocí mas adelante y con quienes cursé las últimas materias: Santi, Andy y Nico. Y a los que conocí siendo ayudante como Partu. Gracias a todos, esto hubiese sido imposible sin ustedes.

A todos mis amigos. A los que conozco hace muchos años y a los que conozco hace menos. Gracias por haber estado siempre que lo necesité y sepan que lamento haberme perdido muchas cosas por tener que estudiar, pero que el día de hoy defendiendo esta tesis, se que valió la pena.

Al PPPP. Además del agradecimiento del punto anterior, para ustedes hay uno aparte. Fueron parte de los que probablemente fueron los mejores 6 meses de mi vida. En esos meses trabajé en esta tesis, viajé, aprendí mucho de mi y de otras culturas, sufrí momentos muy angustiantes, cumplí el sueño de aplicar y conseguir el trabajo que siempre quise y todo fue al lado de ustedes. Gracias, gracias y mas gracias. Que la vida siempre nos encuentre viajando.

A toda la gente de Medallia que me bancó en este último tramo de la tesis, en especial a Santi Ceria, Lean, Franco, Martiniano, Caro, Mati y el Tucu por estar siempre atentos y apoyandome cuando mas lo necesitaba.

A mi familia, porque lo mas importante siempre se deja para el final. Gracias por bancarme todos los días en todo lo que hago. Perdón por mis reacciones en los momentos de sufrimiento. Sepan que aprecié mucho su apoyo todos estos años, por mas que muchas veces no lo haya demostrado por los nervios o preocupación en el momento. Los quiero mucho!

A Beatriz, Kuky, el Negro y Luis

CONTENTS

1. INTRODUCTION

Cloud applications are typically layered above a database engine. A recent trend is to geo-replicate these databases across several data centers (DCs) around the world, in order for clients to avoid wide-area network latency and to tolerate downtime. These databases are also replicated in each DC for horizontal scalability.

With such a setting, the database is highly available and allows high parallelism, which enables low latency and high throughput. This setting has a clear downside, since it can not simultaneously provide Consistency, Availability and Partition Tolerance, as described in the CAP theorem.

The CAP Theorem[1, 2] states that it is impossible for a distributed computer system to ensure Consistency (C), Availability (A) and Partition tolerance (P) at the same time. Because network failures are inevitable, a geo-replicated data store has to provide partition tolerance. This implies Data Base (DB) architects have to choose between Consistency and Availability when designing which guarantees will be provided on each access to the DB. Most architects choose an AP design, since unavailability can not be tolerated.

A geo-replicated system is said to support a given consistency model if its transactions change data only allowed by certain well-defined rules in that model[1]. Given an AP design, the database can not offer strong consistency, since this would break CAP, so recent work has focused on enhancing AP designs with stronger semantics[3–5].

Looking into Weakly Consistent protocols to fit the database design, Causal Consistency is an interesting one to consider. A distributed database provides causal consistency if operations that are causally related are seen by every observer in the same order. Concurrent writes (i.e. not causally related to each other) may be seen in different order by different nodes. Causal consistency is considered a sweet spot in the consistency spectrum since it is the strongest model that fits an AP design.

Conflict-free replicated data types (CRDTs) [6] provide a principled approach to shared, mutable and replicated data. Updates to a CRDT are convergent and do not need synchronization. This allows updates to execute immediately, knowing that those updates will be propagated to, and replayed, at the other DCs. CRDTs can be operation-based or state-based.

An operation-based CRDT propagates its state by only transmitting updates. Replicas receive the updates and apply them locally.

In contrast, a state-based CRDT sends its full local state to other replicas. A replica that receives a remote state merges it, using a merge function.

In this work, we will study Antidote[7] a geo-replicated versioned CRDT (operation-based) data store. Antidote features scalable, conflict-free implementations of transactions, by providing consistent stable snapshots and atomic multi-CRDT updates[8].

[1] This does not guarantee correctness of the transaction in all ways the application developer might have wanted. Guaranteeing correctness is the responsibility of application-level code, but any programming errors cannot result in the violation of the rules.

1.1 State of the art

Today, several large companies have developed their own geo-replicated data store that can tackle their specific problems.

- Google has developed Spanner[9] a scalable, multi-version, geo-replicated, and synchronously-replicated database.
- Facebook has developed Tao[10] a geo-replicated data store that provides efficient and timely access to the social graph for Facebook's workload.
- LinkedIn has developed Ambry[11] a highly available and horizontally scalable distributed object store, optimized for storing trillions of small immutable objects.

This examples illustrate how big companies are developing their own geo-replicated data stores, that can serve their own needs, meeting the ever-increasing demand of their users.

What all these current, well-known, geo-replicated DBs are lacking, is CRDT support. CRDTs are designed to work correctly in the presence of concurrent updates and partial failures, avoiding developers of the pain of adding code to handle this scenarios.

Laking CRDT support, makes a comparison between all this databases and Antidote, clearly unfair. We can find CRDT support in:

- Lasp[12], provides a comprehensive programming system for planetary scale applications. Lasp has a prototype implementation of CRDTs (which is shared with Antidtote) in Erlang[13].
- Riak DB[14], implemented by Basho, offers a set of state-based CRDTs implemented in Erlang.

All these implementations of CRDTs are only an example of some companies putting efforts to add CRDT support for DBs and programming languages. None of them are fully implemented into a causally consistent transactional CRDT geo-replicated DB.

There are currently two op-based CRDT databases. SwiftCloud [15, 16], is a causally-consistent CRDT object database for client-side applications. This distributed object database is the first to provide fast reads and writes via a causally-consistent client-side local cache.

Unlike Swiftcloud, Antidote is a database designed to be used in the back-end of applications. Currently, it is the only planet-scale, available, transactional database with strong semantics and full pure op-based CRDT support. This is the reason why we are considering Antidote, and its implementation of op-based CRDTs, as *state of the art*. This also means there has been no work related to data structures designed to cache and store persistently operations from CRDTs, other than the current implementation in Antidote.

1.2 Problem description and motivation

Since Antidote is a CRDT database under causal consistency, at any given time, a replica may have different versions of the same object available to clients. This implies the replica supports object multiversion in order to allow consistent reads and updates throughout the system. Object versions should be cached and stored persistently, in order to provide performance and durability.

For this work, we will focus on implementations of a caching and logging module in Antidote. This module has to provide durability, so the DB can provide fault tolerance. It has to be efficient, since it is the module that ends up handling all read and write requests. If this module fails to provide durability, the whole DB becomes useless. If it does not provide acceptable performance, it can become the bottleneck for the DB throughput and latency.

The current implementation of Antidote tries to provide all the previously mentioned features, but has several issues:

- After a failure, a log is replayed sequentially, making recovery extremely slow.
- For caching objects, Antidote assumes that all reads are done from memory, which implies that all objects should be kept in memory at all times.
- To remove objects from memory, Antidote implements a simple garbage collection algorithm, which is synchronous, impacting throughput.
- All the code related to caching and logging is strongly coupled, impacting the ability of Antidote developers to introduce changes and to write unit tests.

All these problems are detailed in depth in Section 3.3.

1.3 Objective

The main objective of this thesis, is to study a new way of caching and logging operations in Antidote using a data structure based on a Log-Structured Merge-Tree (LSM-tree)[17] such as LevelDB[18].

Since each tree level in an LSM-Tree is optimized for its underlying storage medium, this data structure is widely known for having performance characteristics that make it perfect for providing access to files such as transactional log data, which have a high insert volume. This makes an LSM-Tree based DB, a good candidate to store the multiple object versions Antidote has to handle.

LevelDB is a key-value datastore written at Google, based on LSM-Trees which is known for its excellent performance, while offering out of the box persistency and a built in cache. This characteristics are the ones we were looking for our new caching-logging module for Antidote.

By changing Antidote's underlying data store to LevelDB, we aim to provide a more scalable and modular solution, at the same time we try to improve performance.

For this work, we proposed ourselves with the thesis "an LSM-tree like data structure can be used for storing multiversion objects of CRDTs". This work verifies this thesis in practice.

We will compare the advantages and disadvantages of this new approach, and run a complete set of benchmarks to understand its practical implications.

1.4 Contributions

Our first contribution is an exhaustive analysis of how Antidote uses its cache and log to store operations and snapshots, identifying its weak spots.

We are also addressing several new systems problems. This thesis analyzes how to support Multi Version Concurrency Control (explained in detail in Section 2.5) with partial order. We also adopt an LSM-tree based backend to Antidote and design and implement an efficient cache and log of op-based CRDTs under causal consistency.

Antidote has a strongly coupled codebase, so we wrote all the new code for this thesis in a standalone module. This module is not coupled to Antidote in any way, making it reusable. This abstraction helped decoupling the existing codebase, but also increased testeability for the whole DB, making Antidote developers more productive.

2. BACKGROUND

In this chapter, we present the notions and basic definitions that will be used in this thesis.

2.1 Consistency

First let's take two examples to analyze why and when consistency is important.

First let's consider a worldwide bank with a geo-replicated database. For any given transaction happening at the bank, we always want to read and write to the most up-to-date information of the accounts involved. Not reading the latest data, could lead to several anomalies (like double spending), which are clearly undesired and hard to roll-back. Since all replicas of the DB always contain up-to-date information, we consider this a *strong consistent* DB.

On the other hand, let's analyze a social network where a famous person has a profile with millions of followers. Each time a fan accesses the social network and sees the amount of *likes* or *comments* on a post from this famous person, do we care to show them the exact, most up-to-date amount? Probably not, since it would not make a difference for the fan. This means, that in any given instant in time, each replica of the database may hold different values for the same object. In this case, the social network is using a *weak consistent* DB.

As stated in previous sections, the CAP theorem states that going with an AP design of a database, implies we have to choose between which Weak Consistency protocol to use.

2.2 Weak consistency

In Weak consistency protocols, the local replica is updated first (returning the control to the client) and then updates are asynchronously propagated to other replicas. The client that initiated the transaction sees the update immediately, but clients in other replicas may take longer time to see it. Therefore, at a given point in time, different replicas of the DB may return different values for the same object.

Weakly consistent protocols provide an always-available model that allows concurrent reads and writes at all replicas, therefore providing high throughput. On the downside, these protocols suffer from ordering concurrent updates. They therefore expose application developers to inconsistency anomalies (the code doesn't receive the updates in order), making them add extra code on top of the protocol, to correct and prevent these potential anomalies, which is error prone.

2.2.1 Eventual Consistency

Eventual Consistency (EC)[19, 20] is the weakest level of consistency offered by a DB. EC protocols achieve high availability which informally guarantees that, in some point in time, all replicas will converge to the same value for a given object, but there are no guarantees of when that will happen.

2.2.2 Causal Consistency

When a transaction performs a read operation followed later by a write operation (in the same object or even on a different one), the write is said to be causally dependent on the read. Similarly, a read operation is causally dependent on the earlier write on the same object that stored the data retrieved by the read. It is important to note that causal dependency is transitive.

A protocol is said to provide **Causal Consistency** [5, 21] if write operations that are causally related are seen by every client of the system in the same order. Concurrent writes (i.e., ones that are not causally related) may be seen in different orders by different clients.

This kind of protocols provide the strongest model compatible with availability, and they are easy to reason about for programmers and users.

2.3 Conflict-free replicated data types (CRDTs)

CRDTs[6] provide convergence under Weak Consistency, where an update can execute immediately, and then be propagated to and replayed at the other DCs. There are two types of CRDTs: State-Based and Operation-Based.

2.3.1 State-Based

In State-Based CRDTs, an update modifies the state of a single replica. Every replica occasionally sends its local state to other replicas. A receiving replica merges the received state into its own state.

A state-based CRDT has a *merge* method that merges the states received from another replica into the local one. This merge function solves potential conflicts (i.e. receiving concurrent non related updates for the same object) in a deterministic way, with no further synchronization. In other words, saying that the merge function is deterministic, is saying it is commutative and associative. The function must also be idempotent since updates that are received more than once, should only be applied once.

2.3.2 Operation-Based

Instead of sending and storing states (which can be large), an Operation-Based CRDT sends an update as an operation. This operation is eventually applied to all replicas. Consequently, an operation-based CRDT does not need a merge function. There is no further synchronization needed once a remote update is delivered to a replica.

Operation-based CRDTs require causal delivery of updates, making them commutative, but may not be idempotent, since the underlying communication infrastructure must therefore ensure that all operations on a replica are delivered to the other replicas, without duplication, but in causal order.

Upstream & Downstream generation for Operation-Based CRDTs

A CRDT client queries and modifies objects by calling operations in its interface, known as *upstream* part of the operation. This operation is performed against a replica of its choice called the source replica. A query executes locally, i.e., entirely at the source replica. An update has two phases: first, the client calls the operation at the source, which may

perform some initial processing. Then, the update is transmitted asynchronously to all replicas; this is called the *downstream* part of the operation.

2.4 Transactional Causal Consistency

A transaction is called AP if it is highly available, meaning it can run without cross site communication.

To support multi-object operations we consider *transactional causal consistency* (TCC), a variant of causal consistency, were all the reads of a causal transaction are issued on the same causally consistent snapshot, and either all of a transaction's updates are visible, or none is.

2.5 Multiversion concurrency control

Let's assume we have someone reading an object from a database, at the same time as someone else is writing to it. This is a classic problem in databases, since it can lead to half-written or inconsistent data. A simple approach could be to make all readers wait until the writer has finished but this approach can be very slow, since it does not allow parallelization of readers, which are all blocked waiting the writer to finish. This problem can be solved using a Multiversion Concurrency Control method (MVCC) [22].

In MVCC each transaction sees a **snapshot** of the data. Any changes made by writes in the transaction will not be seen by concurrent transactions, until the transaction is committed and a new snapshot is available.

When a database implementing a MVCC protocol finishes committing a transaction, it would not override the old snapshot, but it will create a new one, therefore storing **multiple versions** for an object. This allows readers to access a consistent version of the object, even if a new version becomes available in the middle of their read.

This will generate lots of snapshots that will become obsolete in the future, so a mechanism to garbage collect old versions is needed.

2.5.1 Vector Clocks

To enforce causal consistency, databases normally use vector clocks (VCs) to time-stamp events. VCs represent a partial order of these events in a distributed system.

For example, let's consider a database that uses VCs to order transactions, and a transaction that is associated to VC: [{dc1,2},{dc2,1}]. This particular representation, consists of a list of objects (one per DC), indicating the id of the DC, and a monotonically increasing integer indicating the time (in that DC) in which a transaction was executed. In this example, the VC indicates that, the transaction associated to it, was committed at time 2 in DC1 and time 1 in DC2.

Let's now take VC1: [{dc1,2},{dc2,1}] and VC2: [{dc1,1},{dc2,2}]. In this example, we only know that the events associated to VC1 and VC2 happened at about the same time, but we have no information to decide which one happened before. This is the reason why VCs can only provide a partial order of events.

In an AP design, operations can not be totally ordered, so a partial order is created by using the vectorclocks associated to them.

To understand how VCs are compared to decide on this partial order, let's consider VC1 and VC2 as two vector clocks, and see how the comparison of them works:

- **VC1 == VC2** all DCs in VC1 have the same time in VC2.
- **VC1** $\leq$ **VC2** all DCs in VC1 have times $\leq$ than the same DC in VC2.
- **VC1** $\geq$ **VC2** all DCs in VC1 have times $\geq$ than the same DC in VC2.
- **VC1** $\parallel$ **VC2** we say VC1 is concurrent to VC2 ($\parallel$), when (VC1 $\not\leq$ VC2) AND (VC1 $\not\geq$ VC2).

It may occur that a distributed system is running and a new DC is added to it. This causes a new entry to be added in new VCs, which is not present in existing VCs in the system. In this cases, when comparing a VC with less entries than another, the missing entries are considered to be 0.

3. ANTIDOTE

To provide an easier reading of Antidote related topics, when we talk about Antidote, it indicates a general idea of the DB that was not modified in this work. We will use Antidote-DL to talk about the implementation using the old architecture, and Antidote-LevelDB for the new implementation proposed in this thesis.

Antidote [7] offers TCC over a geo-replicated data store, which relies on CRDTs for convergence. Antidote uses op-based CRDTs, and implements a key-value interface to access the data store. It is written in Erlang and features scalable, conflict-free implementations of transactions, by providing consistent, stable snapshots and atomic multi-CRDT updates [8]. This type of transactions guarantee Causal Consistency.

Antidote was built as part of the SyncFree Project[23], funded by the European Union 7th Research Framework Programme, ICT call 10.

Antidote incorporates cutting-edge research results on distributed systems and modern data stores, and is the first to implement Cure[8], providing:

- Transactional Causal+ Consistency (causal consistency with convergent conflict handling)
- Replication across DCs
- Horizontal scalability by partitioning of the key space
- Full Operation-based CRDT support
- Partial Replication

To provide fast parallel access to different data, data is sharded among the servers within a cluster using consistent hashing. Consistent hashing is a special kind of hashing that guarantees that when a hash table is resized, only K/n keys need to be remapped on average, where K is the total number of keys, and n is the number of slots in the table.

In contrast with most traditional hash tables, a change in the number of available slots causes nearly all keys to be remapped. and organized in a ring. A read/write request is served by the server hosting a copy of the data.

A transaction that reads/writes multiple objects contacts only those servers that have the objects accessed by the transaction. This master-less design allows serving of requests even when some servers fail.

3.1 Antidote-DL architecture

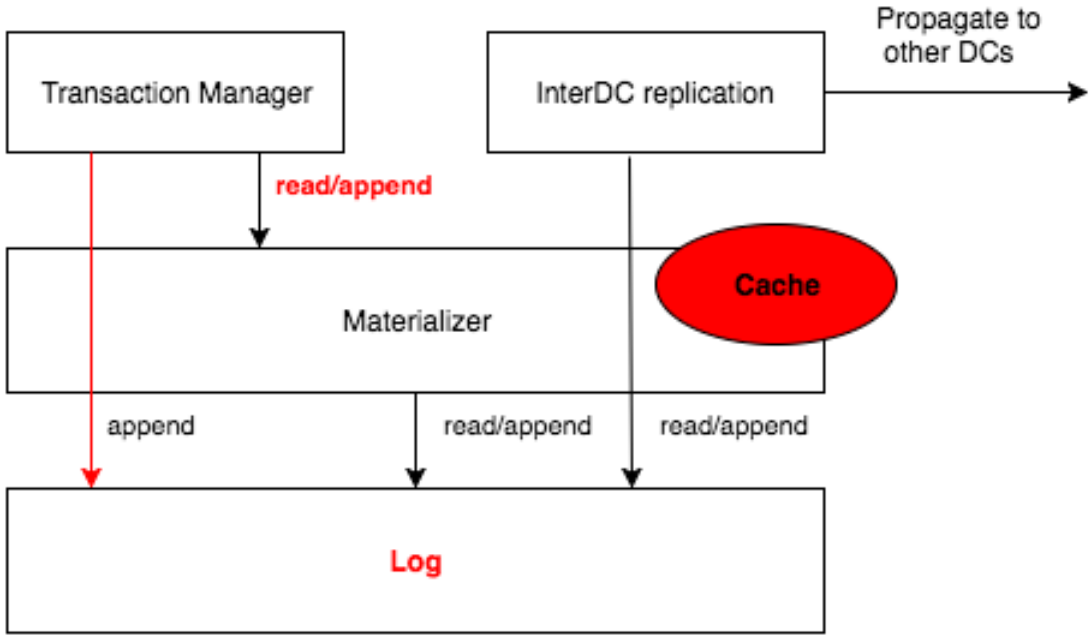


Fig. 3.1: Antidote-DL Architecture

The transaction manager module is responsible for executing all transactions in a DC. A client can contact any node in a DC and start a transaction coordinator (TC). The client then issues update and read operations via that TC.

The InterDC replication module is in charge of propagating all transactions to other DCs. Together with the transaction manager they guarantee that the transactions are causally consistent, while the use of CRDTs permits the mergeability of them.

The materializer layer is the module in charge of handling all high-level operations. Its responsibility is to generate snapshots of an object by applying operations and to reduce latency by avoiding access to the log on each read operation. To achieve these goals, each partition runs a materializer process with an in-memory cache, represented in the form of a key-value store mapping the key to a CRDT replica.

The log layer constitutes the foundation of the system architecture. Antidote uses a log-based backend to provide fast and fault-tolerant write access and efficient management of multi-versioning for CRDT objects.

As we can see in Figure 3.1, all layers either read, write or both, to the log. In this work we will focus on the Log layer as well as the cache in the materializer.

3.2 Supporting multiple versions and snapshots

As Antidote is a key-value store where objects are multi-versioned op-based CRDTs under causal consistency, we need to keep different (possibly concurrent) versions of objects, and here is where the cache and log come into action.

Antidote-DL uses Erlang's disk_log [24] as an append only log-based backend (log layer in Figure 3.1), to provide fast and fault-tolerant storage for CRDT operations to recover from faults.

The other layer we will focus on is the Materializer, the module responsible for generating the correct versions of objects out of the updates issued by clients. The module is placed between the *Log* and the *Transaction Manager* modules. The goal of the *Materializer* is to compute and cache in memory, the versions required by the *Transaction Manager*.

The *materializer* uses two caches (one for operations and another one for snapshots) based on Erlang Term Storage (ETS) tables[25], to provide faster access to the most recent operations/snapshots. This cache is in place to avoid reading from the log every time, which would be inefficient.

Let's take an example. The materializer receives a read request with vector clock X. It starts off by looking in the *Snapshot cache* for the matching snapshot with the highest time less than X (if there is more than one concurrent snapshot matching the condition, anyone could be picked since they all will yield the same result), let's say Y. Now, it checks the operation cache to see if there are any operations between Y and X (it returns all ops which have a VC concurrent or larger (but not equal) than X, and smaller or equal (for all entries) than Y) that should be added to the snapshot. If there are no ops to add, the materializer returns the value associated to Y. If there are operations to apply in the interval (X-Y) (operations included here are the ones concurrent with Y and the ones that happened before X (not concurrent)), it applies them, generating a new Snapshot with time Y' (the most recent VC of all the applied operations), and returns the new value.

3.3 Limitations of Antidote-DL

We found several limitations in how Antidote-DL provides support for multiversioned objects:

Logging

Every update to a CRDT is stored in the log, for recovering after a crash, and for retrieving operations when a cache miss occurs. In Antidote-DL, if a crash occurs, the cache can be filled up by parsing the log. However, when a cache miss occurs, the functionality of looking for the missing object in the log, is not implemented.

Moreover, the log does not provide random access to its records. Accessing an operation would heavily impact performance, since we can not get all updates for a given key without scanning the full log sequentially.

Caching

- Antidote-DL assumes that active transactions read from memory.
- Another problem is how Antidote saves and retrieves operations and snapshots in/from the cache. Antidote keeps one cache for operations and another one for Snapshots, indexed by object key. Each table keeps a list of operations/snapshots as value. The list of operations in cache must be kept small, since it needs to be scanned sequentially to get the desired operations/snapshots.

- To keep this list small, the GC algorithm keeps the latest N (set at startup) operations per key. When this number is reached, it keeps the newest L (also set at startup) in memory and discards the rest, so N < L can be assumed. A new transaction may need to read a garbage collected version from the cache, making the transaction time increase since that version should be looked up in disk.
- GC is triggered by reads (in the Snapshot table) and writes (operations table) and is synchronous. This means that the execution of each operation that triggers the GC, will be stopped for some time until the GC finishes.
- If an operation that is not in the cache is required to construct a snapshot, the transaction aborts, instead of looking it up in the log.

3.4 Benchmarks

In this section we will show the performance of Antidote-DL, empirically validating the limitations presented in the previous section.

3.4.1 Basho Bench

To run these benchmarks we will use Basho Bench[26], a benchmarking tool created to conduct accurate and repeatable performance and stress tests on Erlang projects.

Basho Bench focuses on two performance metrics:

- **Throughput:** number of operations performed in a timeframe, captured in aggregate across all operation types (reads and writes).
- **Latency:** time to complete single operations, captured in quantiles per-operation, discriminated by operation type.

3.4.2 Configurable parameters in Antidote

Antidote has setup a special driver to work with Basho Bench, and some of the parameters it receives include:

- **Duration:** the duration of the experiment in minutes.
- **Key generator:** the distribution of the key space. For experimentation, we use integers as keys for Antidote, so this parameter indicates from which distribution this integers come from.
- **Operations:** the type of operation/s to be performed and frequency. In other words, we define how many reads and writes to do per transaction. For example saying 80 reads and 20 writes per transaction, will issue transactions with a single read or write operation in an 80/20 frequency. We can also batch those operations into single transactions, which will end up issuing transactions with 80 reads and 20 writes (in random order) each.
- **Antidote Types:** the types of CRDTs to be used in the benchmark.

Regarding the log, Antidote offers three configuration parameters:

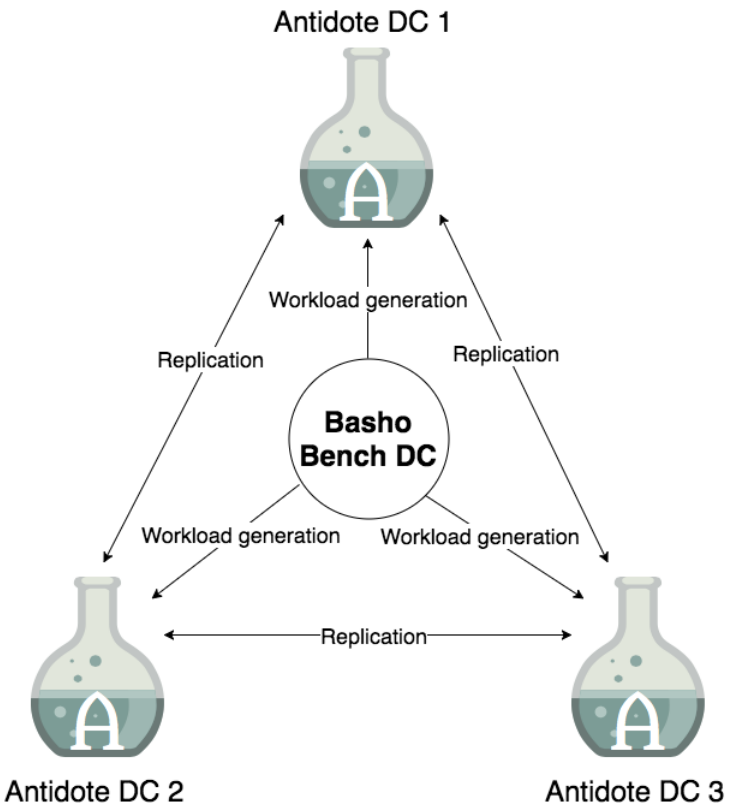
- **enable_logging:** If set to true, writes to disk are enabled. If not, updates are non-recoverable after shutting down Antidote, or under failure.
- **sync_log:** If set to true, local transactions will be stored on the log synchronously, i.e., when the reply is sent, the updates are guaranteed to be stored on disk. If not, then the updates will be eventually stored, so there is no guarantee that they are stored durably on disk on failure.
- **recover_from_log:** If set to true, on Antidote start, the ETS cache will be filled with operations and snapshots recovered from the log. If not, the cache will be empty.

3.4.3 Grid5000

All benchmarks presented in this section will be run in a Multi-DC setting using dedicated servers in Grid5000[27], a large-scale and versatile testbed for experiment-driven research located in France.

Each server runs 2 Intel Xeon E5520 CPUs with 4 cores/CPU, 24GB RAM, and 119GB SSDs for storage. Nodes are connected through shared 10Gbps switches with average round trip latencies of around 0.5ms. NTP is run between all machines to keep physical clocks synchronized.

Multi-DC setting



We have 3 servers, each running one Antidote DC. Each server is connected to the others using ZeroMQ[28] sockets running TCP, with each node connecting to all other nodes to avoid any centralization bottlenecks.

To generate the workload for all three Antidote DCs, we have one server dedicated to Basho Bench. In preliminary testing we found that with 8 Basho Bench instances, for write intensive scenarios, and 12 instances for read intensive, we can achieve a CPU usage of approximately 85-95% on each Antidote server.

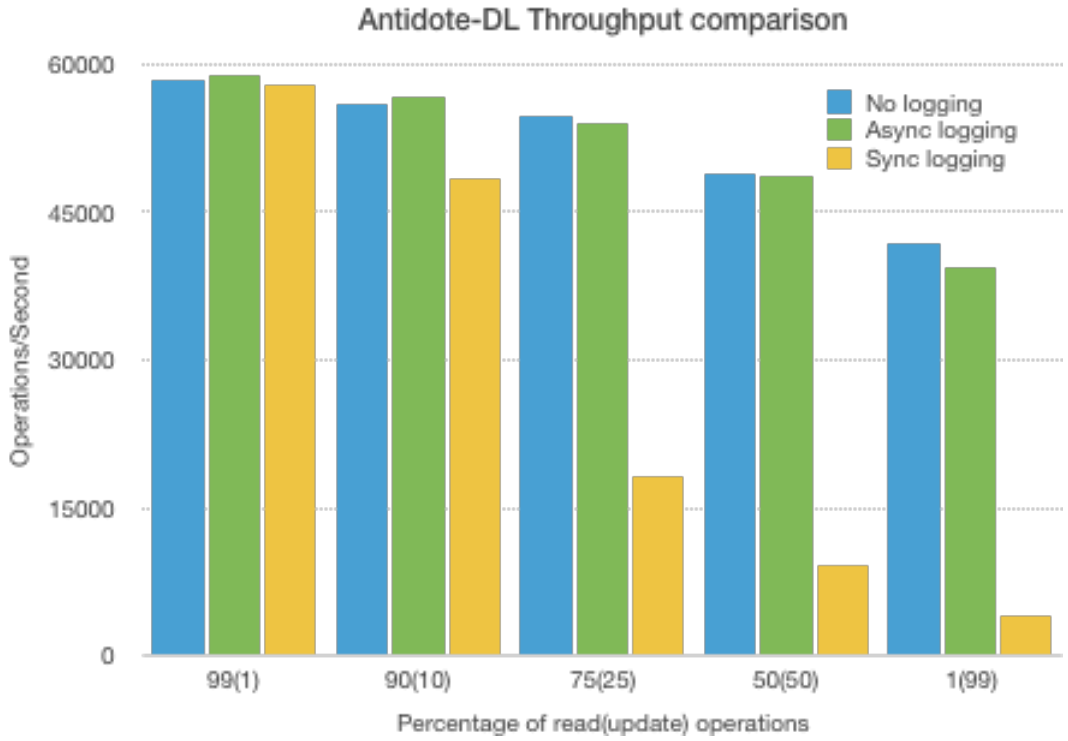
All experiments were run 5 times and the average of the runs is presented.

No logging vs Async logging vs Sync logging Benchmarks

For this first benchmark, we will use the following percentages of read(update) operations: 99(1), 90(10), 75(25), 50(50) and 1(99). For each percentage, we will run a single Antidote cluster for 10 minutes, to compare how the different logging options (no logging, async logging and sync logging), impact throughput, read and append latency.

The benchmarks are performed on a counter CRDT with transactions containing a single read, increment or decrement by one operation. The key space was generated by a Pareto distribution of 100 000 values.

Throughput



The first key aspect to point out, is that having no logging at all, or making async writes to the log, show practically the same performance for all workloads. This shows a good performance of disk_log on how efficient it can handle async writes.

Having a workload composed of 99% reads, shows practically no difference between the logging setups. Slightly changing the workload to 90% reads, the sync setup already

shows a downgrade in performance of nearly 1.2x.

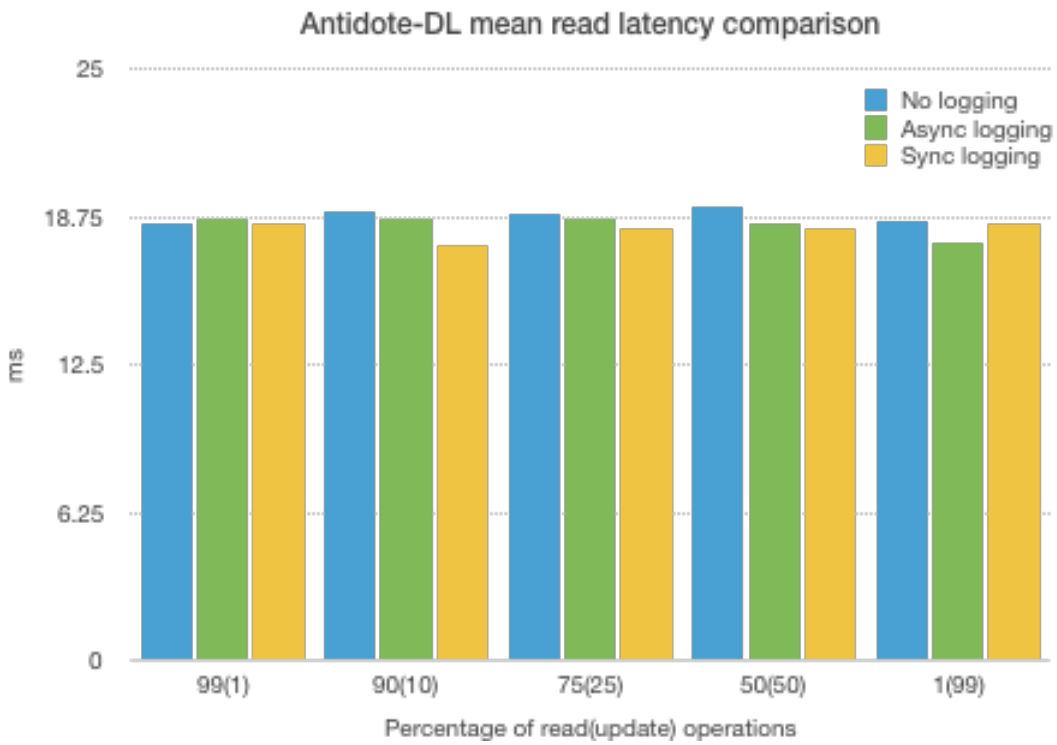
This trend continues as we shift into workloads composed mainly of writes, reaching its maximum peak in 1(99), where the difference between no logging and sync logging is 10x.

From this first benchmark we can take two important points to analyze why this downgrade in performance occur:

- Making Antidote fully fault-tolerant by syncing the log on every write, is really expensive.
- As we move to write intensive scenarios, not only the log is impacting throughput. ETS tables can be configured to allow read concurrency, write concurrency or both. Since Antidote is designed to work better on read intensive scenarios, it configures ETS tables to only allow read concurrency. ETS documentation states that allowing both can clearly impact performance if the access pattern is not even. This is the reason why write concurrency is disabled, meaning that an operation that mutates (writes to) the table, obtains exclusive access, blocking any concurrent access of the same table until finished.

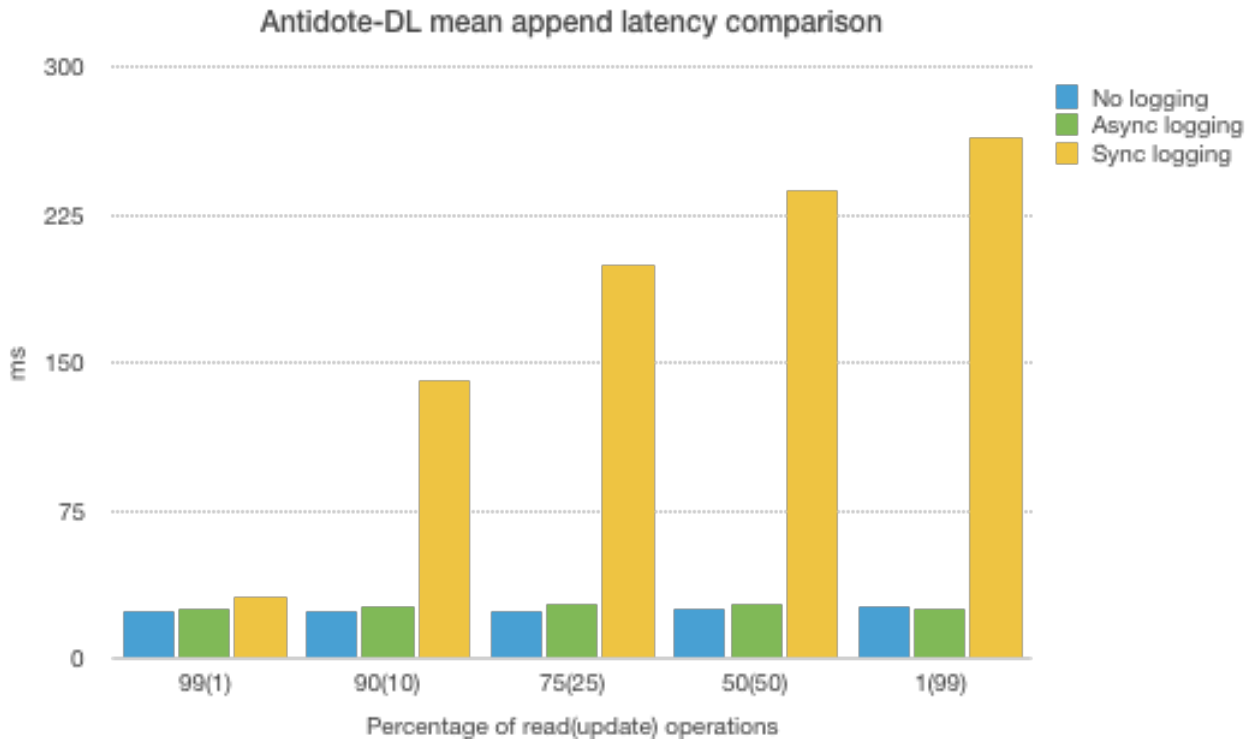
Not allowing parallelization of writes, plus syncing the log for fault tolerance, is impacting heavily on the performance of write intensive scenarios.

Mean read latency



Read latency keeps stable between all workloads since there are only some nanoseconds of difference between them. This shows in practice that ETS tables are very efficient on how they handle reads.

Mean write latency



Write latency shows clearly the difference between syncing or not the log on every write. Latency keeps stable between all workloads considering no logging and async logging. Comparing sync logging to either of the two setups, the difference grows from 1.5x more latency in 99(1) to an incredible 10x in 1(99).

This again shows the impact of syncing the log and the limitation set to ETS tables so no concurrent writes are allowed.

3.4.4 Conclusions

Throughout these benchmarks we showed how Antidote-DL throughput and write latency are heavily impacted if we want to provide recoverable updates under failures.

We also showed that Antidote's read latency is very low under all circumstances, while writes stand in the opposite direction, impacting directly on throughput.

This benchmark supports all our suspicions, and show that a more efficient solution to handle sync writes and provide fault tolerance, would boost Antidotes performance.

4. ANTIDOTE-LEVELDB APPROACH

After analyzing the current limitations in Antidote, we realized that its design is inefficient. The cache in the materializer provides fast reads, but slow updates, where the log provides fast updates but slow reads. We propose a mixture of both, which enables both fast reads and recoverable updates.

Antidote-LevelDB approach is based on LSM-Trees and its implementation in LevelDB. Adapting LevelDB to work with Antidote was the main challenge of this work. By achieving this, both fault tolerance and good performance can theoretically be obtained, since LevelDB is persistent, stores its keys in order (favoring searching) and allows indexed access to records. Another proposed challenge for this thesis was to decouple Antidote from the backend module storing operations and snapshots. To fulfill this, the Antidote Backend Module was created as an abstraction layer with Antidote-LevelDB acting as its first implementation.

4.1 Log-Structured Merge-Tree (LSM-Tree)

The LSM-Tree[17] is a data structure designed for having performance characteristics that make it great for providing access to files such as transactional log data. LSM-Trees store key-value pairs sorted by key and cascade data over time from smaller, higher performing (but more expensive) stores to larger less performant (and less expensive) stores.

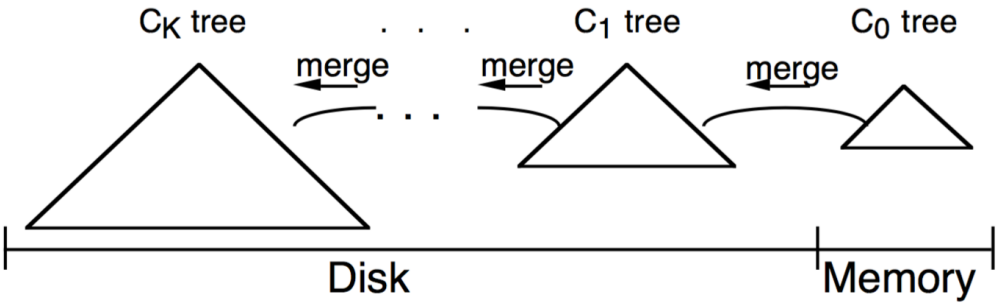


Fig. 4.1: LSM-Tree architecture

The typical architecture of an LSM-Tree is reflected in Figure 4.1. Here we can see two main components: a memory component and a disk component.

The *memory component* (C_0) is a sorted data structure residing in main memory. It contains the latest updates performed on the key-value store. Its size is typically small (can be configured by the user). When it fills up, it is replaced by a new, empty component. The old memory component is then flushed to level 1 (C_1) of the LSM disk component, where it is merged with the existing (C_1) component, generating a new one.

The *disk component* is structured into multiple levels (C_1 ... C_k) with increasing sizes. Amount of levels, maximum sizes of each level and the policy on how to proceed when the last level fills up, are implementation decisions taken by developers using this data structure.

When a level is filled up, data is compressed and sent to the next level, where it is merged with the existing component, generating a new one.

It is important to note that when a level fills up, the merge function merges the existing component in a level with the one from a higher level. This flow of inserting sorted keys, and merging sorted levels is reminiscent of merge sort.

When a *key is updated*, it is added to the memory component with the new value. The previous version of the key will be looked up (asynchronously) in all components and marked as deleted. Now, when a read for that key occurs, the new value will be the only one returned.

From the previous point we can see that in the whole structure we could have the same key repeated a couple of times, so this is where *compaction* comes into play. When a level is filled up, the merge function is applied and keys marked as deleted (because they where completely deleted, or they are an old version of an updated key), are effectively deleted.

Another important component in this structure, not present in Figure 4.1, is the *commit log*. This component is a file residing on disk which has the responsibility to temporarily store the updates that are made to level C_0 (in small batches), if the application requires that the data is not lost in case of a failure. All updates performed on level C_0 are also appended to the commit log. Typically, the size of the commit log is kept small in order to provide fast recovery in case the operations need to be replayed to recover from a failure. This log is erased completely when the C_0 level is sent to level C_1, since level C_1 is already persistent.

Let's now analyze why reads and writes are efficient.

Every *write* happens in the memory component and is stored in a small log for recovery. This is extremely fast, since writing to memory is fast, and the log is kept very small. If the memory component was filled up with a write, the process of merging and compacting levels, happens in an asynchronous thread, not impacting write performance.

Reads are a bit trickier. To look up for a key, binary search has to be done on every level. Even though it is efficient to do this, it can be further optimized using bloom filters. We are not going into detail of how bloom filters are used, but essentially each level has a bloom filter associated to it, indicating if a key is present in that level or not. Taking this into account, reading a key, is now simplified to check each level (starting at C_0), checking the presence of the key in the bloom filter, and when it is found it can be accessed directly.

Having analyzed LSM-Trees, and their potential to act as the underlying data store for this new approach, we proposed ourselves with the thesis "LSM-trees are a good approach for storing multiversion objects of CRDTs", which we will demonstrate true in the following sections.

4.2 LevelDB

LSM-Tree based LevelDB[18] is a fast persistent key-value storage library written in C++ at Google. Since Antidote is an Erlang project, we will be working with the Erlang bindings (a simple translation layer between programming languages) developed by Basho, called ElevelDB[29].

The structure under LevelDB is an LSM-Tree, storing data in 6 levels. The first one is in memory, and all the others on disk. Therefore, this design serves as a caching and logging system at the same time.

ElevelDB sorts its keys using a user-defined comparator function. We will have to look into how we design this comparator function and the keys it will sort.

Another nice feature of LevelDB is that it provides a fold method. This method receives the starting key from which key-value pairs are retrieved from the DB one by one in order, until a break signal is sent by the caller.

4.3 Antidote's new architecture with LevelDB

In Figure 4.2 we observe the new architecture, and in green the changes performed in comparison with Figure 3.1. With this new model, the cache in the materializer and log layer are replaced with the new **Antidote backend module** layer, which caches and persists operations and snapshots. The transaction manager no longer interacts with the materializer to persist transactions, instead doing it directly to the new module. The inter DC replication module remains untouched.

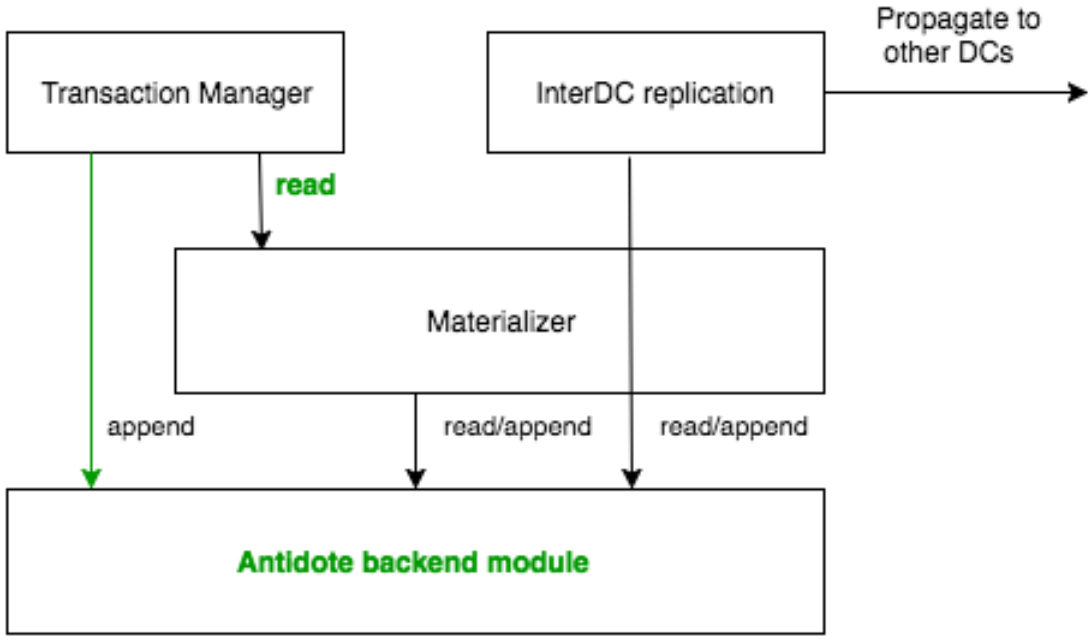


Fig. 4.2: New Antidote Architecture

4.4 Antidote backend module

Antidote's code base is strongly coupled, impacting on developers performance, making it difficult to introduce changes easily. After introducing a large change, Antidote developers can not easily unit test the code in the Materializer, since its logic is strongly coupled to ETS tables. To take a first step into modularizing Antidote, we developed this new module as a standalone Erlang application.

To explain the API exposed by the new module, we will first consider the following data types:

- **BackendModule_Type** an enum which indicates the type of the backend module.

- **BackendModule** a tuple containing a BackendModule_Type and references to all the objects needed to use the backend data store. In the case of LevelDB, this object contains the LevelDB identifier plus a reference to the LevelDB object.
- **Key** a String representing the Key used in Antidote to reference an object.
- **CommitTime** a VC indicating the commit time of an operation/snapshot.
- A **Snapshot** is a triple *(k, VC, v)* where *v* is the value at time *VC* of the object with key *k*.
- An **Operation** operation is a triple *(k, VC, op)* where *op* is the operation applied to object with key *k* at time *VC*.

Having seen the data types used, we now present the API of the module itself. This API is not coupled to LevelDB, so it can easily be extended in the future with another backend datastore, in a very simple way and no code in Antidote needs to be changed.

- **new(BackendModule_Type)** Given a BackendModule_Type, returns a new BackendModule.
- **get_snapshot(BackendModule, Key, CommitTime)** Retrieves the most recent snapshot (or returns a not found message) with commit time less than **CommitTime**, for the given **Key** in the **BackendModule** data store in case of concurrent snapshots found, the last saved is the one returned.
- **put_snapshot(BackendModule Key, Snapshot)** Saves the given **Snapshot** of the object identified with **Key** to the **BackendModule** data store.
- **get_ops(BackendModule Key, VCFrom, VCTo)** Retrieves a list of operations (empty if none is found) in the range **[VCFrom, VCTo)**, for the given **Key** in the **BackendModule** data store.
- **put_op(BackendModule, Key, VC, Operation)** Saves the given **Operation** of the object identified with **Key** to the **BackendModule** data store.

The idea of this module is driven by Antidote needs, and that is why it is called Antidote backend module. It is interesting to note that the API does not use anything that is Antidote specific, so it can be of use in any other scenario where the API is a match for a multiversion data store.

4.4.1 Module example usage

To see how this module works, let's take the same example used in Section 3.2. The materializer receives a read request for an Antidote CRDT (with id: ID) and vector clock X, performing the following steps:

1. The materializer calls the backend module to get the most recent snapshot matching time X: **get_snapshot(BackendModule, ID, X)**, retrieving Snapshot S, with commit time Y.

2. Now the materializer has to check if there are any operations to apply to S. It does so by calling: **get_ops(BackendModule, ID, Y, X)**, retrieving a list of operations L.

3. If L is empty, the value of S is returned.

   If not, the materializer applies all operations in L to S, generating a new Snapshot S'. Finally the materializer calls **put_snapshot(BackendModule, ID, S')** to store the new snapshot, and returns S' to the caller.

4.5 Key selection for ElevelDB

In order to allow the *merge* function of the LSM-Tree running under ElevelDB to work, we need to create a unique key per operation/snapshot, that can be sorted by a comparator function we will implement.

We have available the Antidote key and a vector clock of the time each operation/snapshot was committed, so let's see what we can do with them.

If we want to find a specific snapshot for example, the new algorithm should just perform a single lookup. The tricky scenario is how to efficiently implement the **get_ops** method described previously. This new algorithm using LevelDB should leverage that keys are sorted, to make this efficient searching algorithm.

As keys have to be unique, a multilevel key has to be implemented. Lookups will be done by Antidote object ID, so it makes sense to have it as the most significant value.

Let's now see what we can append to the key after the Antidote object ID.

4.5.1 First approach

This first approach to find a suitable key to use in ElevelDB was composed of these three parts, presented in the same order as how the key is created (more significant to less significant):

- **antidote_key** each object in Antidote has an **antidote_key** associated to it. Using this value as the first part of our key, implies we are grouping together all values of the same object.

- **vector clock** for operations/snapshots that are associated to the same **antidote_key**, we will use the VC of its commit time to differentiate them. We are sorting this VC by DC id.

- **is_op** as we are storing together both operations and snapshots, we are using this boolean to indicate if the value associated to this key, stores an operation or a snapshot.

Given a range [VCFrom, VCTo) (where VCFrom $\leq$ VCTo) we want all operations that have a VC that matches the condition: **(VC $\not\leq$ VCFrom) AND (VC $<$ VCTo)**. In other words, we want all operations which have a VC concurrent or larger (but not equal) than VCFrom, and smaller or equal (for all entries) than VCTo.

To accomplish this, we chose to sort the VC by DC name, so we could implement an efficient stop condition for the fold method previously explained. We sorted the VCFrom

and VCTo clocks in the same way, and decided to stop the fold when the time for a DC was bigger than the time of the same DC in VCTo.

We implemented this idea into the new module, and refactored Antidote to work with it. To test the correctness of it, we used PropEr[30]: a tool for automated, semi-random, property-based testing of Erlang programs.

We basically told PropEr what a valid VC is and how to generate operations in Antidote. Then we asked PropEr to validate that for a valid pair of VCs (VCFrom has to be $<$ VCTo) calling the get_ops method resulted in all operations matching (VC $\not\leq$ VCFrom) AND (VC $<$ VCTo).

We run this validation and PropEr yielded errors after approximately 10 000 tests. PropEr returned this execution with the case that failed the validation, and we quickly identified the problem to be in the stop condition of the fold.

This is a simplified example of why this approach is failing:

- Operation 1: [{dc1,1},{dc2,1}]
- Operation 2: [{dc1,1},{dc2,2}]
- Operation 3: [{dc1,2},{dc2,1}]
- Operation 4: [{dc1,3},{dc2,1}]

Searching for all the operations in the range [{dc1,2},{dc2,1}] and [{dc1,3},{dc2,2}], retrieves only operation 4, when the correct answer is operation 4 and operation 2.

This error occurs because the algorithm will search for the first operation that has dc1 with a value greater or equal to 2, finding operation 3. We are folding from older to newer operations, so the next operation will be operation 4, which will be returned. There are no further operations to check for this key, so the fold will end with only operation 4 returned.

The problem here resides in the fact that sorting the VC and comparing the value for each DC by its own, is not a sufficient condition to make a fold work. Obviously doing a sequential scan will always return the correct value, but that would not be as efficient as only accessing a small portion of the key space.

Adding more conditions to the fold stop condition did not help, since the problem resided in how the key was selected.

We ended up concluding that with this key and sorting algorithm the only way of retrieving the correct operations, was to iterate the whole key space as in Antidote-DL, which is not acceptable.

Having failed with the first approach, we had to think of a new idea that did not have to iterate through all the keys to find a specific antidote key, since that "solution" is exactly what we want to change.

4.5.2 Second approach

The main problem with the first solution was that we did not have an easy way to stop the fold when looking up a set of keys. Having discarded all ideas to drastically change the algorithm and architecture, we focused on understanding the weakness of the first approach, designing a new algorithm based on the maximum value seen in the VC.

We will consider the **max time of a VC**, the maximum integer value found in all DCs of a VC. For example if we take [{dc1,3},{dc2,8},{dc3,7}], this value will be 8.

Instead of looking at the whole VC to make the decision if it matches the condition, we can only look at the biggest value for all DCs in the VC.

Taking this into account, the new algorithm looks like this:

Algorithm 1

```
1: procedure GET_OPERATIONS(DB, KEY, VCFROM, VCTO)
2:     operationsList ← new empty list
3:     MinTimeToSearch ← min time in both VCFrom and VCTo
4:     iterator ← start iterating from the newest key matching the prefix {Key,
   maxValue(VCTo)}
5:     key, value ← iterator.next()
6:     while MinTimeToSearch ≤ max time of VC in current key do
7:         if vcInRange(key.VC, VCFrom, VCTo) then
8:             operationsList.add(value)
9:         key, value ← iterator.next()
       return operationsList
```

To simplify the explanation of this algorithm, we omitted minor details. Nevertheless, one detail that should not be omitted is that line 7 is calling an auxiliary method which validates the condition **(VC $\not\leq$ VCFrom) AND (VC $<$ VCTo)** using the full VCs. This means that for a first check (line 6) we compare the MinTimeToSearch integer with the max value in the current VC, and while that remains true, we check the whole VC to see if it should be included in the answer.

With this new approach, the composition of the key for LevelDB is (more significant to less significant):

- **antidote_key** same as in the first approach.
- **max_time_in_vc** the maximum time of all DCs in the VC.
- **vc_hash** we may have keys that share the same antidote_key and **max_time_in_vc**, but come from a different VC. To avoid comparing all the VC to see if they are the same, we added this hash of the VC to the key, which makes comparison much faster.
- **is_op** we are storing together both operations and snapshots, so we are using this boolean to indicate if the value associated to this key stores an operation or a snapshot.
- **vector clock** the VC (not sorted) associated to this operation/snapshot.

With this new key setup, the key still has the Antidote object ID as the most significant part of it. Following the Antidote object ID, we have the **max_time_in_vc** which will determine how to perform a lookup of keys.

Example

As en example to see how this second approach works, let's analyze the failed case of the first approach. We have:

- Operation 1: [{dc1,1},{dc2,1}], with max_time_in_vc = 2
- Operation 2: [{dc1,1},{dc2,2}], with max_time_in_vc = 2
- Operation 3: [{dc1,2},{dc2,1}], with max_time_in_vc = 2
- Operation 4: [{dc1,3},{dc2,1}], with max_time_in_vc = 3

Searching for all the operations in the range [{dc1,2},{dc2,1}] and [{dc1,3},{dc2,2}], implies we want to check all operations that have max_time_in_vc between 1 and 3, which in this case is all four of them.

For each operation the condition (VC $\not\leq$ VCFrom) AND (VC $<$ VCTo) will be tested, returning operations 2 and 4, which is the correct result.

Correctness

Given a range of VCs, from which we want to retrieve all operations that fall in it, we want to prove that only the correct operations are returned, and none are left out. This also implies that we are proving we have a safe condition to stop the fold in the get_ops method.

Proof. Given two VCs, *VCFrom* and *VCto*, and operation *op* with op.VC = *VCop*, we want to return *op* in the result list, if and only if (VCop $\not\leq$ VCFrom) AND (VCop $<$ VCTo) is true.

We take *MinTimeToSearch*, as the minimum time in both *VCFrom* and *VCTo* and *MaxTimeInVc* as the maximum time in *VCop*.

Operations are sorted by Antidote key and then by max value in VC. This implies that all operations with a *maxValue(VCop)* greater than *maxValue(VCTo)* would not satisfy the condition (VCop $\leq$ VCTo), so it is safe to start folding operations from newest to oldest key matching the prefix {Key, maxValue(VCTo)}. This leaves 3 cases to analyze:

- **MinTimeToSearch** is bigger than *MaxTimeInVc*: *op* should not be returned, since (VCop $\not\leq$ VCFrom) is false. Moreover, we do not need to see any other operations, since having them sorted in ascending order of *MaxTimeInVc*, guarantees that following operations would not fall in the required range either.
- **MinTimeToSearch** is smaller than *MaxTimeInVc* and (VCop $\not\leq$ VCFrom) AND (VCop $<$ VCTo) is true: *op* was found and will be returned since all conditions are met.
- **MinTimeToSearch** is smaller than *MaxTimeInVc* and (VCop $\not\leq$ VCFrom) AND (VCop $<$ VCTo) is false: even though *op* is not returned, we still have to continue looking, since there may be other operations with the same *MaxTimeInVc*, that fall in the range.

Since this are all possible cases for *VCop*, we can guarantee that *op* will only be returned when all conditions are met, and no operations are left out.

Testing

The new Antidote backend module has several unit tests to validate this approach, by testing both general and corner cases, and internal helper methods.

Several unit tests were added to avoid regression of the failing cases from the first approach, ending up with a robust test suite that provides 100% code coverage of the new module.

Configuring Antidote-DL or Antidote-LevelDB is done using a configuration parameter on Antidote startup. This implies the code coexists perfectly well and can be easily triggered for testing.

To test this new approach in Antidote, we refactored all existing unit tests in Antidote, to run with both Antidote-DL and Antidote-LevelDB setup.

Once all unit tests (both module specific and Antidote) passed, we went again to PropEr and made an exhaustive testing of several runs of up to 300 000 cases each, which were all successful.

Recovery

LevelDB is out of the box a persistent database, and can be configured to synchronous or asynchronous on every write. This implies that if we use the sync configuration, when a crash occurs, all updates should be correctly saved.

After a crash, the first level cache of the last operations will be empty when Antidote-LevelDB comes back up, but no log has to be replayed in order to start serving requests again, Antidote-LevelDB can start up immediately. When a new read/write request is sent, Antidote-LevelDB will work as described before, using the same algorithms.

This implies that Antidote-LevelDB is persistent without adding any additional code (than the algorithms described before), which has several advantages:

- Antidote-DL uses two ETS caches and a disk log instance. This implies that for each write, Antidote-DL has to write both one ETS table and the log. For reads, it has to lookup the ETS table and maybe consult the log if it does not found an operation/snapshot.

  On the other hand, Antidote-LevelDB uses only one LevelDB instance. This implies all reads and writes are handled via the same code, and all writes are only persisted once, which should prove more efficient.

  Having only one LevelDB instance in Antidote-LevelDB, could become a bottleneck, so this will be benchmarked in the following chapter to make a fair practical comparison.

- Having only one algorithm serving both needs, means it is less error prone, and since no special recovery logic is added, it is easier to reason for developers.

Code implications of this approach

The new key comparator in ElevelDB is comprised of approximately 50 new LOC written in C++.

The new backend module comprises approximately 200 new LOC written in a Standalone Erlang Application.

In order to achieve the new architecture in Antidote, we had to refactor approximately 500 LOC, mainly in the Materializer layer.

5. RESULTS

In this chapter we present the results that empirically validate the improvements of Antidote-LevelDB vs Antidote-DL.

We will use the same terminology as previously defined in Section 3.4. If not stated otherwise, the setup for all the benchmarks will also be the same as the Multi-DC setting described in Section 3.4.

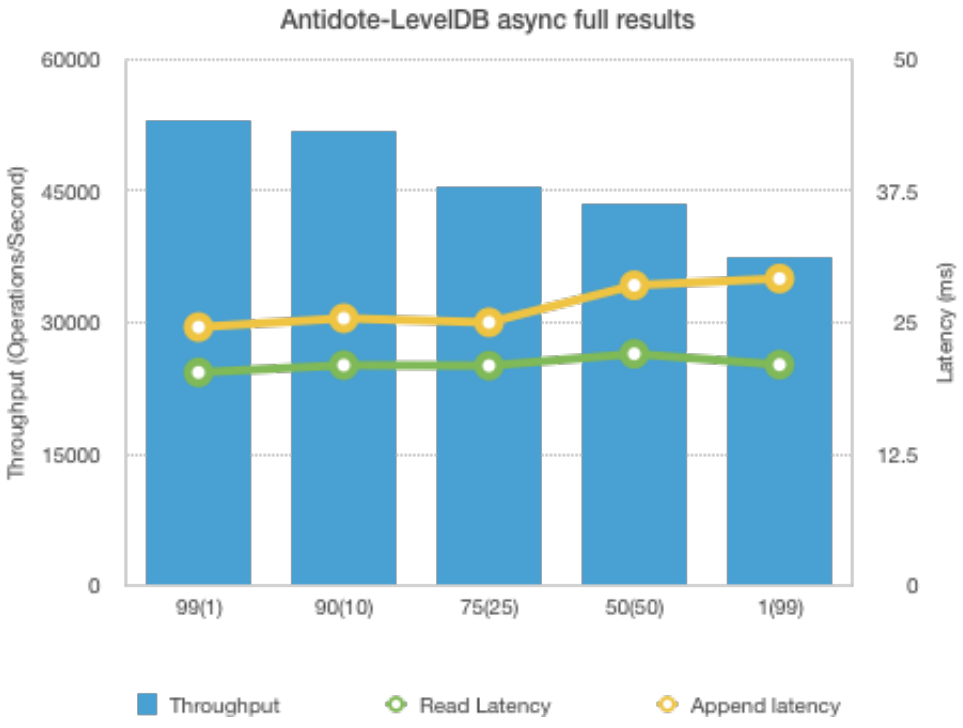
As stated previously, a DB that does not provide efficient fault tolerant guarantees, is not useful. That is the reason why, for this benchmarks, we will compare the sync logging versions of both Antidote-DL and Antidote-LevelDB.

One important thing to take into account, is that Antidote-DL does not store snapshots on the log, so they need to be recalculated in case of failures or if they are deleted from the cache. This also implies a performance difference in its favor since Antidote-LevelDB stores snapshots persistently in the new backend module, making them available for access at all times.

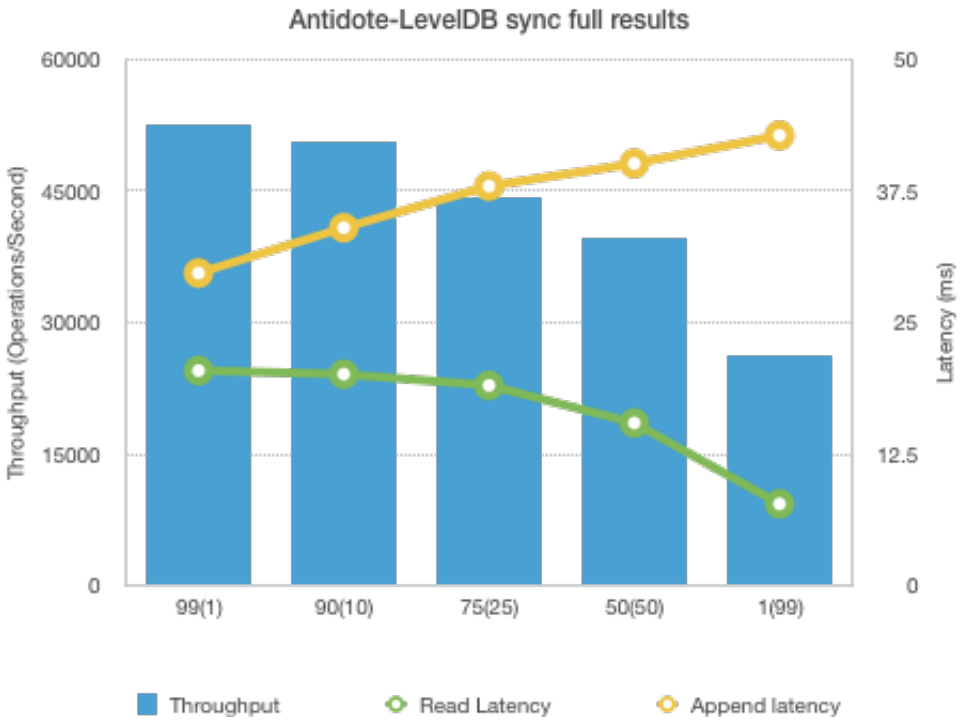
5.1 Antidote-LevelDB overview benchmarks

To get a general overview of Antidote-LevelDB performance, we run the same benchmark as for Antidote-DL (described in Section 3.4.3), for both async and sync write configurations.

5.1.1 Async writes



5.1.2 Sync writes



As expected, having LevelDB sync on every write, decreases throughput and increases write latency. Read latency shows a clear trend to get lower with sync writes, but it is out of the scope of this work to further explore this phenomenon that is left for future work.

Even though the performance decreases when we switch to sync operations, in 99(1), 90(10) and 75(25) we can observe a downgrade of 3%, which shows perfectly well how Level-DB handles this workloads. In 50(50) the loss is about 25% and in 1(99) is of approximately 50%, which is a big loss, but this number shows an outstanding stability of LevelDB to handle workloads composed mainly of writes.

Regarding write latency, we can see it grows an approximate 40% from 99(1) to 1(99) in the sync configuration, which is expected.

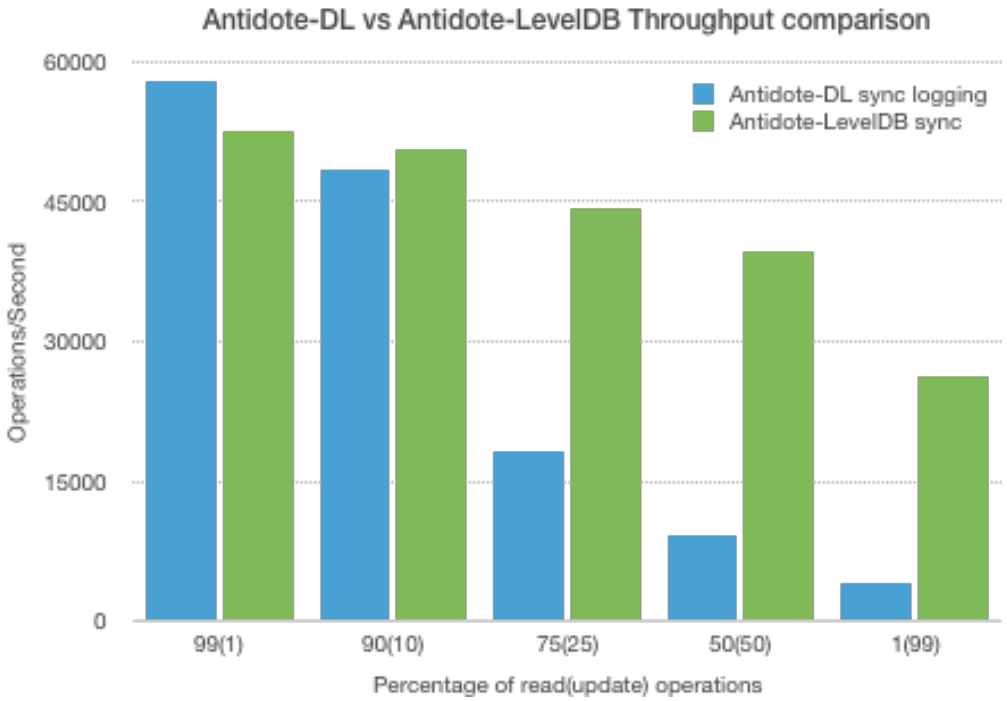
The surprising figure in this benchmarks is that read latency decreases considerably as writes grow in the sync configuration.

This benchmark validates most of the assumptions we had, which will be detailed in following analysis.

5.2 Antidote-DL sync vs Antidote-LevelDB sync

In this section we will compare Antidote-DL sync vs Antidote-LevelDB sync.

5.2.1 Throughput



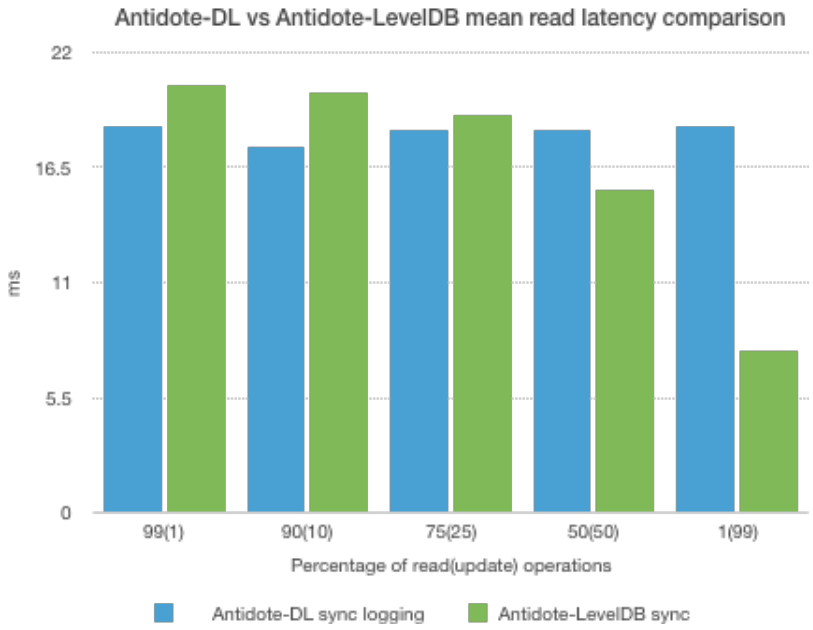
This comparison validates all the assumptions we had. 99(1) and 90(10) are scenarios where both solutions show similar results which is expected since Antidote-DL is designed for read intensive scenarios. 75(25), 50(50) and 1(99) show how Antidote-LevelDB provides from 3.2x throughput in 75(25) to an overwhelming 6.2x in 1(99).

It is also interesting to note that Antidote-DL has a downgrade in performance that can be compared to a rational function like $f(x) = 1/x$, while Antidote-LevelDB has a downgrade comparable to a linear function of the kind $g(x) = -ax$, for some constant a. This is an interesting point to explore in detail in future work.

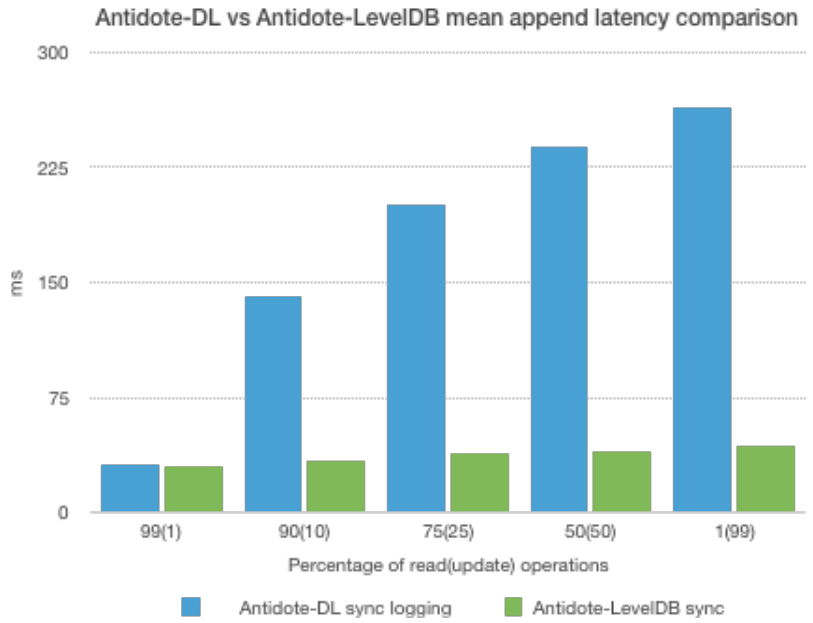
As detailed before, on write intensive scenarios, Antidote-DL has to perform a lot of synchronous garbage collection in the ETS tables, has to sync every write to ETS tables and also sync the log for every write to provide fault tolerance, so it was expected to fail against Antidote-LevelDB.

Antidote-LevelDB is perfectly handling all scenarios, even when the amount of writes performed to LevelDB is greater than the ones done to disk log, since Snapshots are saved in LevelDB and not in disk log.

5.2.2 Read latency



5.2.3 Append latency



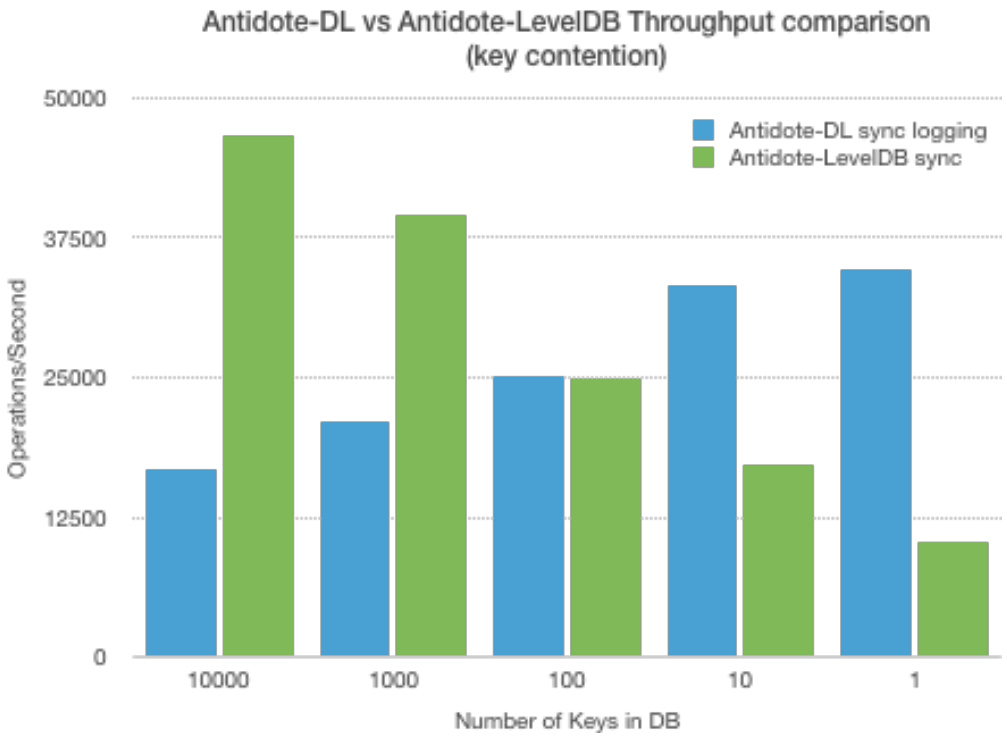
Regarding read latency, both Antidote-LevelDB and Antidote-DL show a similar outcome, and even Antidote-LevelDB shows lower latency on write intensive scenarios. This confirms that LevelDB can perfectly handle all the proposed workload scenarios.

Regarding write latency, it shows the same trend as throughput, which is reasonable since it is the main cause Antidote-LevelDB has more throughput in write intensive scenarios. In 99(1) Antidote-LevelDB shows a 1.1x better latency, while moving to 1(99) the difference is 6.25x.

5.2.4 Key contention

After performing all the experiments presented before, we wanted to see what happened if we changed the contention of the keys, since we had the hypothesis that as the number of keys decrease, the throughput should also decrease since a lot of key contention is taking place.

To do this, we decided to go with a 75(25) read/write ratio to have a good balance between the amounts of reads and writes performed. We varied the amount of keys from 1 to 10000, finding this results:



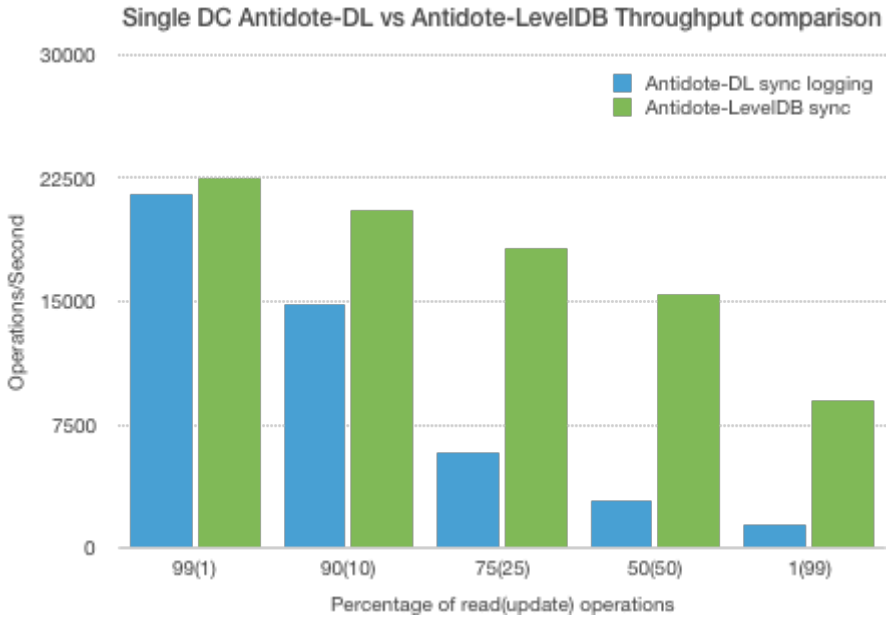
Antidote-LevelDB works as expected, while Antidote-DL increases throughput as the amount of keys decreases. This was not the behavior we expected to get from this experiment, since throughput should theoretically decrease in both implementations.

This experiment was run several times to check results and debug the code, with no further findings that can be presented in this thesis, therefore requiring future work to analyze deeply and understand this case in Antidote-DL.

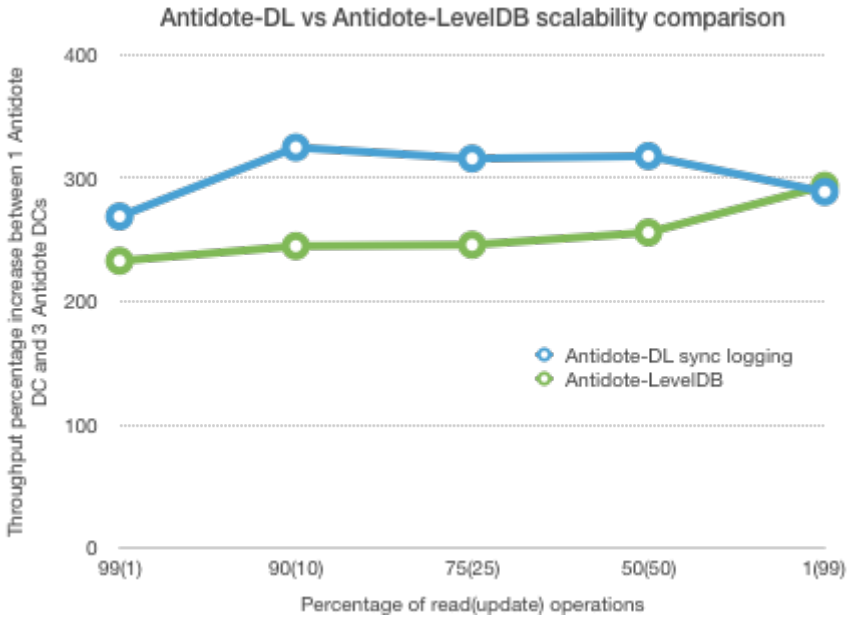
Another important aspect to point out is that the number of failed operations went from 0.09% with 10000 keys, to nearly 1% with 1 key in both implementations. We think this could be a cause of the key contention, and how Antidote manages transactions since both implementations failed with the same percentages.

5.2.5 Scalability

Since Antidote is a distributed database, we have been presenting all of our benchmarks with a Multi-DC setting of 3 Antidote DCs and one DC for Basho Bench. For this final benchmark, we want to see how Antidote-LevelDB scales compared to Antidote-DL. In order to achieve this, we run a benchmark with a single Antidote DC, keeping one separate DC for basho bench. The results are presented here:



We can observe that the trend obtained in the Multi-DC benchmarks is still observed in Single-DC settings. To better understand this numbers lets analyze how throughput changed from Single-DC to Multi-DC.



While Antidote-DL provides better scalability numbers in this benchmark, it is important to remember that it is not providing more throughput than Antidote-LevelDB.

It is also important to note that Antidote-LevelDB shows a clear increase in scalability while we switch to write intensive scenarios, while Antidote-DL is more inconsistent in its trend.

6. CONCLUSIONS

Having no work previously done in the field, in this thesis we took the challenge of analyzing Antidote-DL to design and implement the first scalable caching and logging module for multiversion geo-replicated storage. We took it even further, as we tried to show LSM-Trees are a data structure suitable for this kind of purposes. To do this, we also tackled the problem of having to sort vector clocks somehow, since they don't have a total order.

We have introduced Antidote-LevelDB, the first scalable caching and logging module for multiversion geo-replicated storage. Antidote-LevelDB provides not only better proved performance than Antidote-DL, but also empowers modularity and testing in Antidote. Both implementations can coexist perfectly in the same codebase and be triggered by changing only a configuration parameter. This modularity will help Antidote developers not only write unit tests, but also run experiments on the backend data store much easier.

We also introduced the concept and implementation of a *max value comparator* for vector clocks. This concept aims to reduce the amount of integer comparisons done while comparing vector clocks to check for concurrency. It not only reduces the complexity of algorithms, but also improves performance since a single scalar comparison is needed, instead of a linear search.

We have evaluated Antidote-LevelDB in different scenarios, showing that it presents scalability compatible with eventual consistency in all kinds of workloads, while offering better performance and append latency in write intensive workloads. Read latency is similar to Antidote-DL, which proves the efficiency of Antidote-LevelDB.

We consider Antidote-LevelDB a great first step into improving latency, throughput and modularity in Antidote.

We plan to follow up with a publication of this new approach.

7. FUTURE WORK

With the investigation and experimentation done in this work, some questions have risen that need further investigation.

Once a level is full, LevelDB uses snappy[31] to compress data to send it to lower levels. Although this library is very efficient, Antidote does not need to keep all versions for ever in LevelDB. One idea to work on the future could be implement a garbage collector in C++, to delete operations and snapshots that are older than certain time. This time could be for example the current stable time between all DCs, guaranteeing that those versions would never be accessed again.

Since LevelDB become so popular and is open source, several modifications have been made to it, ending up in variants like RocksDB[32, 33]. This DB has it is Erlang bindings in ERocksDB[34] and could be an interesting backend to try out and benchmark its performance in Antidote, comparing it against Antidote-LevelDB.

One of the achieved goals of this work was to decouple the current code in Antidote by writing all new code in a standalone module. Antidote tests were refactored to make them run with both Antidote-DL and Antidote-LevelDB backends, but no tests were refactored to make them "more unitary". Tests in Antidote today can be considered to be categorized as integration tests rather than unitary, so it would be good to refactor them, and add new ones, now that mocking the persistence layer of operations and snapshots can be easily done.

REFERENCES

[1] E. A. Brewer, "Towards robust distributed systems (abstract)", in Proceedings of the nineteenth annual acm symposium on principles of distributed computing, PODC '00 (2000), pp. 7–.

[2] S. Gilbert, and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services", SIGACT News **33**, 51–59 (2002).

[3] Y. Sovran, R. Power, M. K. Aguilera, and J. Li, "Transactional storage for geo-replicated systems", in Proceedings of the twenty-third acm symposium on operating systems principles (ACM, 2011), pp. 385–400.

[4] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Stronger semantics for low-latency geo-replicated storage", in Proceedings of the 10th usenix conference on networked systems design and implementation, nsdi'13 (2013), pp. 313–328.

[5] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual: scalable causal consistency for wide- area storage with cops.", In Symp. on Op. Sys. Principles (SOSP) Assoc. for Computing Machinery, 401–416 (Cascais, Portugal, Oct. 2011).

[6] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types", In Int. Symp. on Stabilization, Safety, and Security of Distributed Systems (SSS). vol. 6976 of Lecture Notes in Comp. Sc., Springer-Verlag, 386–400 (Grenoble, France, Oct. 2011).

[7] Sync Free, *Antidote*, `http://syncfree.github.io/antidote/`.

[8] D. D. Akkoorath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro, "Cure: strong semantics meets high availability and low latency", in Distributed computing systems (icdcs), 2016 ieee 36th international conference on (IEEE, 2016), pp. 405–414.

[9] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: google's globally-distributed database", in Proceedings of the 10th usenix conference on operating systems design and implementation, OSDI'12 (2012), pp. 251–264.

[10] V. Venkataramani, Z. Amsden, N. Bronson, G. Cabrera III, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, J. Hoon, S. Kulkarni, N. Lawrence, M. Marchukov, D. Petrov, and L. Puzar, "Tao: how facebook serves the social graph", in Proceedings of the 2012 acm sigmod international conference on management of data, SIGMOD '12 (2012), pp. 791–792.

[11] LinkedIn, *Ambry*, `https://github.com/linkedin/ambry`.

[12] Lasp lang, *Lasp*, `https://lasp-lang.readme.io/`.

[13] Lasp lang, *Lasp crdt support*, `https://github.com/lasp-lang/types`.

[14] Basho, *Riak dt*, `https://github.com/basho/riak_dt`.

[15] M. Zawirski, N. Preguiça, S. Duarte, A. Bieniusa, V. Balegas, and M. Shapiro, "Write fast, read in the past: causal consistency for client-side applications", in Proceedings of the 16th annual middleware conference (ACM, 2015), pp. 75–87.

[16] M. Zawirski, A. Bieniusa, V. Balegas, S. Duarte, C. Baquero, M. Shapiro, and N. Preguiça, "Swiftcloud: fault-tolerant geo-replication integrated all the way to the client machine", (2013).

[17] P. E. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (lsm-tree)", Acta Informatica 33 (4), 351–385 (June 1996).

[18] Google, *Leveldb*, `https://github.com/google/leveldb`.

[19] W. Vogels, "Eventually consistent", Commun. ACM **52**, 40–44 (2009).

[20] Y. Saito, and M. Shapiro, "Optimistic replication", ACM Comput. Surv. **37**, 42–81 (2005).

[21] M. Ahmad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto, "Causal memory: definitions, implementation, and programming.", Distributed Computing 9, 1, 37–49 (March 1995).

[22] P. A. Bernstein, and N. Goodman, "Concurrency control in distributed database systems", ACM Comput. Surv. **13**, 185–221 (1981).

[23] Sync Free, *Sync free*, `https://syncfree.lip6.fr/`.

[24] Erlang, *Disk log*, `http://erlang.org/doc/man/disk$%5C_$log.html`.

[25] Erlang, *Ets tables*, `http://erlang.org/doc/man/ets.html`.

[26] Basho, *Basho bench*, `https://github.com/basho/basho$%5C_$bench`.

[27] *Grid5000*, `https://www.grid5000.fr`.

[28] iMatix, *Zero mq*, `http://zeromq.org/`.

[29] Basho, *Eleveldb*, `https://github.com/basho/eleveldb`.

[30] Software Engineering Laboratory, National Technical University of Athens, *Proper*, `http://proper.softlab.ntua.gr/`.

[31] Google, *Snappy*, `https://google.github.io/snappy/`.

[32] Facebook Database Engineering Team, *Rocksdb*, `http://rocksdb.org/`.

[33] Facebook Database Engineering Team, *Rocksdb open source code*, `https://github.com/facebook/rocksdb`.

[34] Leo project, *Erocksdb*, `https://github.com/leo-project/erocksdb`.