Pedro Miguel Sousa Lopes

B.Sc. in Computer Science and Informatics

Antidote SQL: SQL for Weakly Consistent Databases

Dissertation submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Informatics Engineering

Adviser: Nuno Manuel Ribeiro Preguiça, Associate Professor,
NOVA University of Lisbon

Examination Committee

Chairperson:	Prof. Dr. João Leite
Raporteur:	Prof. Dr. José Pereira
Member:	Prof. Dr. Nuno Preguiça

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

December, 2018

Antidote SQL: SQL for Weakly Consistent Databases

This document was created using the (pdf)LaTeX processor, based in the "novathesis" template[1], developed at the Dep. Informática of FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

To my parents and brother.

Acknowledgements

My first acknowledge goes to my adviser, Professor Doctor Nuno Preguiça, for accepting me in this project and for the time he spent on guiding me throughout the past years. Despite being out of functions, the professor has always been supportive and available to help me in this dissertation, and I appreciate all his efforts for the sake of the project.

I would like to thank Valter Balegas as well for his opinions, ideas, occasional conversations, and for his presence in our meetings that led us to new paths for this project.

My third acknowledge goes to Annette Bieniusa and Maximilian Kohl, from the University of Kaiserslautern, for their time on providing me with all the help to understand and use the locking system presented in this work.

I would like to express my gratitude to Professor Doctor Carla Ferreira and Professor Doctor Rodrigo Rodrigues for their time on the discussion of some solutions for this work. My gratitude goes also for the PhD student Gonçalo Tomás, for his time on providing this project with a HTTP server.

In general, I thank the FCT-NOVA for the opportunity, and especially the Department of Computer Science for all the conditions provided for the realization of this thesis.

To my colleagues and friends, of whom we shared good conversations, exchanged ideas, and spent good times together inside and outside the work time. To some other close and special friends, for whom I appreciate the attention, time and emotional support provided, that made the elaboration of this thesis much easier.

At last, I thank my parents and my brother for all their support at the good and bad moments during my academic journey, for their patience, and also for their love and emotional support. Without them, I could never have reached this stage in my life and I really thank them for all their sacrifices.

This work was inserted in the scope of the EU H2020 LightKone project (732505) and was partially co-financed by FEDER funds through the LISBOA 2020 Programme and by National funds through FCT/MCTES under the project LISBOA-01-0145-FEDER-032662 / PTDC/CCI-INF/32662/2017.

Abstract

Distributed storage systems, such as NoSQL databases, employ weakly consistent semantics for providing good scalability and availability for web services and applications. NoSQL offers scalability for the applications but it lacks functionality that could be used by programmers for reasoning about the correctness of their applications. On the other hand, SQL databases provide strongly consistent semantics that hurt availability.

The goal of this work is to provide efficient support for SQL features on top of AntidoteDB NoSQL database, a weakly consistent data store. To that end, we extend AQL, a SQL interface for the AntidoteDB database, with new designed solutions for supporting common SQL features. We focus on improving support for the SQL invariant known as referential integrity, by providing solutions with relaxed and strict consistency models, this latter at the cost of requiring coordination among replicas; on allowing to apply partitioning to tables; and on developing an indexing system for managing primary and secondary indexes on the database, and for improving query processing inspired on SQL syntax.

We evaluate our solution using the Basho Bench benchmarking tool and compare the performance of both systems, AQL and AntidoteDB, regarding the cost introduced by the referential integrity mechanism, the benefits of using secondary indexes on the query processing, and the performance of partitioning the database. Our results show that our referential integrity solution is optimal in performance when delete operations are not issued to the database, although noticeably slower with deletes on cascade, and implementing secondary indexes on the NoSQL data store shows improvements on query performance. We also show that partitioning does not harm the performance of the system.

Keywords: Web applications; geo-replicated databases; NoSQL; query language; CRDT.

Resumo

Os sistemas de armazenamento distribuídos, tais como bases de dados NoSQL, empregam semânticas fracamente consistentes para fornecerem boa escalabilidade e disponibilidade aos serviços e aplicações Web. O NoSQL oferece escalabilidade para as aplicações, mas carece de funcionalidades que podem ser usadas pelos programadores para se fundamentarem na correção das suas aplicações. Por outro lado, as bases de dados SQL fornecem semânticas de consistência forte que prejudicam a disponibilidade.

O objetivo deste trabalho é fornecer o suporte eficiente de funcionalidades SQL sobre a base de dados NoSQL AntidoteDB, um sistema de armazenamento fracamente consistente. Para esse fim, estendemos AQL, uma interface SQL para o AntidoteDB, apresentando novas soluções para suportar funcionalidades comuns de SQL. Focamo-nos em melhorar o suporte da invariante SQL conhecida como integridade referencial, ao fornecer soluções em modelos de consistência relaxados e estritos, com este último modelo exigindo um custo adicionado pela coordenação entre réplicas; em permitir que seja aplicado particionamento em tabelas; e em desenvolver um sistema de indexação para gerir índices primários e secundários na base de dados e para melhorar o processamento de consultas, com inspiração na síntaxe SQL.

Avaliamos a nossa solução usando a ferramenta de *benchmarking* Basho Bench e comparamos o desempenho de ambos os sistemas, AQL e AntidoteDB, quanto ao custo introduzido pelo mecanismo de integridade referencial, aos benefícios do uso de índices secundários no processamento de consultas e ao desempenho do particionamento na base de dados. Os nossos resultados mostram que a nossa solução de integridade referencial é ótima em desempenho quando não são emitidas remoções à base de dados, embora visivelmente mais lenta com remoções em cascata, e a implementação de índices secundários na base de dados NoSQL mostra melhorias no desempenho das consultas. Também mostramos que o particionamento não prejudica o desempenho do sistema.

Palavras-chave: Aplicações Web; bases de dados geo-replicadas; NoSQL; linguagem de consulta; CRDT.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Context

In today's world, the need to have highly available data and fast data retrieval is increasing substantially. Worldwide users who enjoy the services provided on the Internet are becoming increasingly demanding about their data, regarding its persistence, availability, and the time that it takes to get the data requested to the services [70]. This indicates that users are becoming less tolerant to the delays in retrieving data from applications and/or services, and data loss, even if it is only a partial loss.

To meet these requirements, services use storage systems to hold data in a scalable, properly consistent, persistent, and available manner. These storage systems are built in such a way that their data is reached by geographically spread users. The straightforward approach for supporting this requirement is to implement storage systems in the cloud, where multiple, interconnected data centers are spread across the globe. Typically, in a geo-replicated setting, data is replicated across multiple data centers allowing for a more fault-tolerant environment for crucial data. Additionally, with this approach users can get data with decreased latencies by accessing their data in the closest data center. Since users do not tolerate an unavailable service, globally distributed storage systems resort to lower consistency data models for providing availability in the presence of faults. In this scenario, users may see stale data but always receive a reply from the system. Often, users do not mind this setting and tolerate a certain level of data inconsistency, which opens new doors to storage systems that are willing to deploy weaker specifications on data consistency.

1.2 Motivation

The literature of distributed databases defines two major database classes: SQL and NoSQL databases.

SQL databases are widely used by organizations who wish to organize their data in a structured and well-defined manner. A SQL database follows the relational model [74], whose schema is composed of tables, indexes, relationships, stored procedures, materialized views, and many other features. For querying this kind of database, a structured query language (SQL) is used, which is based on relational algebra. SQL databases provide strong data consistency, enforcing ACID properties. In distributed systems, this may lead to lower data availability, as the execution of some operations requires coordination among multiple replicas.

On the other side, NoSQL databases do not follow the relational model and present a variety of choices in what concerns data modeling and structure. Therefore, many of the NoSQL databases present very basic and limited interfaces composed of simple *put*, *get*, and *remove* operations to manipulate data items. Opposed to SQL, NoSQL databases are ideally used for availability and scalability, which implies that strong consistency may be dropped.

NoSQL databases are often preferred to SQL databases in distributed environments since they are more scalable, available, and exhibit better latency for clients. Yet, NoSQL lacks functionality when compared to SQL, which in turn presents very rich features that allow for complex data structuring, better query performance, and ease of programming. For instance, SQL's strong data consistency prevents global invariants (such as primary key and integrity constraints) from being violated by applying transactional isolation semantics for this purpose. Maintaining these database invariants are crucial for data coherence and consistency.

Therefore, the main challenge of NoSQL databases, and the consequent purpose of this thesis, is extending their functionality by adding some SQL features that allow for stronger consistency and better query performance, which may improve the usability of NoSQL databases.

1.3 Proposed Work

In this section, we briefly describe the proposed work for this thesis. For a better understanding, we separate the proposed work into subgroups, where each group touches a specific functionality.

This work builds on AntidoteDB [3, 8], a key-value store that supports data geo-replication, for creating a new version of the Antidote Query Language (AQL) [75], the interface that provides SQL functionalities for the AntidoteDB data store. The current versions of the two systems presented some limitations which we address in this work by proposing new solutions, which we will describe in the following sections.

1.3.1 Referential Integrity

The main purpose of the AQL was to implement SQL invariants on AntidoteDB, making it the first system to deploy this SQL feature in a NoSQL database. However, the current design and prototype include two main limitations for referential integrity: the *delete-cascade* mechanism is the only option in AQL concerning the deletion of rows, and the solution for referential integrity lacks stronger concurrency semantics to be specified on foreign keys.

To meet these limitations, we design an optional **delete-cascading mechanism** on AQL, allowing to define a restrict behavior on foreign keys that prevents rows related through referential integrity from being deleted. We changed the algorithm for enforcing referential integrity as well to implement more correct semantics.

In addition, we study the design of **strict foreign key semantics** that allow concurrent updates to be executed on rows related through referential integrity in an exclusive basis. The programmer may choose whether operations on rows are executed on restrict or more relaxed semantics.

1.3.2 Querying Optimization

The query implementation on AQL is very limited, as only basic conditional AQL SELECT statements can be used to query the database. In fact, an AQL SELECT can express only a WHERE clause with a single equality on the primary key column.

To overcome this limitation, we propose the implementation of two sub-systems, at the database level: an **indexing sub-system** and a **query optimizer**. The first manages all data related to the indexes, including primary and secondary indexes; it is intended to trigger index updates or read index data whenever a table is updated or a read operation (mostly a high-level query) is issued to the database, respectively. The second is responsible for processing in a well-structured manner the high-level queries issued from the AQL interface, and for retrieving back to AQL data already filtered by the parameterized queries. This latter sub-system takes advantage of the indexing sub-system and the partial object retrieval (mentioned in the next section).

1.3.3 Partial Object Retrieval

The AntidoteDB design imposes a significant limitation on performance and usability due to only supporting full database objects to be retrieved from the database.

For overcoming this limitation, we propose the extension of the AntidoteDB's API to support **partial object retrieval**, that allows objects to be retrieved from the database in smaller data chunks, such that these chunks reflect the application of a *read function* on an entire object. We export this functionality into CRDTs, whose specifications must export properly the read functions that allow to retrieve partial parts from the respective objects. We also provide index data structures with this semantics.

1.3.4 Database Partitioning

For preparing AntidoteDB for supporting partial replication, where an object is stored in a subset of data centers, we propose the design of **table partitioning**, which affects both AQL and AntidoteDB systems. The goal of this feature is to allow tables to be split into smaller tables, such that each small table contains rows that share the same value of a column. We design this feature such that it would make possible for AntidoteDB to resort to different replication techniques for splitting small tables strategically among several data centers.

1.3.5 System Architecture and Model

The architecture proposed for this work follow the architectures from AQL and AntidoteDB: a single AQL module is coupled to a single instance of AntidoteDB within a single server, where this relation is analogous to a relation between a query processor and a storage system, respectively. Clients communicate remotely with the AQL module to query the database, which in turn communicates with the AntidoteDB instance to obtain the data. Each server belongs to a single data center and replicates data within and across data centers. We maintain the server-side architecture provided by AQL, since it is more efficient in this way[1].

Some of the aforementioned features such as secondary indexing, query optimization, and partial object retrieval are implemented within AntidoteDB. Following the AntidoteDB's architecture, we create separate modules near the middle components of the architecture, such as the Materializer component [8].

The proposed data model will follow the AQL's data model. We keep the existing mapping between tables and *buckets* (from the AntidoteDB abstraction) and between table rows and key-value objects. Likewise, we expect secondary indexes to be internal data structures of the AntidoteDB data store: each secondary index will be mapped into a single database entry, i.e. into a CRDT object [72, 73], and will belong to a single table. To address this design, we propose the creation of brand new CRDT specifications designed for attending the needs for storing index data. Therefore, we propose two distinct designs of CRDTs, one for each type of index, that differs on the data that each type will store. For instance, primary indexes tend to store primary keys in their raw form (i.e. the textual or numeric representation of the primary key) and the correspondent database key, while the secondary indexes store raw column values (from the indexed column) and the primary keys that point to it. Moreover, the AQL's query language will include an operation to create secondary indexes.

As mentioned earlier, this work added support for range queries, deferred by sequences of logically connected comparisons (i.e. through logical AND/OR operators) on columns and typed values (e.g. strings and integers).

[1] A client-side architecture was already studied and tested in the past [75].

1.3.6 Contributions

The main contributions of this work reflect extensions to both AntidoteDB and AQL for improving both systems in terms of performance and latency. Essentially, with this work we contribute with:

- An improved design for supporting referential integrity with an optimistic approach in AQL, implementing additional options for programmers;
- A new design for supporting referential integrity in AQL by imposing minimal restriction to concurrent updates;
- The embedding of explicit secondary indexes within the AntidoteDB data store, in order to optimize the performance of read operations and range query supporting;
- The support of partial retrieval of AntidoteDB's objects, preventing the retrieval of full objects for message overhead decrease;
- The support for partitioning database tables for designing new replication techniques on the database;
- The design, implementation, and evaluation of these features in AQL and AntidoteDB.

1.4 Document Organization

The remainder of this document is organized as follows:

Chapter 2 describes the concepts, definitions, and existing systems related to this thesis, with focus on storage systems. The concepts and definitions include consistency models, replication, data storage, and SQL interfaces for NoSQL data stores.

Chapter 3 describes the main systems that support this work, namely the AntidoteDB, Conflict-free Replicated Data Types, and the Antidote Query Language.

Chapter 4 describes the solution proposed for this work in terms of its architecture and design. The design focuses on database invariants and query optimization.

Chapter 5 explains the programming environment and additional tools for employing, testing, and deploying the solution proposed. Additionally, the internal representation of the solution is described in this chapter.

Chapter 6 describes the evaluation of the proposed solution recurring to benchmarking.

Chapter 7 concludes this thesis with a discussion and future work.

Chapter 2

Related Work

In distributed systems, system designers must define several important architectural aspects about the underlying storage system that help to accomplish several goals, such as providing a good user experience, data availability, fault tolerance, and data persistence.

In this chapter, we focus on the main architectural design aspects of storage systems that represent a large part of the related work for this thesis and that contribute for its understanding and realization. The organization of this chapter is as follows: we start by defining data consistency models on Section 2.1, then we introduce data replication concepts on Section 2.2. In Section 2.3, we present the database storage concepts that are important for this thesis, and finally, we present in Section 2.4 some existing systems that serve as examples of mapping SQL interfaces to NoSQL data storages.

2.1 Consistency Models

The consistency models are important properties that influence the design of distributed storage systems, as they define the set of rules for executing operations. These rules allow to define how restrictively we want operations to execute under concurrency.

There are two predominant consistency models on distributed systems: strong consistency and weak consistency. These two models have different characteristics and may have different choices of consistency implementations. Depending on the demands of an application or service, one must select which consistency models must be implemented in the storage system.

2.1.1 Strong Consistency

A strong consistency model is a model in which operations seem to execute sequentially, as if they were executed in a single machine. The most known strong consistency models are linearizability and serializability.

Linearizability. *Linearizability* [49] is defined as means for guaranteeing that executions of operations on a single object follow a real time order among each other. More precisely, from a certain point in the execution of an operation over a single object, the effects of applying the operation are seen in all the executions (concurrent or not) of operations (over that object) that follow the first operation. That point in time is also known as *linearization point*, defined as an instant in time between the invocation and the end of an operation over an object in which the operation appears to be instantaneously executed on the object. Due to these properties, linearizability is defined as a local property of an object.

This type of consistency is well known for being deployed in systems that implement State Machine Replication [54], in which we want to establish an order in which operations must be executed by all processes in the system.

Serializability. In contrast to linearizability, *serializability* [61] is not related to a single object, but to a set of objects and their operations executed in the context of transactions. Serializability assures that transactions that execute a group of operations over a set of different objects are seemed to be executed sequentially, creating a total order between all transactions of a system. This is a strict consistency model designed for databases and it provides guarantees that preclude any concurrent anomalies in the database (more details about the anomalies in Section 2.3.4.1).

A strong consistency model related to serializability is *strict serializability*. Unlike serializability, in this model, all transactions in the system follow a real time order and provide a serial (sequential) execution of operations compatible with the real time [58]. It follows that strict serializability is a combination of linearizability and serializability.

2.1.2 Weak Consistency

A weak consistency model is a more relaxed model compared to the strong consistency one, as a process may see an inconsistent view of the system's state during a certain time period. A state may consist in a set of data objects, for example. Currently, there is a variety of this type of consistency model, but we will focus on eventual consistency and causal consistency.

2.1.2.1 Eventual Consistency

This is the weakest model known among all consistency models. Its only requirement is that eventually, when there are no more writes on the system, all processes will converge to the same state. This may cause clients to see different and stale states from the system. The convergence property behind the eventual consistency model can also be defined as a *liveness* property. *Liveness* ensures that during the whole execution of the system, processes will not block forever and will continue their operation, despite concurrency.

This consistency model is sometimes a subject for discussion, as storage systems dubbed as providing eventual consistency often provide additional guarantees, such as conflict resolution policies or strong data guarantees (e.g. transactions), for addressing the limitations of eventual consistency and addressing the requirements of their applications [21].

Strong Eventual Consistency. *Strong Eventual Consistency* (SEC) is a more recent consistency model introduced by Shapiro et al. [73] covering single objects. This consistency model introduces stronger properties to the regular eventual consistency, as besides guaranteeing liveness, it also provides a *safety* property at the granularity of a single object. A *safety* property ensures that nothing bad is going to happen during the whole execution of the system, which is equivalent to say that some conditions hold in any point of the execution. In the case of SEC, the condition proposed is that nodes of a replicated system, also known as replicas, that receive the same, possibly unordered set of update operations for a single object converge to the same state. Conflict-free Replicated Data Types (CRDTs), the data structures proposed by the authors, follow this consistency model.

2.1.2.2 Causal Consistency

In *causal consistency*, operations follow a causal order among them. More specifically, operations that are causally related have a precedence relationship among them which determines their execution order in the system.

The *happened-before* relation [53] (denoted by $\rightarrow$) is commonly associated with causal consistency. Using this relation, it is possible to create a partial ordering of events[1] among the events in a distributed system. A *happened-before* relation $\rightarrow$ between two events, a and b, is defined as holding the following conditions:

1. If events a and b belong to the same process, and if a occurs before b, then $a \rightarrow b$;

2. If the event a is the sending of a message by a process, and b is the receipt of the same message by other process, then $a \rightarrow b$;

3. If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$ (transitivity).

It follows from the definition that two events, a and b, are concurrent if neither a occurred before b, nor b occurred before a. That is, neither $a \rightarrow b$, nor $b \rightarrow a$. As a consequence of this property, concurrent operations do not need to be seen in the same order by all processes. Instead, this is only required for causally related operations. Thus,

[1] In the context of data storage systems, these events may be the execution of read or write operations in the system's state, or the sending and the receipt of messages between storage nodes in the network.

system convergence may never be achieved under causal consistency, unlike eventual consistency.

This relation of partial ordering revealed to be very important in the context of distributed systems since it is quite infeasible nowadays to synchronize processes' clocks to order events. Logical clocks [53] help on causality since they can be used to relate events through the *happened-before* relation thus making it possible to develop algorithms for the ordering of updates. For instance, the well-known algorithm for vector clocks [41] is an implementation of logical clocks. Vector clocks are based on the assignment of timestamps to events (or operations, when it applies to a data storage system, for example) and are broadly used by many distributed systems.

Causal+ consistency. *Causal+ consistency* [4, 56] can be seen as an extension to causal consistency, with the particularity that, besides providing a causal order of operations, it also requires processes in the system to converge to a single state, in the presence of concurrent operations. Thus, this consistency model assumes that processes will diverge for a limited amount of time, until they eventually synchronize into a single state.

2.1.3 Discussion

Distributed storage systems designers always try to achieve a balance between consistency models: one could implement one of the consistency models (under strong or weak consistency) or try to get the best of both models. Usually, systems aim to achieve the strongest consistency model that offers the best data consistency but that does not compromise other important factors such as availability, user's latency, performance, and scalability.

Although it is a priority to meet most of these factors, it might not be possible to accomplish all of them at once. The CAP theorem [43] states that a distributed system cannot achieve simultaneously strong consistency, availability, and tolerance to network partitions. As partitions are inevitably present in systems that resort to the network to establish their communications, partition tolerance is not seen as an optional implementation by most systems. Thus, these systems tend to compromise consistency in exchange for better availability and partition-tolerance by implementing a weaker consistency model. These systems are known as AP systems.

ALPS (***A****vailability, low* ***L****atency,* ***P****artition-tolerance, high* ***S****calability*) systems [56], like social media networks, tend to sacrifice strong consistency properties to achieve better availability, scalability and low latency for their users, through the usage of a weakly consistent storage system. For instance, Dynamo [27] is a storage system that offers an eventually consistent and replicated data store to its clients in exchange for availability and performance: write operations issued by clients are propagated asynchronously to all replicas, which can result in different views of the system state.

On the other hand, some systems offer the possibility to execute operations in a strong consistent or weak consistent way. Gemini is a storage system that provides the coexistence of multiple consistency levels in a single place, through RedBlue consistency [55]. With RedBlue consistency, operations can be labeled of blue or red, whether they may be executed in different orders in different sites[2] or they must be executed with strong consistent semantics, following a strict order in different sites. Blue operations must be globally commutative and invariant safe (they do not compromise system specifications), and may follow different orders among all nodes; red operations usually violate system invariants and must be totally ordered among all nodes (e.g. linearizability).

Systems like Sieve [55] follow the previous principle, which is based on simple annotations the programmer must issue to the application that indicate the proper behavior of operations in the presence of concurrent executions.

2.2 Replication

Most distributed storage systems resort to replication to scatter their data across a given set of nodes (i.e. replicas) in the network. The main reasons to replicate data are to achieve fault tolerance, good performance, better latency for clients, high availability, and scalability (based on [56]):

- **Fault tolerance**: Hardware faults and other component outages may happen at any time in a distributed system. Replication introduces fault-tolerant mechanisms that allow for easy data recovery on node faults or network failures (e.g. link failures), while the system operates as expected.

- **Throughput**: Replication provides high throughput to a distributed system when more than one node processes operations. A single node might not be enough to handle a whole workload, and more nodes are added to increase processing power and parallelism.

- **Client latency**: Replication architectures may improve the client communication latency with the system, by placing data geographically closer to clients.

- **High availability**: By placing data across multiple machines, we are improving data availability, since a failure in a node will not cause its data to be permanently lost, because other node(s) will possess some of the (or the whole) data of the failing node.

- **Scalability**: Replication provides scalability when adding new nodes to the system increases not only availability, but also increases throughput on load spikes (i.e. on bulks of client operations) and storage capacity.

[2]A site may be a data center or a group of network components, including application servers and data storage systems.

The use of no replication is opposite to the use of replication on a storage system. In a scenario with no replication, a single machine (or server) holds all system's state data and satisfies client requests. At first, this solution might be beneficial regarding the ordering of updates induced by client requests, since only a single replica is receiving the updates and it is easier to serialize the operations. Plus, with a single server, issues regarding data consistency, replica state convergence, and replica synchronization do not apply since data is all concentrated within a single physical point.

However, this solution presents some drawbacks. Today, web applications and online services require high capabilities to support huge request loads from their clients. In this scenario, a single server could present a performance bottleneck and could also hurt the latency between the server and the clients. Moreover, a single server is seen as a single point of failure where the failure of the server could mean the complete, permanent loss of all user and system data.

In this section, we present the concepts of replication more relevant for distributed storage systems, namely, known replication architectures (Section 2.2.1), replication strategies (Section 2.2.2), and the concurrent execution of operations in replicated storage systems (Section 2.2.3).

2.2.1 Architectures

Depending on the needs of a system and their clients, replication can be performed on a single-site or geo-replicated basis:

- **Single-site replication**: In this type of architecture, nodes that contain data are geographically close to each other in the network.

- **Geo-replication**: Unlike single-site replication, in geo-replication system nodes are distributed and strategically located across the globe. Usually, a major reason to build a geo-replicated system is to reach a high number of clients scattered across the globe, in order to decrease the latency between the clients and the system.

Single-site replication can be a good solution for a private organization that wishes to put data in a single geographical place, in order to allow data accesses at low latencies within the organization. The organization may also not be concerned with (or it is not applied to) data accesses from users outside the organization or scattered across the globe.

On the other hand, **geo-replication** is the common solution employed by companies who are concerned about user latencies. The system is replicated across the globe to reach a higher number of users and improve user experience and data availability. Although this solution is the more complex one and more difficult to manage when it comes to replica consistency, synchronization and fault tolerance, it is employed by well known distributed systems nowadays, for instance, the Amazon, Google, Facebook, and Yahoo. It is also employed by distributed databases, for instance, Yesquel [2], Cassandra [52], and

AntidoteDB [3]. The database we are focusing on is the AntidoteDB database, as it is the core database system for this thesis. It follows from this that our work will be developed over a geo-replicated architecture.

2.2.1.1 Intra-DC and Inter-DC Replication

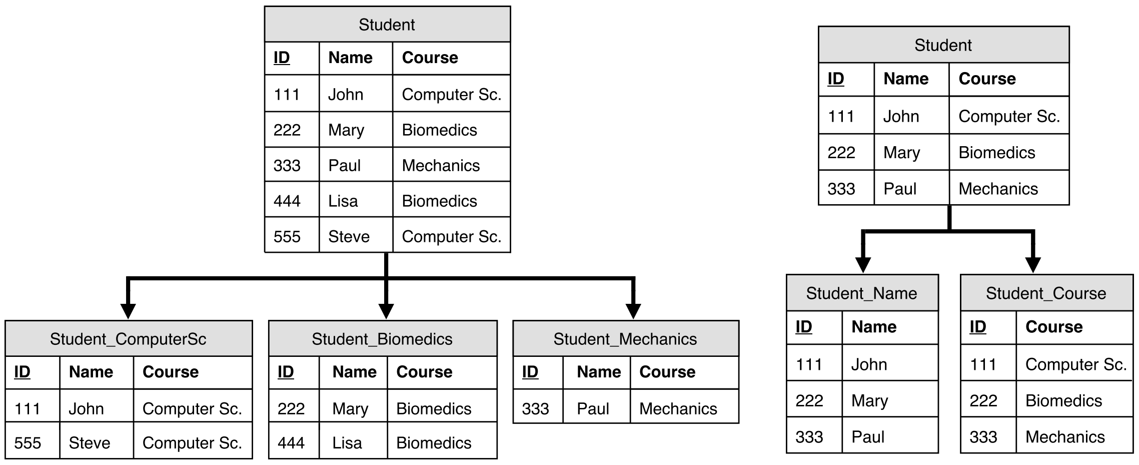
Web applications and Internet-based services that hold large amounts of information about their users usually store their data in multiple data centers across the planet. For fault tolerance and durability, these applications and services resort to storage systems that replicate data within a data center. An **intra-data center** replication architecture is applied such that data is replicated among the hosts inside the data center, let the hosts be storage servers in a cluster that communicate between each other, for instance. Several replication techniques may be used inside a data center, namely partitioning (see Section 2.2.1.2), partial replication (where a data item is replicated among a small number of hosts), and State Machine Replication [57].

Also, replication implementations between data centers may be used, i.e. **inter-data center** replication, where similar clusters belonging to distinct data centers communicate between each other for data replication. Distributed data storages that implement weakly consistent semantics use lazy replication techniques (i.e. asynchronous transmission of updates) to replicate data among data centers. Application clients usually submit requests to a single data center, typically the closest one for lower latencies, and always see their corresponding updates with no delays. On the contrary, with lazy replication, other clients may see the updates from other clients with a small delay, which is tolerable from a weakly consistent system [76].

2.2.1.2 Data *Sharding*

In a replicated system, *sharding* is a partitioning strategy (namely, horizontal partitioning) and a scalability technique represented as the act of splitting data into *shards* across multiple nodes in the network. Most database systems resort to *sharding* to allow database objects, typically large in size, to be partitioned into smaller ones, in order to improve query performance and load/capacity balancing:

- **Query performance**: Data can be split to improve querying from the database and to reduce costs of accessing data. Usually, transactions tend to read related data at once from different data structures (e.g. tables) across the same database. An example of good partitioning is considering a social network application, where a photo album of a user might be placed together in the same machine or same geographical location, since the album photos are accessed at once by a user.
- **Load and capacity balancing**: Due to *sharding*, distinct *shards* of data are spread across different nodes in the network, making it possible for balancing client requests and also for easily managing each node's capacity; if data is too big for a



Student

ID	Name	Course
111	John	Computer Sc.
222	Mary	Biomedics
333	Paul	Mechanics
444	Lisa	Biomedics
555	Steve	Computer Sc.

Student_ComputerSc

ID	Name	Course
111	John	Computer Sc.
555	Steve	Computer Sc.

Student_Biomedics

ID	Name	Course
222	Mary	Biomedics
444	Lisa	Biomedics

Student_Mechanics

ID	Name	Course
333	Paul	Mechanics

(a) Horizontal partitioning.

Student

ID	Name	Course
111	John	Computer Sc.
222	Mary	Biomedics
333	Paul	Mechanics

Student_Name

ID	Name
111	John
222	Mary
333	Paul

Student_Course

ID	Course
111	Computer Sc.
222	Biomedics
333	Mechanics

(b) Vertical partitioning.

Figure 2.1: Two examples of data partitioning: (a) rows with the same value in the *Course* column are collected into a distinct table; (b) table is split by columns *Name* and *Course*, each of which corresponding to a distinct table.

given node, *sharding* may be a solution to balance data redistribution among the other nodes.

Horizontal partitioning. *Sharding* is also known as a horizontal partitioning technique. In the context of relational databases, horizontal partitioning consists in dividing a table/relation into smaller tables containing subsets of rows of the source table. The rows are collected according to a column of the source table and put into a single *partition*. Those partitions, when combined, build the original table with all rows from all partitions. Figure 2.1a depicts an example of horizontal partitioning.

On non-relational data stores, such as NoSQL, data records are typically identified by a unique key. As such, horizontal partitioning involves splitting a set of records by the key and distribute records to different nodes. A well known horizontal partitioning technique is **consistent hashing** [51], where nodes are disposed in a logical network ring. Each node is responsible for a range of keys (i.e. a key space) that identify pieces of data: each node is assigned an identifier inside the ring, calculated through a hash function, and each data key is assigned an identifier that allows to know its responsible node inside the ring[3]. Organizing network nodes in a ring is indeed a good solution since the departure and the entry of a node only affects its closest neighbors. This technique is used in popular data stores, such as Cassandra [52], Dynamo [27], and Riak [50].

Horizontal partitioning is often related to *shared-nothing* databases, where multiple, independent (i.e. there is no sharing of hardware resources, such as CPU, memory and disk), and interconnected nodes are deployed to hold data. Developing horizontal partition involves concerns regarding the number and the set of nodes holding partitioned

[3]The identifier calculation is performed by calculating the hash of a key, k, using a hash function, H, modulo the number of nodes in the system, n: $H(k) \bmod n$.

data (e.g. rows of a table), for performance and load balancing [2, 59].

Vertical partitioning. On the other hand, **vertical partitioning** consists in physically splitting a relation not by rows, but by a set of columns instead, named *column-group* [23]. A whole table is split into smaller tables, each of which containing a subset of columns (ideally, a column-group) of the first table. This suggests that parts of a record may be present in multiple nodes across the network. All these tables must have a column in common, normally the primary key. In that way, the set of columns of each sub-table intersects in, at least, one column, making it possible to rebuild the original table or a subset of records. Figure 2.1b presents an example of vertical partitioning.

2.2.2 Strategies

For the sake of completeness, we summarize in this section the more relevant replication strategies that may be applied to data replication:

- **Active and Passive replication**: In an **active replication** strategy, all the system replicas execute all operations received by the clients. We are restricting here the operations to be write operations that modify the state of the system. On the other hand, in a **passive replication** strategy only one replica of the system executes operations and it is responsible to send the calculated results to the other replicas.

- **Synchronous and Asynchronous replication**: In a **synchronous replication** strategy, replication is performed between replicas before a response is returned to the client. That is, the client issues a request, and after replication is successfully performed the client receives the response. In a more flexible way, the **asynchronous strategy** allows the user to receive a response before or while replication is being performed by the system replicas. This approach allows a client to see an inconsistent state of the system and, in fact, it may not even ensure that the operation issued is going to be executed or fully replicated.

- ***Master-Slave* and *Multi-Master* replication**: The ***master-slave*** replication strategy is characterized by a single replica (i.e. the *master*) receiving update operations (i.e. operations that modify the state) and shipping the operations to other replicas (i.e. the *slaves*). All replicas may receive read operations from clients. Conversely, in the ***multi-master*** strategy all replicas are capable of receiving update operations, to execute them, and to propagate them to the remaining replicas.

2.2.3 Operations on Replicated Systems

A distributed data storage system may have thousands to millions of clients issuing operations to the system at the same time. To keep the service available during demand peaks, operations may be executed by a small number of replica nodes (i.e. a replica set) and propagated to other replicas for a total replication. This is a common approach when storage systems apply weakly consistent data models for availability.

However, relaxing consistency introduces a problem: since clients will issue operations at the same time, there is a high chance of those operations be concurrent and lead to an inconsistent data state. For instance, imagine when two users of an online shopping store (e.g. Amazon) try to buy the same and only product in the stock. If both users issue the buy operation to different replica sets, those replicas may not see the operations issued by each other, which may result in an inconsistent, divergent state of the system where both users may think each got the last product, whereas this should not be possible. Divergence reconciliation semantics allow to choose which client will buy the product and will *win* over the other client.

2.2.3.1 Divergence Resolution

One of the many challenges of replication over a weakly consistent data storage is the divergence reconciliation. Results of concurrent executions must be merged to keep the system state consistent and to resolve divergences. A merge procedure is used to merge concurrent results of executing operations at different replica sets and all replicas must know and apply the same procedure.

Perhaps the most known procedure is the *last-writer-wins* [52] technique, where the last update to the same data piece will remain comparing to previous updates. An implementation of this resolution technique is assigning unique timestamps that respect causality for updates to determine which is the more recent update among a set of concurrent updates [53, 73]. As a consequence, some of the concurrent updates will be lost, since the latest update will win over the oldest ones.

Some other storage systems [27, 81] implement their divergence resolution through application dependent policies, where client applications (outside the data storage) are responsible for merging concurrent results, or automatic application-specific procedures are applied for detection and resolution of concurrent conflicting operations.

Conflict-free Replicated Data Types (CRDT) [72, 73] is a solution for divergence resolution. CRDTs are data types with different designs and concurrency semantics. Typically, these data types are used in weakly consistent systems for providing convergence [4, 55].

2.2.3.2 Invariant Preservation

In a weakly consistent and geo-replicated storage system, read and write operations can be performed locally in a subset of replicas in the system and be later propagated to all replicas asynchronously. This solution improves significantly the client experience, since it reduces the latency between clients and the system, and improves system availability since network contention is lower due to low replica synchronization. As explained earlier, writing on a subset of replicas may lead to state divergence and may lead to invariant violation.

For instance, consider the previous example of the online shopping store, but only with a single user buying the product. Before buying the product, the user must add it first to its shopping cart. Now, consider that the original owner of the product removes the product from sale (perhaps also removes it from the website) at the same moment the client adds it to its shopping cart. The concurrent execution of these two operations may lead to a state where the product is added to the shopping cart of the user but no longer exists in the system. Thus, these two operations are said to be conflicting operations, leading to an invariant violation; in this case, the invariant states that the user can only buy a product if it exists and it is for sale.

Usually, in this type of situation we always want to prevail one of the operations. In the example, either we want to add the product to the shopping cart (hence, keeping the product for sale), or we want to remove the product from sale (hence, not adding it to the cart), but never both. In this context, CRDTs are the best tools to merge the results, since they provide policies to determine which operation will prevail from a pair of conflicting operations. Therefore, the convergent semantics of CRDTs can determine the prevalence between conflicting operations and return final results with low synchronization between replicas. However, CRDTs alone are not sufficient – in the example above, the updates modify the same object, i.e. the product in the online shop; if the updates modified different objects, cross-object invariants could not be preserved nonetheless.

Indigo. Indigo [17] is a middleware system originally created to support a specific consistency model over a weakly consistent and geo-replicated key-value store. The consistency model is known as *Explicit Consistency* and is based upon the designation of application annotations to preserve application invariants over the state of the data store. Through the annotations, it is possible to induce invariant-repair or invariant violation-avoidance techniques to preserve assertions that may be violated by the effects of concurrent operations on the database. For the invariant-repair techniques, *Explicit Consistency* resorts to several CRDT objects to automatically repair invariants. On the other hand, invariant violations are avoided using reservations [63], which assign rights to operations in a multi-level lock basis [47].

IPA. IPA [18] is a system that provides invariant preservation on weakly consistent applications. Similarly to Indigo, this system allows to identify pairs of conflicting operations and their effects (through annotations on the application code) that, when executed concurrently, would lead the application to an inconsistent state. In addition, IPA allows to prevent inconsistencies at development time, by proposing alternative solutions/operations for the identified conflicting operations. For these not solvable conflicts, this system allows the application programmer to manually choose resolution techniques to circumvent the inconsistencies.

2.3 Data Storage

Data organization and data storage are fundamental concepts for storing data for web applications or web services. Today, there is a wide range of options for database engines and distinct database models. Nonetheless, there are two fundamental database models that applications may resort to organize their data: relational databases and non-relational databases.

Non-relational database models, also known as NoSQL, are increasingly emerging in the market of distributed databases because they are more flexible, faster, available, concurrent, and scalable than relational databases, also known as SQL. Yet, NoSQL systems have their limitations, since they lack some features that are seen in SQL systems – query joins, secondary indexes, and transactions –, and they may be more difficult to solve data inconsistencies [2, 4].

In this section, we start by describing some features of SQL and NoSQL systems for the sake of comparison. The following sections describe SQL features that are important for this thesis, namely database constraints, indexing, and transactions.

2.3.1 Database Models

The most notable difference between a SQL and a NoSQL database is their respective data models. A database model defines how data is structured within a database. A SQL database typically follows a relational schema, where data is stored in relations or tables. Each table has a finite set of rows and finite set of columns, each of which with a specified data type. Hence, there is a notion of *records*, where a *record* has a fixed size, i.e. a fixed number of attributes (represented by the columns), and a table has a finite number of *records*. For example, the top table in Figure 2.1 depicts the relational schema, where the primary key attribute name is underlined.

On the other hand, NoSQL databases do not follow the relational model. NoSQL presents several non-relational data organization types, such as *key-value*, *document*, *wide-column*, and *graph* stores. Compared with the SQL data model, NoSQL presents a wider variety of choices that must be used accordingly with specific needs.

A *key-value* store is characterized by storing *key-value* pairs in the database, where a *value* must be assigned to a single and unique *key* and values may acquire different data formats [3, 27, 66, 68]. A *document* store is responsible for storing uniquely identified documents in the database [23]. The objects may be of different types in the same database. Similarly to the relational model, a *wide-column* database stores data on several rows. Each row is uniquely identified by a key and may have multiple attributes (i.e. columns). In turn, each column belongs to a column group or a column family [24, 52]. At last, a *graph* store holds data in a graph format, where *vertices* represent entities and *edges* represent the relationships among vertices [46].

In Chapter 3, we describe AntidoteDB, a *key-value* store used as the supporting database system of this work.

2.3.2 Database Constraints

When designing a database schema, concerning its objects and their relationships, a set of assertions is defined in order to keep the database consistent, avoiding anomalies such as broken relationships among tables, for instance. Relational databases present a more complex set of database constraints than non-relational databases. Database constraints include primary keys, referential integrity, data type checks, numeric checks, among others.

The most basic and also most known constraint implemented by common database systems is the primary key. A **primary key** is a constraint applied to a table in the relational model, more precisely to a column that is characterized for its uniqueness on its values. This constraint implies that two records in the same table cannot have the same value on the primary key attribute, since the primary key uniquely identifies a record in the table. In a non-relational database, typically the primary key is a unique key that identifies an object within the data store.

In relational database systems, tables are related to each other through the use of foreign keys. A **foreign key** is an attribute that references an attribute of another table, usually the primary key of that table, creating a relationship between them. This relationship indicates a rule among them known as *referential integrity*.

The need to build referential integrity arises from the fact that it helps a database designer to reason about query optimization (including deletion optimization), data organization, and data incoherence and integrity. Referential integrity is a common feature of SQL systems, unlike NoSQL systems which usually do not implement this functionality, since there is not a notion of relationship. This is understandable, in fact, due to NoSQL systems being designed for performance and scalability and this feature would impact their performance numbers at high scales. Nevertheless, some SQL interfaces [75] already tried to implement relational foreign keys to enrich NoSQL data stores, with some negative results on latency and throughput.

2.3.3 Indexing

An (nearly) optimal query planner is a must-have for a database, since users query huge amounts of data and they want it with fast accesses. The traditional database access to obtain records from a table is known as a *full table scan*. A full table scan is the act of going through all rows of a table in order to search for specific row(s) (i.e. rows that respect certain requirements) in a sequential order.

Typically, a full table scan demands high number of disk I/O accesses, depending on the result size of the query, which is bad for performance [74]. Achieving good performance can be even more complicated if data blocks are segmented on the disk (i.e. data blocks to be retrieved are in different and distant disk sectors) or there are disk blocks that need yet to be retrieved into main memory. To avoid (or to reduce the effects of) a full table scan, most SQL systems implement indexes.

An index is a data pointer structure that allows to retrieve the necessary data with lower disk I/O and fast accesses. Ordered indexes over table columns (from the relational model) are very useful in SQL queries, since they allow to perform exact value scans and range value scans, and allow to retrieve records sorted by the values of the indexed column (by default, indexes are stored in ascending order). There are two predominant types of ordered indexes, the primary (or clustering) indexes and secondary (or non-clustering) indexes:

- **Primary index**: Typically, a table with a primary key column has implicitly assigned an index for that attribute; this index is also known as primary index. Primary indexes usually apply for queries over the primary key column.
- **Secondary index**: In most queries, the column search is not necessarily the primary key column; in those cases, a secondary index may be created for querying the database. The indexed column of a secondary index might not be, and usually is not, ordered on the source table – records may be disposed randomly within the table which affects the ordering of the indexed column. A secondary index may be a good solution when it is required for the query results to be ordered by the indexed column.

Primary indexes are known as data structures that respect the sequential order in which a table is stored on persistent storage, hence being often related (but not necessarily) to the primary key column. On the other hand, secondary indexes are not related to the sequential order of the table's file(s) and are used for more efficient querying on columns that are not ordered. Usually, the common indexing structures to hold secondary indexes are B-Tree/B+Tree's, although there exists other supporting indexing structures such as bitmap indexes, hash indexes, and others [74]. Secondary indexing will be the main focus of this thesis.

Distributed indexing. Implementing secondary indexes is interesting for Internet applications that support a large number of read requests to their databases, since they can improve query performance and data accesses for frequently queried attributes of the database objects [77]. Several distributed systems have already approached secondary indexes in order to efficiently retrieve data to their clients in a more distributed and fault-tolerant manner. Aguilera et al. [1] propose a distributed B+Tree where each tree node is stored in a single server and each node can be accessed through transactional environments, where common B+Tree operations[4] are grouped together to be executed in

[4] B+Tree management operations include *Insert*, *Update*, *Delete*, *Lookup*, and tree traversal operations include *GetNext* and *GetPrevious* [1].

an atomic fashion. Similarly, Pavlo et al. [62] propose the implementation of distributed secondary indexes by placing each of these data structures in all distinct data partitions that compose the database, thus providing queries with single-partitioned access to data[5].

2.3.4 Transaction Processing

A transaction is a group of operations (read or write operations) over a database and is executed as an independent, atomic unit. Relational database systems that implement strong consistency guarantees, such as SQL databases, tend to implement transactions that provide the ACID properties [47], described as follows:

- **Atomicity**: Such property states that either all operations inside a transaction are executed (and thus, all their respective effects are visible in the database) or none does.
- **Consistency**: Both before the start and after the end of a transaction, the database continues in a consistent state, i.e. all database invariants (e.g. integrity constraints) are still preserved.
- **Isolation**: Every transaction must be executed such that does not interfere with other running, concurrent transactions; a transaction should not watch partial effects from other ongoing transactions.
- **Durability**: Even with the occurrence of failures (e.g. hardware faults, power outages, system crashes), the effects of a *committed* transaction must persist in the database state permanently.

Distributed transactions are similar to general purpose transactions with the particularity that the data read and written by the transaction may be located in different places (e.g. servers) [62]. ACID properties are commonly employed in distributed transactions for distributed SQL systems since the main priority of these systems is to guarantee strong data consistency [15]. For NoSQL database systems, implementing ACID guarantees in distributed transactions may not always be the ideal solution or the main emphasis on the implementation. NoSQL databases are meant to be scalable and fast, and guaranteeing strong properties decreases these metrics, as well as their performance. Hence, instead of following ACID properties, non-relational database transactions tend to follow the BASE properties: **B**asically **A**vailable, **S**oft state, **E**ventually consistent [23]. Dynamo [27], Cassandra [52], and Riak [68] are examples of weakly consistent storage systems that follow this model, by not supporting transactions.

However, some non-relational data stores [2, 56, 57, 76] resort to transactions to execute operations within a controlled environment, avoiding inconsistent states to be

[5]The solution proposed by this paper resorts to full replication of secondary indexes in order to minimize the need to contact all data partitions for retrieving table columns not used in horizontal partitioning. The "single-partitioned" concept used by the authors refers to data accesses to a single partition to retrieve the necessary data. More details in [62].

Table 2.1: Anomalies supported by some database isolation levels (adapted from and more details about the anomalies in [76]).

Anomaly	**Serializability**	**SI**	**PSI**	**Eventual Consistency**
Dirty read	No	No	No	Yes
Non-repeatable read	No	No	No	Yes
Lost update	No	No	No	Yes
Short fork	No	Yes	Yes	Yes
Long fork	No	No	Yes	Yes
Conflicting fork	No	No	No	Yes

shown to their clients. For example, consider a social network application, where a user creates a photo album and then uploads a photo into the album. In an eventually consistent storage system, friends of the user may see the photo before the album is created, which is the wrong behavior. To avoid this anomaly, the two operations of creating an album and uploading a photo may be causally executed inside a transaction.

2.3.4.1 *Parallel Snapshot Isolation*

Snapshot Isolation (SI) [20, 71] is an isolation level implemented by traditional databases, such as Oracle SQL and SQL Server, that guarantees that all read operations inside a transaction see a consistent snapshot of the database[6], and precludes conflicting transactions to commit simultaneously. Conflicting transactions are transactions that update the same pieces of data during their executions, i.e. both cause a write-write conflict. Snapshot Isolation also provides the database with a total order of committed transactions, at their commit time.

This isolation level is used by many database systems because it offers better performance than serializability, for not blocking on read operations and for solving most of the anomalous behaviors of concurrency [76]. Serializability solves all these anomalies, as shown in Table 2.1. Yet, because SI imposes a total order of transactions, it may not be properly indicated for distributed transactions, since it requires more synchronization between the nodes of a replicated storage system.

Parallel Snapshot Isolation (PSI) [76] comes with the need of providing most of the guarantees of SI in a geo-replicated data store that supports transactions. This isolation level extends SI and allows different sites to have different orders of committed transactions, unlike SI that requires a total order. PSI provides asynchronous replication of transactions (also known as lazy replication) across distinct sites and guarantees a causal order of transactions among sites.

For the purpose of implementing the PSI, Sovran et al. proposed a geo-replicated data store, Walter, that besides supporting PSI transactions and all its semantics, it also

[6]A snapshot represents a consistent copy of the database state composed of the effects of all operations committed until the time a transaction starts.

provides techniques to preclude write-write conflicts from occurring on concurrent transactions, namely, the *preferred sites* and the *counting sets*.

2.3.4.2 Highly Available Transactions

Highly Available Transactions (HAT) [4, 14, 71] are characterized as providing a transactional environment that aims for availability, low latencies, and weaker semantics for distributed settings.

Currently, some storage systems resort to HAT transactions to operate with a consistent view over a group of their data keys, whether for reading data, writing data, or doing both. For these purposes, implementations of HAT transactions include read-only transactions [4, 56, 57], write-only transactions [57] or interactive transactions [4] (composed of read and write operations).

Transactional Causal Consistency. The Transactional Causal Consistency (TCC) [4] is a consistency level that supports interactive transactions and mitigates the limitation of implementing solely read-only or/and write-only transactions for a distributed transactional environment, by allowing read and write operations within a single transaction. Additionally, the TCC model assures that all transactions read a snapshot of the data store that is causally consistent. In this context, a snapshot consists in a state that contains the updates of previously committed transactions following the *causal+ consistency* model. TCC also assures the atomicity property of transactions that update keys on the data store.

2.4 SQL Interfaces over NoSQL

In this section, we describe some existing systems that implement the mapping between a SQL interface and a NoSQL data store, or that attempted to achieve the best of NoSQL features from relational data stores. These systems were chosen due to implementing some of the SQL capabilities that we are willing to achieve with this work.

2.4.1 Yesquel

Yesquel [2] is a relational distributed storage system whose main goal is to surpass the low capabilities of non-relation databases by implementing practical SQL features in a scalable environment for web applications.

This storage system attempts to avoid some NoSQL issues such as the transferred complexity to the application, the abundance of NoSQL alternatives that hinder the choices of the programmer, and the vendor lock-in issues when switching between NoSQL systems.

Main Features. Some of the Yesquel main goals and features are presented in the list bellow:

- **SQL functionalities**, that scale as well as NoSQL databases. Some of these features include join queries, secondary indexes, transactions, and aggregations that NoSQL data stores usually do not implement;
- **Distributed data structure**, responsible for storing tables and indexes necessary to hold system's data and to improve querying performance, across a group of servers within a data center;
- **Transactional environment**, which handles simultaneous accesses to data partitioned across storage servers. An underlying key-value storage system is implemented to support the transactional environment. Yesquel provides strong consistency through transactions and data snapshots;
- **Workload distribution and concurrency**, through special tree operations (namely, *splits* and *replits*) that allow to handle data distribution, in order to balance client requests across servers.

Data Model. For scalability, Yesquel implements distributed balanced trees (DBT) [1, 77] scattered across the servers in the system. A DBT, also known for Yesquel as YDBT, is a balanced tree data structure whose nodes are distributed across a group of storage servers, and each node belongs to a single server and is stored as a key-value pair. Additionally, tree nodes may be replicated across servers. Each one of these data structures can hold a table or an index.

System Architecture. Yesquel's system operates inside a data center, where multiple servers are interconnected to provide the service. The data center assumptions involve low-latency and high-bandwidth communications among servers.

As for being a SQL storage system, Yesquel follows the database architecture composed of a query processor and a storage engine. A **query processor** is the component responsible for receiving, parsing, compiling, and executing SQL queries. This component compiles a query plan, executes it, and generates requests to the storage engine.

The **storage engine** is responsible for holding data structures that compose the database, namely tables and indexes, and for receiving requests for reading or writing data on those structures (possibly through transactions). The data structures in cause are stored in the form of distributed balanced trees, or YDBT's.

Therefore, each **storage node** is responsible for locally storing tree nodes and the whole set of storage servers represents an underlying transactional node storage system.

Clients who wish to communicate with the system have embedded the query processor, the storage engine, and a client library that allows the communication between the client and the storage servers. Clients communicate with the system by issuing reads and writes enclosed in transactions, and tree traversal operations. As clients own a query processor, Yesquel can scale processing capacity when the number of clients increases.

2.4.2 Cassandra

Cassandra [22, 52] is a distributed, geo-replicated, and non-relational storage system initially deployed for the Facebook company but that has increasingly growing for other markets that operate with very large datasets.

As any other NoSQL storage system, Cassandra aims for scalability, availability, and fault tolerance to provide a better experience to its clients. For scalability and availability, Cassandra resorts to weak consistency to store its data, allowing clients to see inconsistent states, yet not waiting too long for responses.

For fault tolerance, Cassandra resorts to consistent hashing to partition data across the nodes within a cluster. Each data item (identified by a key) is assigned a **coordinator node** according to a hash function, such that a coordinator becomes responsible for the items whose keys comprise between itself and its predecessor.

Additionally, the system asynchronously replicates its data across a small number of replicas, i.e. by a *replication factor*. This *replication factor* may be configured to be unique across all data centers in a cluster or to exist an individual *replication factor* for each data center. Also, this factor helps coordinator replicas to replicate their data items across the system in such a way that a data item is replicated across as many locations as the *replication factor* indicates.

Cassandra allows clients to define the consistency level of the operations, in terms of whether operations are considered successful or not, depending on the number of responses (by the replicas in the system) obtained by coordinators upon replication. In any case, write operations are sent to all replicas, while read operations may be sent to a smaller number of replicas (defined by the current consistency level).

Data Model. Cassandra is most known for being a column-oriented data model, where data is stored in distributed, multi-dimensional maps. Each map is representative of a table with a set of rows, with each row being identified by a unique key that maps to an object consisting in various $(column, value, timestamp)$ triplets. Maps also employ the column family concept similar to that on Google's BigTable [24], where a set of columns is grouped together to form a column family.

Cassandra Query Language. The Cassandra Query Language (CQL) [26] is a SQL-like query language provided by Cassandra that allows a user to issue queries to the database. The main goal of CQL is to provide the clients with an abstract interface to its data structures plus administrative statements for key space management, consistency levels, and few others. With CQL, it is also possible to create/manage secondary indexes on columns and materialized views on tables, where the former is suitable for faster lookups and low cardinality[7] data, while the latter is suitable for high cardinality data. Currently, CQL does not support range queries or order-by statements for indexed columns.

2.4.3 Redis

Redis [66] is a geo-replicated key-value data store with the particularity of being one of the fastest and scalable NoSQL data stores.

To be fast, Redis is (almost) fully implemented in memory, and although its main purpose is for database implementations, it may be also suitable for caching implementations. Even so, Redis implements persistent techniques to avoid full data loss due to the memory's volatile nature. Hence, durability is achieved either by writing database snapshots to disk or by logging database operations to a file.

Scalability is achieved by following a *master-slave* replication technique. In this case, a *master* replica receives write operations and propagates them asynchronously for all remaining *slave* replicas to perform the operations.

Data Model. Redis stores data in key-value pairs. A key must be unique and is always represented as a simple byte array independently of its type or contents (it may be an empty string or the contents of a picture, for example). For the values, there is a limited set of data structures provided by Redis: *strings*, *lists*, *sets*, *sorted sets*, and *hashes* (composed of field-value pairs).

For indexing, Redis attempted to create secondary indexes for applications through the use of some of its own data structures, namely the ZSET's (Redis sorted sets). The technique of using existing data structures does not hurt performance, since there is no extra complexity added to the indexing. But, on the other hand, this solution has revealed to be impractical because the indexes needed to be manually created and managed (i.e. adding data manually to the index) by the user.

RediSQL. RediSQL [67] is a non-official SQL-like query module developed to embed a SQLite [79] database on Redis, hence adding SQL capabilities on top of a non-relational database. The main motivation of RediSQL is to give Redis a more structured environment to store/cache data. RediSQL provides the database creation, key removal, and query statements execution/creation. Input statements are mostly first converted to SQLite statements and posteriorly executed on the database. Currently, RediSQL supports common queries (e.g. SELECT), table statements (e.g. CREATE), update statements (e.g. INSERT, UPDATE, DELETE), and other complex queries (e.g. JOIN).

[7]The cardinality of a column is characterized by the number of unique values that the column holds. More unique values means higher cardinality, while low cardinality stands for a more dispersed set of values (i.e. few unique values).

2.5 Summary

This chapter presents the building blocks for this work in terms of system characteristics and designs. The design of our proposed solution, as a storage system, takes into account the important subjects mentioned throughout this chapter.

Section 2.1 describes the two predominant consistency models in distributed systems, which are the strong consistency and weak consistency models, and presents a brief discussion on the use of both models.

Section 2.2 presents the basis of replication on storage systems, with special focus on geo-replication. It also presents some replication techniques such as *sharding*, and makes some discussions on invariant preservation on replicated data stores.

Section 2.3 presents some principles on data stores, such as the difference between the SQL and NoSQL database models, the common database constraints/invariants in these kind of databases, the indexing specifications on data stores, and the transactional systems on distributed databases.

At last, Section 2.4 describes some examples of storage systems inspired by this work, such as Yesquel, Cassandra, and Redis. Unlike the previously described systems, the Antidote Query Language (AQL) presented in this work provides a SQL-like query language that supports a more complete set of SQL features on a weakly consistent data store. Among these features include a table/row data model, a referential integrity mechanism, an indexing system, a query optimizer, and table partitioning. The other difference between AQL and these systems is that AQL uses invariant preservation techniques with low-coordination between the objects of the data store, for providing the aforementioned SQL features in a weakly consistent manner.

Chapter 3

Background

This chapter introduces the systems that serve as a starting point to this work, namely AntidoteDB, a weakly consistent, geo-replicated database, Conflict-free Replicated Data Types, the data structures that build the objects of the AntidoteDB database, and Antidote Query Language, a SQL-like interface for AntidoteDB.

3.1 AntidoteDB

AntidoteDB [3, 8] is a geo-replicated key-value store initially created in the context of the SyncFree EU project [80]. Its characteristics are based on providing high availability through geo-replication, data *sharding*, data types with strong semantics for data state convergence, and highly available transactions:

- **Geo-replication**: The AntidoteDB resorts to geo-replication to scatter the database across multiple geo-located data centers, composed of clusters of servers. For availability and fault tolerance, the database is *sharded* and replicated on a defined set of replicas [27, 50, 52].

- **Data model and convergence**: The AntidoteDB resorts to Conflict-free Replicated Data Types (CRDTs) for stronger semantics than the offered by eventual consistent systems. CRDTs allow for asynchronous propagation of updates across all replicas in the system and embed conflict resolution techniques for concurrent operations. The data store supports multiple data types (e.g. registers, sets, counters) and supports the notion of *buckets* as agglomerated sets of key-value pairs (similar to Riak's buckets [68]). Plus, it allows the programmer to embed her own data types.

- **Transactions**: AntidoteDB implements Highly Available Transactions (HAT) [71] to support aggregation of updates to be executed in an atomic fashion, with causal

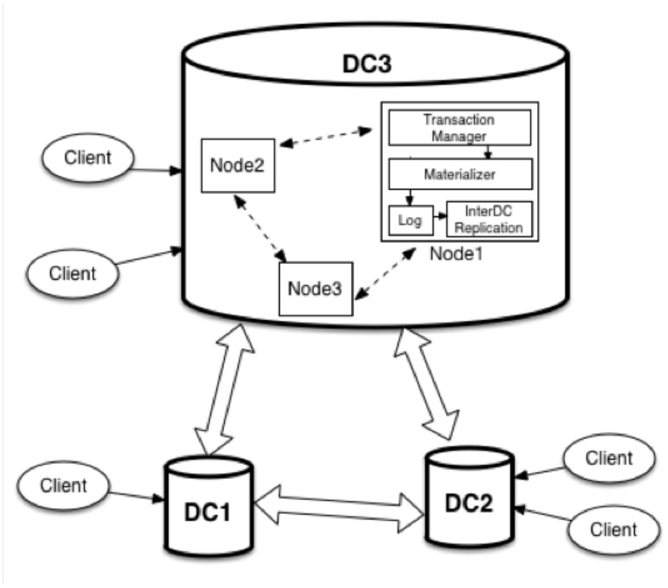


Figure 3.1: Architecture of the AntidoteDB (taken from [8]).

consistency among transactions. To provide HAT transactions in a causally consistent manner, AntidoteDB uses the Cure [4] system, allowing several read and write operations to be performed over multiple objects.

The AntidoteDB data store currently supports interactive transactions and static transactions. The former allows the execution of several data updates and reads in a single transaction before it commits. The latter consists in issuing a single operation where multiple objects are updated or read at once and in an atomic way; updates and reads are not mixed together in the same transaction.

3.1.1 System Architecture

AntidoteDB architecture is designed to be deployed in a group of geographically spread data centers (DC), each one containing a set of nodes. Each node within a cluster is uniquely identified in such a way that all nodes form a network ring, and each one takes over a subset of the objects key space [51].

Additionally, each node inside a data center is composed of four interconnected components, as illustrated in Figure 3.1 and summarized below:

- **Log**: this component is responsible for the persistent storage of updates to a log, typically stored on disk, and for providing a cache component for fast accesses to the log.
- **Materializer**: the materializer creates versions for each object accessed by a client.
- **Transaction manager**: the transaction manager is responsible for fully implementing a transactional protocol to manage transactions issued by clients to the system.

The protocol involves receiving requests, executing them on transactions, and replying to clients.

- **Inter-DC replication**: this module retrieves updates from the log and propagates updates across data centers asynchronously.

In this architecture, clients issue single operation invocations through *Remote Procedure Calls* (RPC) to a single node from the closest data center. Depending on the objects covered in the invocations, and since data is *sharded* inside a data center, only the nodes that own the objects (or *shards*) respond to requests for reads or writes over the objects. To invoke a group of related operations over multiple objects, clients must issue a transaction for an atomic execution of the operations.

3.2 Conflict-free Replicated Data Types

Conflict-free Replicated Data Types (CRDT) [72, 73] are special data types consisting of unique semantics that are suitable for storage systems that provide eventual consistency.

A CRDT is an abstract data type, with a well defined interface, designed to be replicated at multiple processes and exhibiting the following properties:

(i) Any replica can be modified without coordinating with another replicas;

(ii) When any two replicas have received the same set of updates, they reach the same state, deterministically, by adopting mathematically sound rules to guarantee state convergence.

The designs of these data types include *registers*, *sets*, *counters*, *graphs*, and more recently *bounded counters* [16]. All designs present concurrency semantics that define the outcome of applying concurrent operations that lead to divergence. The concurrency semantics are defined by the *add-wins* and *remove-wins* approaches. In an *add-wins* approach, under the concurrent execution of an add operation and a remove operation to the same data item, the add operation will prevail over the remove. In a similar scenario, a *remove-wins* approach states that the remove operation prevails over the add operation. These semantics allow a programmer to reason about the desired behavior to follow for her application. For instance, in a CRDT *set* that may be modified concurrently by two different clients, if one client adds a new version of one element in the set and another client issues a remove of the same element, in an *add-wins* approach the element must persist in the set; conversely, in a *remove-wins* approach the element is removed from the set.

These semantics are very important for automatic invariant preservation in replicated data stores, by not breaking data consistency and preserving relationships between data objects. CRDTs play an important role on the referential integrity feature of the Antidote Query Language system (described in Section 3.3), allowing relationships between records to be preserved even in the presence of concurrency.

CRDTs are implemented following two modes of operation: *state-based* and *operation-based* specifications. A *state-based* CRDT specification allows for full transfer of the state between replicas. That is, replicas propagate their current state asynchronously with other replicas in order to converge. Convergence is guaranteed if all possible states of a CRDT form a monotonic semi-lattice, that is, the sequence of all states is partially ordered and merging two sequences implies the computation of a *Least Upper Bound* (LUB) on both sequences. On an *operation-based* specification, replicas asynchronously propagate update operations to other replicas. To allow convergence across replicas, it is required for concurrent updates to be commutative.

CRDTs allow to design mixed-based specifications where both *operation-based* and *state-based* approaches are met in a single object. In addition, some CRDT designs, namely *Delta State Conflict-free Replicated Data Types* [5] (or δ-CRDT for short), vary the *state-based* approach by allowing small portions of the state of an object, named *delta-states*, to be transfered between replicas, to decrease the overhead of transferring full states in the network.

3.3 Antidote Query Language

The Antidote Query Language (AQL) [75] is a SQL-based interface for the AntidoteDB data store, presented in Section 3.1, and the most important prior work for this thesis. The main goal of AQL is to provide common database clients with a more familiar interface for querying a non-relational database. In the same way, AQL extends NoSQL with some additional features that are commonly used in SQL databases, such as:

- **Relational database statements**, unlike the key-value operations used in AntidoteDB, with table management statements (e.g. CREATE, DELETE), and record handling (e.g. INSERT, UPDATE);

- **Traditional SQL queries**, such as SELECT statements with basic WHERE clauses that include equalities over primary key columns;

- **Database assertions**, such as primary keys, referential integrity (foreign keys), and numeric invariants.

3.3.1 System Architecture

As depicted in Figure 3.2, AQL operates on a geo-replicated setting where multiple data centers (DC) are interconnected with each other and communicate through AntidoteDB instances. Each data center has a group of servers that store and replicate the data within the data center and among data centers. Each server implements an AntidoteDB instance that takes care of these issues. Additionally, each server contains an AQL module that

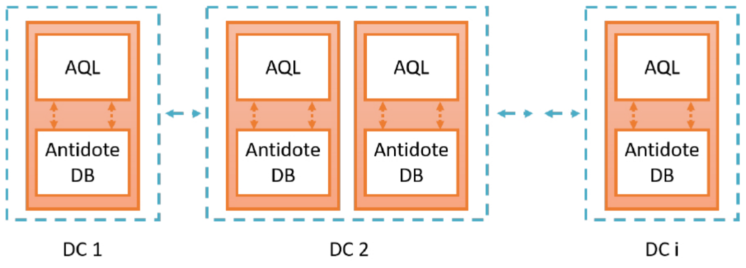


Figure 3.2: Architecture of AQL (taken from [75]).

talks directly to the AntidoteDB instance of that server. The AQL module essentially provides an interface for the client that directly connects to the key-value database provided by the AntidoteDB.

The interface provided by AQL allows the client to issue high-level operations to the system via *Remote Procedure Calls* (RPC). The high-level operations are database queries expressed as SQL-like statements that are parsed by the AQL module, translated to the AntidoteDB operation, and executed in a transaction. AQL replies back to the client with the properly parsed answer.

The AQL module also provides additional guarantees when performing client requests, namely additional requests that AQL needs to issue to other AntidoteDB instances when clients requests try to reach inaccessible data within the server.

3.3.2 Data Model

Since AQL is a SQL interface for a NoSQL data store, it needs to map data, from the key-value interface provided by AntidoteDB to the table abstraction introduced by AQL [15, 83]. As illustrated in Figure 3.3, AQL follows the representation of a table resorting to the *bucket* abstraction from the Antidote: the key of an AntidoteDB entry is composed of a triplet containing the key itself, the CRDT type, and the bucket name, which translates to the primary key column value, the CRDT *map* type, and the table name, respectively. Intuitively, each table row has associated a CRDT *map* object that represents the row of the table, containing a mapping between column names and column values.

For a better separation of concerns, a table's metadata is maintained separately from the table data itself in a distinct database entry. The metadata includes referential integrity constraints (e.g. foreign keys), column metadata (i.e. column names and respective types), table policy (i.e. where the table follows an *add-wins* approach or a *remove-wins* approach on concurrent operations), and other column policies.

3.3.3 Database Assertions

Following the features presented on the beginning of Section 3.3, AQL main concerns are in enforcing database invariants in the presence of concurrent updates on the database.

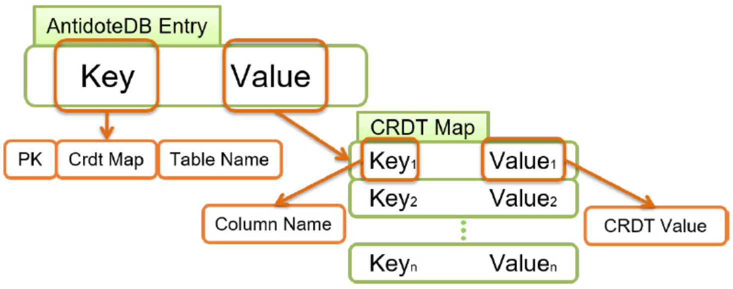


Figure 3.3: Representation of AQL data modeling (taken from [75]).

Primary keys. The primary key assertion, that says that no two records in the same table have the same primary key value, is addressed through the uniqueness property of AntidoteDB's keys used for data storage. AQL implements the record abstraction using CRDT *map* objects that are assigned with a key (i.e. the primary key). By itself, AntidoteDB already guarantees that no two key-value objects coexist with the same key in the data store, which enforces the primary key constraint.

Numeric invariants. Numeric invariants imply that a numeric column imposes a lower or an upper bound on the set of values admissible for that field. For instance, a table "Product" may have a column "Amount" that only accepts numeric values that are greater than or equal to zero. In this case, zero is the lower bound defined for column "Amount". Defining an upper bound on a column is straightforward and follows the same idea. For both cases, AQL resorts to a special type of CRDT named *bounded counter* [16] already provided by the CRDT library of AntidoteDB.

Referential integrity. Guaranteeing referential integrity on AQL was the main modeling and implementation challenge in the first version of AQL [75]. AQL builds relationships between tables like a SQL database does: some table columns may reference other columns from other tables that present the uniqueness property (e.g. primary key columns). This is done by applying the foreign key concept presented by SQL databases on AQL interface.

To counter the difficulty of keeping referential integrity on a weakly consistent data store, AQL resorts to visibility tokens, and table level and foreign key level rules. A *visibility token* is defined by AQL as a special attribute added to a record (i.e. at record-level) that states if the specified record is visible or not to clients. The tokens are identified by the values I and D, where I is issued to a record after an INSERT or UPDATE statement, and D is set as the visibility token of a record on a DELETE statement. AQL opts for not explicitly deleting records as they may be needed for a future reviving, on a scenario where one wants add operations (e.g. INSERT, UPDATE) to prevail over remove operations (e.g. DELETE).

AQL also implements a solution regarding cascade operations performed over database records, such as *delete* and *delete-cascade* operations that remove all records related with each other through the foreign key constraint. In addition, it implements implicit operations *touch* and *touch-cascade* to oppose for those delete operations.

Table and foreign key rules allow to define conflict-resolution policies between conflicting operations that violate referential integrity. For table rules, AQL defines the *add-wins* and *remove-wins* policies, and their functionalities are straightforward: the *add-wins* policy allows an insert or an update operation to prevail over a delete operation; the *remove-wins* policy allows a delete operation to prevail over inserts and updates. For foreign key rules, AQL defines the *force-revive* and *ignore-revive* rules, that state that all records that are related with each other (parent-records and child-records) must be preserved (resp. deleted) on concurrent execution of conflicting operations.

3.4 Summary

This chapter described the three most important prior works for this thesis.

Section 3.1 focuses on AntidoteDB, the key-value data store that supports some of the new functionalities proposed in this work. This section describes the main features already provided by this data store, and briefly explains its architecture.

Section 3.2 presents the Conflict-free Replicated Data Types (CRDTs), a set of data types specially designed with semantics for weakly consistent and replicated databases. This section presents some of their characteristics and designs. CRDTs are used in our work for supporting database invariants and index specifications.

Section 3.3 describes the Antidote Query Language, a SQL-like interface for the AntidoteDB data store. Its main structure and features are presented in this section, including the architecture and the data model of this system, and in more detail, how it enforces database invariants in a weakly consistent environment.

Chapter 4

System Design

In this work, we present an integrated solution for providing SQL functionalities in AntidoteDB NoSQL database, solving some limitations encountered on two systems.

In this chapter, we introduce the system model and design of the implemented solution for this thesis. The first section focuses on the system's architecture, while the second section details the system's design.

4.1 Architecture

The system's architecture of this work mostly represents a binding between the Antidote Query Language (AQL) [75] and the AntidoteDB [3, 8] database.

The higher view of the system is a group of geo-replicated data centers (DCs) scattered across the globe, each of which containing a set of nodes that communicate asynchronously with the nodes from other data centers for full data replication. In addition, inside each data center, data is *sharded* into several pieces such that each shard is stored in different nodes. Each node, or server from now on, runs an AQL module and an AntidoteDB instance that communicate with each other for exchanging data. Physically, both the AQL module and the AntidoteDB instance reside on the same physical machine. In this scenario, clients are spread out across the world and usually a client communicates with a single server from the closest data center. Typically, this communication is done through *Remote Procedure Calls* (RPCs), where the client issues database queries directly to the AQL module and the latter communicates with AntidoteDB for executing the operations.

This architecture is depicted on Figure 4.1. The blue dotted arrows on the figure represent remote calls to other servers or other data centers, and the yellow circles represent clients that communicate with the data centers.

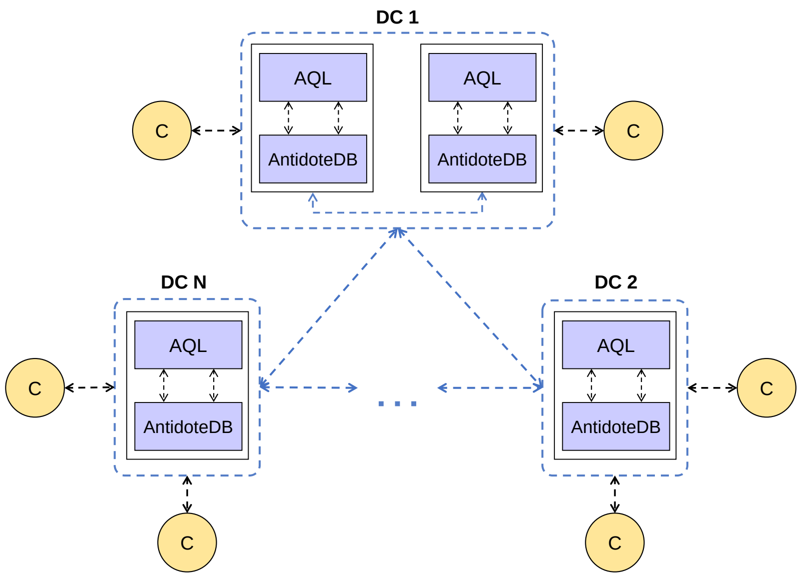


Figure 4.1: A top view of the whole architecture of the AntidoteDB and AQL. Enclosed on dotted rectangles are the servers that compose a data center. Each server contains a running AQL module and an AntidoteDB data node.

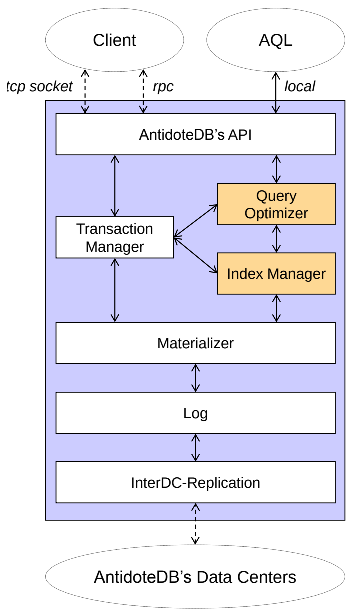
4.1.1 AntidoteDB's Architecture

An AntidoteDB instance, or data node, is composed of a group of interconnected components (or processes), where each component is responsible for a specific task on the database. These components are the inter-DC replication, the transaction manager, the materializer, and the log components, previously described on Section 3.1. In addition, this work added a query optimizer and an index manager to the AntidoteDB node architecture.

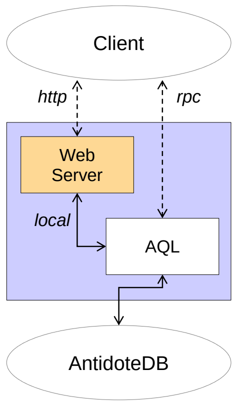
The **query optimizer** receives well-formed search queries from clients, interprets the queries, and computes the final result. This component will allow for more complex queries to be performed by the clients, and allows to optimize the object search in the database recurring to primary and secondary indexes, and to read partial database objects. Since the query optimizer needs to perform accesses to the database, it stands above and it is directly connected to the transaction manager component to which it reads whole or partial objects.

On the other hand, the **index manager** is responsible for managing primary and secondary indexes within the database. This component allows for reading indexes, and adding/deleting index entries, for instance. The index manager resorts to a trigger mechanism for updating automatically the indexes when a table row is updated or deleted, or when a secondary index is created. Unlike the query optimizer, the index manager is used internally within the AntidoteDB node and not directly by the client. It is placed alongside the transaction manager in the component architecture.

The new AntidoteDB architecture is depicted on Figure 4.2a, where it can be seen the connections between the new and the current components of the system. Additionally, the



(a) AntidoteDB node.



(b) AQL module.

Figure 4.2: Architectures of the two systems AntidoteDB and AQL. Dotted arrows represent remote calls; solid arrows represent local calls. Near each arrow is the type of communication between the two entities. The orange rectangles represent the added modules to the respective architectures.

AntidoteDB's API is also represented in the figure[1], depicting the possible connections initiated from the clients to the AntidoteDB instance.

4.1.2 AQL's Architecture

The AQL's architecture is very similar to the original architecture, presented in Section 3.3, yet with some minor changes. The AQL instance that resides on a server receives high-level operations from the clients which are processed and transformed into database operations, that are issued to the AntidoteDB node through local calls. The system is designed for AQL to be the intermediate between the client and the data, creating a whole interface to support the high-level operations. These high-level operations issued by clients are normally represented by SQL-like queries, as to be explained on the following sections.

The most common way for the clients to communicate with AQL is via *Remote Procedure Calls* (RPC). However, clients can also issue operations through a HTTP web server. The web server exports a very simple API composed of only a HTTP endpoint that allows to perform AQL queries over the database. This HTTP server does not fall within the scope of this thesis, and thus it will only be mentioned in this section.

The AQL architecture is depicted in Figure 4.2b.

[1]The AntidoteDB's API is not considered an isolated component of the system, and it is represented in the figure for completeness.

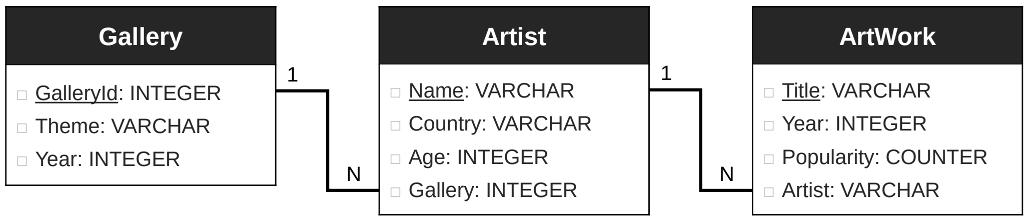


Figure 4.3: A simplistic database schema of an online art gallery. Primary key attributes are underlined.

4.2 Design

This section explains the overall system design for both AQL and AntidoteDB, including modifications performed on these two systems in order to meet the goals of this thesis.

When presenting each system's functionalities, it will be used some examples that concern a database schema of an online art gallery. The proposed schema is composed by three tables: *Gallery*, *Artist*, and *ArtWork*. These three tables, their columns, and their relationships are depicted on Figure 4.3. For simplicity, throughout the examples some of the presented columns for each table will be omitted. To identify a record/row, we will use the following notation:

$$table(primarykey)$$

, where *table* is the table name and *primarykey* is the primary key value that identifies the record within the table. For operations over the records, we will use the following notation:

$$operation(tname, pk, arg_1, \ldots, arg_N)$$

, where *operation* is the operation name (e.g. *insert*, *update*, or *delete*) and $(tname, pk, arg_1, \ldots, arg_N)$ is a list of one or more arguments to be applied by the operation on the record identified by *pk* on table *tname*.

4.2.1 Brief overview on AQL's syntax

The Antidote Query Language is a database query language, similar to SQL, that allows to issue high-level queries to the database AntidoteDB. Compared to its previous version [75], it suffered some changes to its syntax and added new functionalities worth mentioning in this section. Hence, for the sake of completeness, in this section we will indicate and briefly explain the complete syntax of AQL, giving more emphasis to the new added features.

When explaining each AQL statement, we will use square brackets to represent optional elements (i.e. that can be used or omitted), and curly brackets to represent a mandatory value amongst one of multiple possible values. Additionally, each sequence of statements must be separated by a semicolon (a single statement may end or not with a semicolon).

4.2.1.1 Create table statement

The CREATE TABLE statement, as the name indicates, allows to create a new table on the database, given a name, a table policy, a list of columns/attributes, and a partition column as an optional field. Listing 4.1 depicts the syntax for creating a table.

```
CREATE {UPDATE-WINS | DELETE-WINS} TABLE tablename (
    column1 type1 [constraint1],
    column2 type2 [constraint2],
    ...
    columnN typeN [constraintN]
) [PARTITION ON (column)];
```

Listing 4.1: CREATE TABLE statement.

Table policy. A table policy is determined by the tokens `UPDATE-WINS` and `DELETE-WINS`. A table policy represents a conflict-resolution policy (CRP) that allows the programmer to decide the default behavior on concurrent updates on a table: in this case, `UPDATE-WINS` stands for the *update-wins* policy and `DELETE-WINS` stands for the *delete-wins* policy. Since these are table policies, they determine the default behavior on the execution of concurrent conflicting updates on table data, i.e. on rows/records. Hence, the *update-wins* table policy determines that, when a record is inserted/updated concurrently with its deletion, the record will remain on the table. Whereas the *delete-wins* policy states that on the concurrent delete and insert/update of some record the record is deleted from the table. Section 4.2.3 further details table policies.

Columns. Each table column is defined by a triplet constituted by a column name, a data type, and a single constraint. The data types supported by our solution are presented in Table 4.1 and are defined as follows: the VARCHAR data type allows a column value to be defined in a textual representation, as a string enclosed between single quotes; the INTEGER (or INT) data type allows to assign integers to columns (much as the VARCHAR type); the BOOLEAN data type is used to assign boolean values to columns; and a COUNTER_INT data type defines a counter of integer numbers, defined with or without a minimum/maximum bound, that can be incremented or decremented by given amounts.

An important note is that AQL does not support null values on table columns.

Constraints. The constraint statement is optional for a column, but when used can be one of the following: primary key constraint, default constraint, check constraint, or foreign key constraint. The **primary key** constraint determines the uniqueness of a record on a table, which means that no two rows on a table can have the same primary key value. The primary key constraint can be expressed using the following syntax:

```
column type PRIMARY KEY
```

Table 4.1: Data types supported by AQL.

AQL data type	Description
VARCHAR	Common text/strings
INTEGER/INT	Integer numbers
BOOLEAN	Boolean values
COUNTER_INT	Integer counters (with or without bounds)

Some columns may need to define a default value (of the same type as the data type of the column itself) in case its value is missing on a record insertion. In that case, the **default** constraint is set for a column with the following syntax:

```
column type DEFAULT val
```

For COUNTER_INT typed columns, it is possible to establish a bound for the counter, i.e. a limit for which the value of the column cannot surpass upwards or downwards. This is called a **check** constraint. To define a check constraint for a counter typed column, the constraint is set as the listing bellow. The `comparator` is represented by an arithmetic comparison symbol, such as $<$, $\leq$, $>$, or $\geq$, and `val` is an integer number.

```
column COUNTER_INT CHECK (column comparator val)
```

Finally, **foreign key** constraints are determined much like as SQL, with the referenced table and the referenced column being set for an attribute. Note that AQL only supports references to primary key columns. Similarly to table policies, AQL allows to determine concurrent semantics for foreign key columns, through foreign key policies. A foreign key policy is very similar to a table policy, in that it determines the behavior of the concurrent execution of conflicting operations on records. There are three possible ways to define foreign key policies, by using the `UPDATE-WINS` token, the `DELETE-WINS` token, or by using none of these two for semantics with no concurrency allowed. At last, the user can determine a record's behavior when its parent records are deleted, with the notation `ON DELETE CASCADE`. When present on the foreign key specification, this notation states that a record must be deleted when its parent record is also deleted. When omitted, the default behavior is that a parent record cannot be deleted if exists one or more records that reference it [78]. The following listing represents the syntax for a foreign key constraint:

```
column type FOREIGN KEY [UPDATE-WINS | DELETE-WINS]
REFERENCES ref_table(ref_column) [ON DELETE CASCADE]
```

Section 4.2.2 states some of these column constraints as database invariants that need to be preserved in order for the database to remain consistent. Additionally, details about foreign key policies, record existence, and *delete-cascade* are further explained in Section 4.2.3.

Partitioning. The table partitioning option allows to set a table column as a partition column. This partition technique is also known as horizontal partitioning (described in Section 2.2.1.2). In practice, partitioning a table allows to split a table into several smaller tables that can be spread out to several locations on a data center. The partitioning technique in our system is further explained in Section 4.2.4. Note that, in the CREATE TABLE syntax (listing 4.1) the column enclosed in parenthesis before the words "PARTITION ON" must be one of the columns from the table's specification.

Example. Listing 4.2 presents the creation of table *ArtWork* with five columns, one of them being a partition column. All the three aforementioned constraints are also represented, including a reference to the row identified by *Name* on table *Artist*.

```
CREATE UPDATE-WINS TABLE ArtWork (
    Title VARCHAR PRIMARY KEY,
    Year INTEGER DEFAULT 2018,
    Popularity COUNTER_INT CHECK (Popularity >= 0),
    Artist VARCHAR FOREIGN KEY UPDATE-WINS REFERENCES Artist(Name)
) PARTITION ON (Year);
```

Listing 4.2: Example of a CREATE TABLE statement.

4.2.1.2 Create index statement

The CREATE INDEX statement allows to create a secondary index with a given name on a column from a given table, as the Listing 4.3 shows.

```
CREATE INDEX indexname ON tablename (columnname);
```

Listing 4.3: CREATE INDEX statement.

Currently, it is only possible to create an index over a single table column. In the future we are aiming to support composite indexes that comprise two or more table columns, giving a broader spectrum of options on the database schema and on querying process.

Secondary indexes themselves are a feature from our system and are further detailed in Section 4.2.5.

4.2.1.3 Insert, update, and delete statements

The INSERT, UPDATE, and DELETE statements are the common statements to update data within the database. Their behaviors are pretty similar to their correspondents from the SQL language.

Insert statement. The INSERT statement allows the database user to insert a row/record into a table, given a list of columns and the new values to be inserted. The list of columns declared in the statement may be omitted, and in that case all columns are considered for the insertion. On the other hand, all values from all columns must be declared, except

Table 4.2: Operations supported by AQL data types.

AQL data type	Operation	Description
VARCHAR	`Col = 'value'`	Set to `'value'`
INTEGER/INT	`Col = number`	Set to `number`
BOOLEAN	`Col = true`	Set to `true`
	`Col = false`	Set to `false`
COUNTER_INT	`Col = Col + inc`	Increment by `inc`
	`Col = Col + dec`	Decrement by `dec`

for default columns, where the value may be omitted. Listing 4.4 specifies the INSERT statement.

```
INSERT INTO tablename [(column1, column2, ..., columnM)]
VALUES (value1, value2, ..., valueN);
```

Listing 4.4: INSERT statement.

Update statement. This statement allows the user to issue a set of updates for one or more rows/records on a table, given a certain condition through a WHERE clause. If the condition is not specified, all rows from the table are considered and updated accordingly, given the set of updates; otherwise, the condition must be an equality on the primary key column. Each update consists in a given *operation* solely based on an equality and valid for the data type of the attribute in question, as stated on Table 4.2. The UPDATE statement is defined in Listing 4.5.

```
UPDATE tablename
SET operation1
AND operation2
...
AND operationN
[WHERE pkcolumn = pkvalue];
```

Listing 4.5: UPDATE statement.

Initially, updates to a foreign key column were not validated on the submission of this statement, and users could set invalid values (e.g. random or not existing values) as foreign keys on table rows. This behavior could break the referential integrity on the database and therefore, to prevent inconsistencies, each update to a foreign key is validated before updating the corresponding column on a row.

Delete statement. At last, the DELETE statement allows to delete a row from a specific table, given a condition. The condition is defined through a WHERE clause and only supports equalities on the primary key column. When specified, the row that satisfies the condition is deleted; otherwise, all rows of the table are deleted. The DELETE statement is presented in Listing 4.6.

```
DELETE FROM tablename
[WHERE pkcolumn = pkvalue];
```

Listing 4.6: DELETE statement.

It is important to note that a row can only be deleted if: a) the foreign key behaviors from this row and all rows that reference it, from other tables, are `ON DELETE CASCADE`, or b) the row has no other rows that reference it from other tables.

4.2.1.4 Select statement

The SELECT statement is the most common (yet complex) AQL statement, used for querying the database. In the previous version of AQL [75], the SELECT statement had the traditional SELECT clause, for declaring the columns to be projected; the FROM clause, where the source table is stated; and the simple WHERE clause, that, when not omitted, could only accept an equality over the table's primary key column.

For instance, querying the name of the art work whose *ArtId* equals to 1200 is achieved by the following SELECT statement:

```
SELECT Name FROM ArtWork WHERE ArtId = 1200;
```

To overcome this limitation, we present in this work a more complete version of the SELECT statement. The whole improvement of this query statement has its emphasis on the WHERE clause, since this clause limits the user querying and affects its usability. Hence, the new SELECT statement is represented on Listing 4.7.

```
SELECT {* | proj_column1, proj_column2, ..., proj_columnM}
FROM tablename
[WHERE column1 comparator1 value1
{AND | OR} column2 comparator2 value2
...
{AND | OR} columnN comparatorN valueN];
```

Listing 4.7: SELECT statement.

As an ordinary SQL-like SELECT statement, the WHERE clause is optional. That way, when it is omitted, the search comprises all rows from the table. On the other hand, when not omitted, all rows that satisfy the conditions declared in the WHERE clause are collected and returned.

In addition, the WHERE clause can hold several conditions separated by logical connectors `AND` and `OR`. In that way, it can support conjunctions and disjunctions, with conjunctions having precedence over disjunctions when evaluating a query. The conditions supported by the WHERE clause are triplets *column-comparator-value*, necessarily in this order: the *column* is a valid column name for the specified table; the *comparator* can be an equality/inequality (i.e. $=$ or $\neq$) or another valid comparison symbol (i.e. $<$, $\leq$, $>$, or $\geq$); and *value* is a value of the same type of the column.

Although not represented on the listing above, the syntax also supports conditions wrapped around parenthesis (using the '(' and ')' symbols), also named as *sub-conditions*, so that some conditions can take precedence (on the query evaluation process) over others, including over conjunctions.

For instance, the query below searches the names of all art works, from table *ArtWork*, whose artist is "Michael" *or* whose year is greater than or equal to 2010 and lesser than 2019:

```
SELECT Name FROM ArtWork
WHERE Artist = 'Michael' OR (Year > 2010 AND Year <= 2019);
```

The introduction of more complexity to SELECT queries consequently allows for more expressive queries, such as range queries or sub-query encapsulation. Together with well defined index data structures, it is possible to build an efficient query evaluator that uses index data structures. This will be further detailed in Section 4.2.6.

4.2.1.5 Administrative statements

The administrative statements supported in AQL are the SHOW TABLES, SHOW INDEX, and SHOW INDEXES statements. These three statements are represented in Listing 4.8 below.

```
SHOW TABLES;
SHOW INDEX [indexname] FROM tablename;
SHOW INDEXES FROM tablename;
```

Listing 4.8: Administrative statements.

The SHOW TABLES statement shows all tables within the database.

The SHOW INDEX statement allows to read the contents of an index from a given table. This statement supports an additional field, the index name, that allows to determine which index, from a specific table, is to be shown to the user. Only secondary index names are allowed on this statement, and in that case the contents of the secondary index are returned. Otherwise, if no index name is stated, the contents of the primary index from the specified table is returned by default.

The last administrative statement, the SHOW INDEXES statement, simply returns the list of all indexes created for a given table. If no indexes exist for that table, an empty list is returned.

4.2.1.6 Transaction statements

Alongside the already mentioned query statements, AQL supports statements for defining transactions. When executed individually, each query statement is executed in a single transaction and issued to the database. This implies starting a transaction, reading/writing objects, and closing the transaction every time a user queries the database.

This represented a limitation on the first version of AQL, since there was no mechanism to execute several instructions as an atomic unit.

Therefore, to overcome this limitation we introduce AQL transactions. As in the most common SQL databases, the user can start and commit/rollback a transaction and all its changes to the database. Between the start and the end of the transaction, the user can issue a set of instructions separated by semicolons. Listing 4.9 shows the simple AQL transaction syntax.

```
BEGIN TRANSACTION;
instruction1;
instruction2;
...;
instructionN;
{COMMIT | ROLLBACK} TRANSACTION;
```

Listing 4.9: Transactions syntax.

It is important to mention that when a given query (inside a transaction) throws an error (e.g. due to an exception or a fatal error), the transaction aborts automatically and all prior changes are rolled back. For safety, we kept the transaction mechanism to abort in this case, but other solutions may be taken into account (such as keeping current changes from the transaction and continue its normal execution, for instance).

4.2.2 Database Invariants

As in a regular relational database, some invariants can be preserved in order to keep the database consistent and coherent. Some of these invariants arise from table constraints, such as primary key, numeric check, and referential integrity constraints on table columns.

Although these invariants are easily maintained in SQL databases due to their strongly consistent nature, in a weakly consistent database keeping these invariants from being violated is not immediate. We briefly explain the three database constraints mentioned in the syntax above and how they are designed in our solution.

4.2.2.1 Primary key constraint

A primary key constraint allows to define a uniqueness property on a column of a given table. Exactly only one column from a table can be the primary key column.

The uniqueness property from this constraint states that all rows from a table have different values on this column, which in other words means that no two rows can have the same primary key value.

To maintain this database invariant, AQL resorts to the uniqueness property of the object keys introduced by AntidoteDB. That is, since AntidoteDB is a *key-value* data store, each *key* must be unique in the entire database, thus guaranteeing that no two objects in the database have the same key. In addition, AntidoteDB supports Conflict-free

Replicated Data Types [72, 73] as the building objects of the database, that allow two concurrent transactions to write to the same primary key, by merging both writes, and thus imposing no conflict in both operations. Section 5.4.1 explains in detail how AQL takes advantage of AntidoteDB keys to guarantee primary key constraints.

4.2.2.2 Check constraint

A check constraint is used to guarantee that the value of a numeric column does not go beyond a certain bound. More specifically, this constraint is suitable for COUNTER_INT data typed columns, as stated in Section 4.2.1.1.

This constraint is very useful for counter columns where must exist a control in the values of the column. For instance, restricting the number of enrolled players in a "room server" on an online game, given an established maximum number of players, or guaranteeing the number of sold tickets for a sport's match does not go below zero, are real examples for applying check constraints on database columns.

Asserting this constraint in a weakly consistent database is a challenge. Concurrent transactions that update a column that specifies this constraint can easily break the invariant established for that column. That is, though concurrent transactions may not violate this invariant locally, they may violate it after their updates merge, leading to an inconsistent database state.

Therefore, to overcome this anomaly AQL relies on a special CRDT type, named *bounded counter* [16], to hold data for check constraint typed columns. This CRDT has a bound by default that can be used as a building base for other usable bounds in practice, and allows concurrent updates to be executed on an object as long as the bound is not violated.

4.2.2.3 Referential integrity constraint

Referential integrity refers to maintain relationships among rows of different tables on a database. The interest of building a database that supports referential integrity arises from the database needs, such as data organization, hierarchy, and consistency.

Implementing this functionality in a weakly consistent database, such as in the AntidoteDB database, is challenging and can present some difficulties. Given that weakly consistent databases operate in a distributed system where the operations, especially conflicting ones, can be issued concurrently by different clients, violating referential integrity may happen very often. Breaking referential integrity can simply consist in a record pointing to a nonexistent record from other table, for example.

Given this topic is extensive, we leave a more detailed explanation of our solution of this functionality to later sections. Therefore, the next section describes a mechanism that resolves referential integrity in a weakly consistent and distributed context.

4.2.3 Referential Integrity Mechanism

The referential integrity functionality is mostly seen in relational or strongly consistent databases. One of the main focus of the AQL interface was to build an efficient mechanism that implements referential integrity on a weakly consistent database, in this case on the AntidoteDB database. The former version of this mechanism presents the SQL functionality with the default relationship among tables, but also presents specific algorithms that address concurrency and conflict resolutions for referential integrity violations.

However, the deployed mechanism lacks some functionality compared to the SQL's referential integrity, such as not giving the programmer the opportunity to choose whether or not to define the *delete-cascade* behavior for a foreign key, for instance. To address these limitations, we present in this work a revised version of this relational database functionality, that not only employs additional functionality, but also revises the internal mechanism to enforce referential integrity.

4.2.3.1 Visibility system

As previously explained in Section 3.3, AQL does not support the permanent deletion of records from the database. This is due to the need of always keeping data even after its deletion, given that it may be necessary to solve conflicts on referential integrity.

To illustrate the problem caused by explicit record deletion, consider the scenario depicted in Figure 4.4, with two clients operating over the same database schema composed by tables *Gallery* and *Artist*, each table with its corresponding initial state. This scenario corresponds to the concurrent execution of two operations, the insertion of an artist named "Michael", binded to an existing gallery whose identifier is "G1" – operation $insert('Michael', 'G1')$ –, and the explicit deletion of the art gallery "G1" – operation $delete('G1')$. Both modifications are executed locally in a transaction. When both operations are propagated and merged, the outcome is the existence of an artist pointing to a nonexistent gallery, which represents a referential integrity violation.

Hence, instead of managing table records in an explicit manner (e.g. delete a record explicitly), the AQL manages records in an implicit way, through conflict-resolution policies and visibility tokens, such that instead of the records being explicitly deleted, they exist in the database but are marked as not visible to the final user.

Conflict-Resolution Policies. Conflict-Resolution Policies (CRPs) are policies applied to tables that determine the correct states of their records upon concurrent conflicting operations. CRPs are divided into two major policy types: table policies and foreign key policies. **Table policies** are set at the table level and determine if a record is visible following an *update-wins* or a *delete-wins* approach: an *update-wins* approach states that when a record is inserted/updated and deleted concurrently, the record prevails; a *delete-wins* approach states that, under the previous scenario, a record will not prevail on the database.

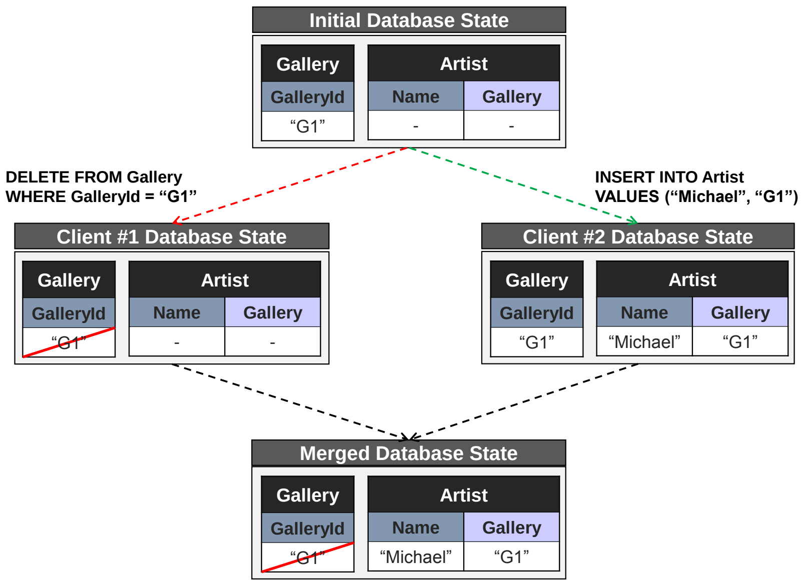


Figure 4.4: An example of how referential integrity can be violated on the concurrent execution of conflicting operations.

Foreign key policies determine the correct behavior of a parent table upon modification on the child table. There are three possible foreign key policies: the *update-wins*, the *delete-wins*, and the *no-concurrency* approaches[2]. An *update-wins* approach states that, when a child record is inserted/updated, its parent is revived; in the example of Figure 4.4, if the foreign key policy of table *Artist* was *update-wins*, the record *Gallery*('*G*1') would be revived in the final database state due to the insertion of *Artist*('*Michael*'). On the other hand, a *delete-wins* approach states that, when a parent record is deleted, it remains deleted and its children are denoted as deleted as well; in Figure 4.4, record *Gallery*('*G*1') would remain deleted and *Artist*('*Michael*') would become also deleted. At last, the *no-concurrency* approach states that the deletion of a parent record has to be exclusive compared to update operations on that record. That is, a certain record cannot be deleted and updated concurrently. However, it still allows concurrent updates to occur, such as adding references to an existing parent record. For instance, the operations depicted in Figure 4.4 could not be executed under this semantics.

Visibility tokens. For implementing the CRPs, AQL implements a mechanism that determines the visibility of each record through visibility tokens. A **visibility token** is a character that, together with a set of rules, determines if a record is visible or not, and always accompanies a record through a hidden column, called the record state. There are three visibility tokens – I, T, and D – and they work as follows:

- **Token I**: the state of a record has this value when an explicit insert or update operation – INSERT and UPDATE statements – is issued for the record. This token

[2]The *update-wins* and *delete-wins* correspond to the former *force-revive* and *ignore-revive* [75], respectively. The names have changed because we found these could be more intuitive for the user.

indicates that a record is *visible*;

- **Token *T***: this token is implicitly activated at the parent record when one of its child records is inserted or updated. This procedure only occurs with an *update-wins* foreign key policy. This token indicates that the record is *visible*;

- **Token *D***: a record takes this visibility token when it is explicitly deleted through a delete operation – DELETE statement – or when is implicitly deleted by a parent record's deletion. This latter case can only occur when the foreign key specifies the *delete-cascade* behavior. This token indicates that the record is *not visible*.

Designing visibility tokens consists in attaching a hidden attribute/column, represented by the name "*#st*", to each table and assigning the correspondent visibility token to each record, as exemplified in Figure 4.5. As stated on the list above, assigning a token to a record depends not only on the update operation being executed on that record or in a child record, but also may depend on the foreign key policy of a table. For instance, inserting or updating a child record only interferes in its parent record, i.e. only changes its parent record state, if an *update-wins* foreign key policy is defined in that case.

In this work, we present a solution that concerns the foreign key policies presented in this section, namely *update-wins*, *delete-wins*, and *no-concurrency*, to determine the necessity of performing the additional burden of updating a parent record visibility, and consequently, we define visibility calculation formulas to determine if a record is visible or not, given table and foreign key semantics. For the sake of simplicity, the visibility formulas and the algorithms that follow omit the computation of a row's state through the table and foreign key policies. That matter is further detailed in Section 4.2.3.5.

4.2.3.2 *Update-wins* approach

The *update-wins* foreign key policy is a conflict-resolution policy that gives advantage to insert and update operations when faced with concurrent deletes. From a parent record perspective, this means that when a delete operation is issued for the parent record, and an insert/update for (at least) one of its child records is issued concurrently, the parent record prevails. That is, the child record revives its parent through its update.

The action of reviving a parent, in this approach, is called *touch*. The action of *touching* the parent is analogous to assert its existence and is done by changing the parent's visibility state to the token T. That way, a parent having this state will always exist on a conflicting scenario regarding the *update-wins* foreign key policy, since token T prevails over the other tokens – I and D – independently of the table policy[3]. A more complete overview on visibility rules will be explained in a further section.

[3]The important aspect to retain from this fact is that state T prevails over the state D, which is the conflicting state that would make a record not visible. In addition, state T also prevails over state I.

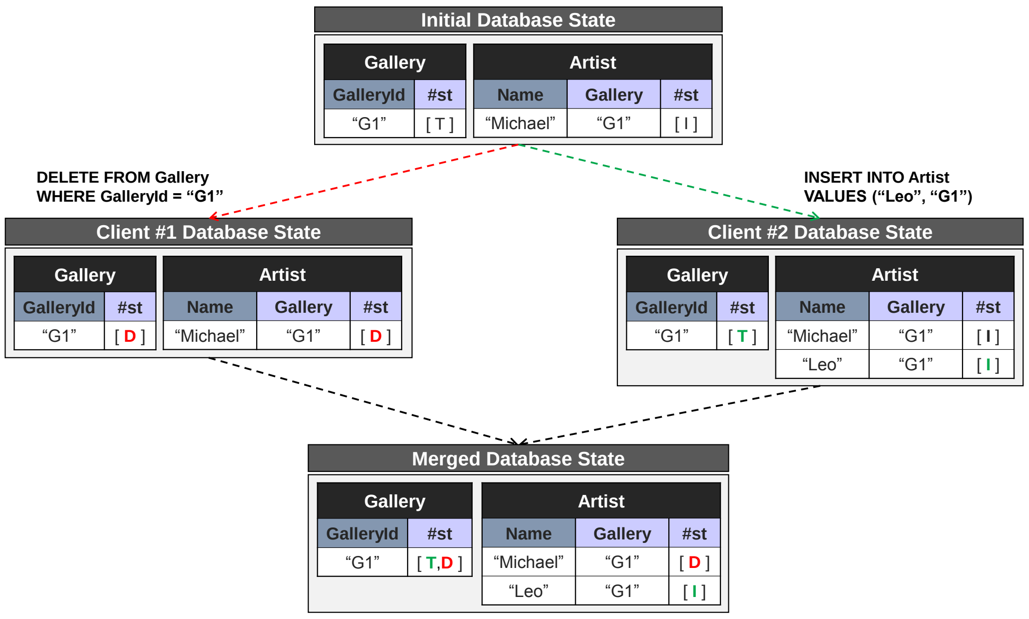


Figure 4.5: Example of a solution using an *update-wins* foreign key semantics and visibility tokens.

On the *update-wins* semantics, it may be necessary to climb up at the hierarchy until all parents are *touched* (unless the parents specify different foreign key semantics). Additionally, this semantics requires that, on a parent record deletion, all its children are individually deleted in cascade (assuming a *delete-cascade* foreign key behavior). This action is analogous to mark the parent row and its child rows as not visible, i.e. with the token *D*. The on delete cascade action may become very complex to perform given the amount of child tables (i.e. tables that depend on others) that exist and whose foreign key behaviors follow the *delete-cascade* behavior. However, if any of the child tables does not support cascading, this action is not possible.

This cascade solution guarantees that no former children of a deleted parent record is indirectly revived due to other child insertion or update. Figure 4.5 illustrates this solution. In this example, there are two clients concurrently issuing two operations on an initial database state, composed by records $Gallery(`G1')$ and $Artist(`Michael')$. The two operations consist in the deletion of gallery 'G1' – equivalent to $delete(Gallery, `G1')$ – and the insertion of artist 'Leo', that points to 'G1' – equivalent to $insert(Artist, `Leo', `G1')$ –, each by a different client. Changes induced by the delete are noted in red text; changes induced by the insert are noted in green text. The merged result from these operations under the *update-wins* semantics is the existence of gallery 'G1' and of artist 'Leo' on table *Artist*. As expected, when record $Gallery(`G1')$ is deleted on Client #1, record $Artist(`Michael')$ is deleted as well by the on delete cascade action, being the reason why this latter remains deleted at the final database state, given that the deletion of artist 'Michael' did not observe the *touch* action on gallery 'G1'. Concerning the visibility of the record $Gallery(`G1')$ on the final database state, the state of this record will contain both tokens T and D due to the concurrent update by the previous two operations. But, because the foreign key semantics is *update-wins*, T prevails over D.

Visibility calculation. In the presented solution, to determine the visibility of a record it suffices to look only into the record's state (through the hidden state column) and verify its visibility token is different than D. In any case, there is no need in this scenario to check the parent record's visibility since, following the natural behavior of this semantics, a record cannot exist if its parent does not. Therefore, for a certain record, *row*, this record is visible if the following SQL inequality satisfies:

$$row[\text{'}\#st\text{'}] <> \text{'}D\text{'}$$

In the condition above, *row* is represented as a map between attributes (or columns) and their respective values. We indicate the map name followed by a key inside square brackets to access a value from a map, given a key, i.e. $map[key]$. On this condition, we say that we are accessing the value in map *row* whose key is *'#st'*. Alternatively, *row* can be replaced by the record specification, i.e. $table(primarykey)$.

The next paragraphs describe the algorithms used by each database operation, select, insert, update, and delete upon these changes. The algorithms are written in a pseudo-code format that mostly uses SQL-like statements and other usual syntax [60].

Select. The pseudo-code for an AQL's SELECT statement is presented in algorithm 2. This SELECT statement is assumed to be executed on a table whose foreign-key policy is *update-wins*. In an *update-wins* scenario, querying the database through a SELECT statement is straightforward: the query is executed as is, with the particularity that it takes an additional condition that determines the records to be retrieved to the user. That condition is the execution of a function $is\_visible(row)$ that receives a row and determines its visibility through its state, returning true if it is visible or false, otherwise (algorithm 1). The execution of this function is appended to the end of the WHERE clause through a conjunction (lines 3 and 5). This function will be a common function for the algorithms of the remaining database operations.

Insert. The execution of an INSERT statement, under an *update-wins* approach, is more complex and involves performing additional queries besides the original insert operation. The pseudo-code for the INSERT operation is presented in algorithm 3. This statement receives a table name and list of pairs $(column, value)$, that correspond to the binding

Algorithm 1 Function that asserts the visibility of a record (*update-wins* version).

```
1: FUNCTION is_visible (row) RETURNS boolean
2: BEGIN
3:     RETURN row['#st'] <> 'D';
4: END;
```

Algorithm 2 Select operation on an *update-wins* approach.

```
    Requires: table : table name, cols : projection columns, conds : conditions.
 1: BEGIN
 2:     IF conds IS NOT EMPTY THEN
 3:         SELECT cols FROM table WHERE (conds) AND is_visible(*);
 4:     ELSE
 5:         SELECT cols FROM table WHERE is_visible(*);
 6: END
```

Algorithm 3 Insert operation on an *update-wins* approach.

```
    Requires: table : table name, values : list of pairs (col, value) to insert.
 1: BEGIN
 2:     DECLARE pk := (SELECT column_name FROM schema_info.key_column WHERE table_name = table);
 3:     DECLARE fkeys := (SELECT column_name, ref_table, ref_column_name FROM schema_info.fkey_column WHERE table_name = table);
 4:     IF fkeys IS NOT EMPTY THEN
 5:         FOREACH (col, ref_table, ref_col) IN fkeys DO
 6:             DECLARE cnt INT;
 7:             SELECT count(*) INTO cnt FROM ref_table WHERE ref_col = values[col] AND is_visible(*);
 8:             IF cnt = 1 THEN
 9:                 UPDATE ref_table SET '#st' = 'T' WHERE ref_col = values[col];
10:             ELSE
11:                 ABORT;
12:     INSERT INTO table(values.keys, '#st') VALUES (values.values, 'I');
13: END;
```

between the table columns and their corresponding values to be inserted into the database. For simplicity, default values (omitted in the statement) are assumed to be present in this list of pairs. The algorithm starts by checking if the foreign keys for the row to be inserted are valid. For this purpose, the table's foreign key specifications are collected through an administrative query (line 3, algorithm 3) and each specification is used to perform the validation (lines 5 to 11). The validation consists in asserting that a referenced primary key from a parent table exists, by means of a counter query (lines 6 and 7). If it validates, the parent row's state is updated accordingly (line 9). Otherwise, the statement is aborted[4]. If all foreign keys validate, the new row is inserted along with its new state (line 12).

Update. The UPDATE operation specification is depicted in algorithm 4. This AQL statement receives a table name, a list of pairs $(column, value)$ similar to the list from the INSERT algorithm, and the primary key value of the row that is to be updated. Additionally, it is required as a pre-condition that the row to be updated must exist and be visible. The remaining of the algorithm is pretty similar to the INSERT statement algorithm: the

[4]The "ABORT" action in the algorithm has a similar side effect to a transaction's rollback, canceling all modifications done until the moment of the abortion.

Algorithm 4 Update operation on an *update-wins* approach.

```
Requires: $table$ : table name, $values$ : list of pairs ($col, value$) to update, $pkey$ : primary key to update.
Pre-conditions: $pkey$'s row must be visible within table $table$.
1: BEGIN
2:    DECLARE $pk$ := (SELECT column_name FROM schema_info.key_column WHERE ref_table_name = $table$);
3:    DECLARE $fkeys$ := (SELECT column_name, ref_table, ref_column_name FROM schema_info.fkey_column WHERE table_name = $table$);
4:    IF $fkeys$ IS NOT EMPTY THEN
5:        FOREACH ($col$, $ref\_table$, $ref\_col$) IN $fkeys$ DO
6:            IF $values[col]$ IS NOT NULL THEN
7:                DECLARE $ref\_val$ := $values[col]$;
8:            ELSE
9:                DECLARE $ref\_val$ := (SELECT $ref\_col$ FROM $table$ WHERE $pk$ = $pkey$);
10:           DECLARE $cnt$ INT;
11:           SELECT count(*) INTO $cnt$ FROM $ref\_table$ WHERE $ref\_col$ = $ref\_val$ AND is_visible(*);
12:           IF $cnt$ = 1 THEN
13:               UPDATE $ref\_table$ SET '#st' = 'T' WHERE $ref\_col$ = $ref\_val$;
14:           ELSE
15:               ABORT;
16:   UPDATE $table$ SET $values$ AND '#st' = 'I' WHERE $pk$ = $pkey$;
17: END;
```

first step consists in traversing all foreign key specifications in order to assert reference existence (lines 5 to 15, algorithm 4), and the posterior update of the states of each parent row (line 13); and the second step is the update operation itself, where the row is updated with new values, including its state that is updated to assert its existence (line 16).

Delete. In an *update-wins* approach, the delete operation of AQL is presented in pseudo-code in algorithm 5. From the previously described statements, the delete operation is the most slow due to its recursive calls. This statement receives as arguments a table name and a primary key to delete. The algorithm starts by collecting, via a SELECT query, dependent foreign key specifications from tables that depend on the table received as argument (line 3, algorithm 5). Each foreign key specification contains the dependent table, the dependent column name (i.e. the column in the dependent table that is pointing to the input table), and the foreign key constraint. The goal of this list is to search all rows that depend on the row to be deleted and to determine if those rows can be deleted as well. Hence, the next step of the algorithm is to traverse the list of dependent foreign keys and proceed with the deletion of child rows (lines 5 to 13). Only the tables whose foreign key constraints are of the *delete-cascade* type can have their records deleted. Otherwise, the execution of this statement fails. If a foreign key constraint is of the type *delete-cascade*, the algorithm proceeds to fetch all rows of the dependent table that have their dependent columns pointing to the deleting row (line 8). Only after, those fetched rows are deleted by performing an update to their respective state, writing token D to their states (lines 9

Algorithm 5 Delete operation on an *update-wins* approach.

```
    Requires: table : table name, pkey : primary key to delete.
 1: BEGIN
 2:     DECLARE pk_col := (SELECT column_name FROM schema_info.key_column WHERE
    table_name = table);
 3:     DECLARE deps := (SELECT table_name, column_name, constraint FROM
    schema_info.fkey_column WHERE ref_table = table);
 4:     IF deps IS NOT EMPTY THEN
 5:         FOREACH (dep_table, dep_col, dep_const) IN deps DO
 6:             IF dep_const CONTAINS 'ON DELETE CASCADE' THEN
 7:                 DECLARE dep_pk := (SELECT column_name FROM schema_info.key_column
    WHERE table_name = dep_table);
 8:                 DECLARE dep_keys := (SELECT dep_pk FROM dep_table WHERE dep_col =
    pkey AND is_visible(*));
 9:                 FOREACH key IN dep_keys DO
10:                     UPDATE dep_table SET '#st' = 'D' WHERE dep_pk = key;
11:                     -- Recursive calls of the delete operation for this row...
12:             ELSE
13:                 ABORT;
14:     UPDATE table SET '#st' = 'D' WHERE pk_col = pkey;
15: END;
```

to 11). The algorithm proceeds recursively down on the hierarchy for all the intermediate deleted rows. At the end, the original row to be deleted has its state updated properly (line 14).

This algorithm states one important aspect already mentioned throughout this document: rows on AQL are not permanently deleted from the database; instead, they are marked as not visible. Marking a record as not visible consists in an UPDATE operation to be performed on a row, instead of an explicit DELETE operation. This small detail can be seen in lines 10 and 14, from algorithm 5.

4.2.3.3 *Delete-wins* approach

Unlike the *update-wins* policy, for a table defined with a *delete-wins* foreign key policy prevails the effects of delete operations over the effects of insert or update operations. That is, when a delete operation over a record is executed concurrently with the insertion/update of the same record, that record does not prevail in the database.

Similarly to the *update-wins* approach, the *delete-wins* approach performs the on delete cascade action by marking all child records as not visible, i.e. by updating their states to D each, at the moment a parent record is deleted.

Given our solution for the *delete-wins* approach, a new way of calculating the visibility of records is created. Similarly to *update-wins*, calculating a record's visibility must concern the record's own state. However, checking only the state of a record may not suffice in this case. Consider the example of Figure 4.6. This example is similar to the example from Figure 4.5, with the difference that it depicts the outcome of executing

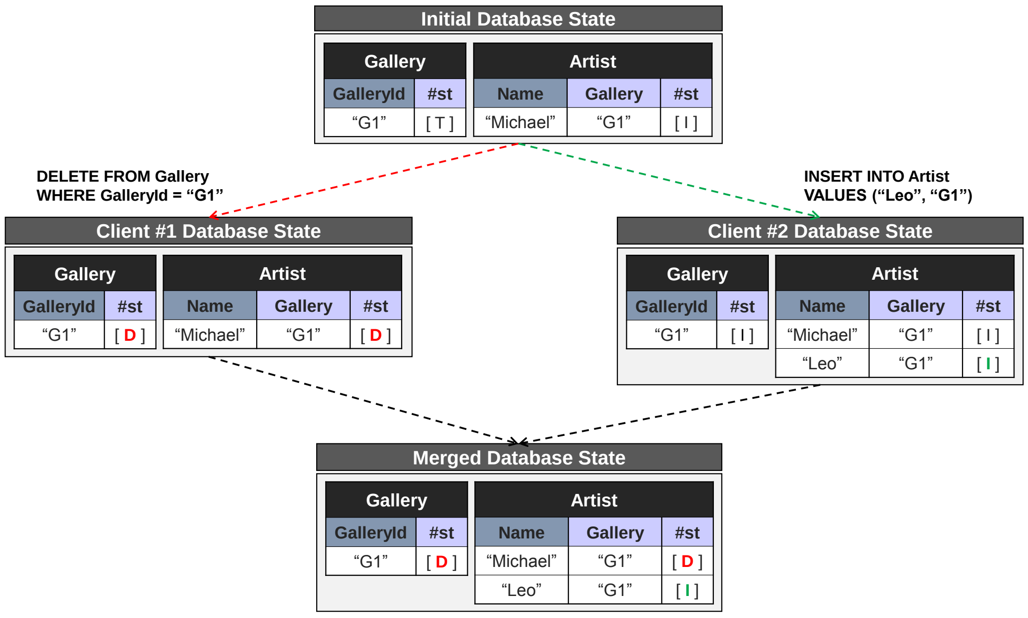


Figure 4.6: Example of the outcome of performing the operations from the example of Figure 4.5 on a *delete-wins* approach.

two concurrent operations on a *delete-wins* foreign key approach. From this figure, it is straightforward to conclude that record $Artist('Michael')$ is deleted because its state tells it directly; however, it is wrong to assume that record $Artist('Leo')$ exists just by looking to its state, because its parent record $Gallery('G1')$ is deleted and so $Artist('Leo')$ must be as well. This concludes that the parent record state also participates in the record visibility calculation.

The first intuition to assert a record's visibility is then to check first the record's own state and, if the record is visible, to check its parent visibility afterwards. Despite of its intuitiveness, this solution introduces an anomaly: inserting a previously existent record leads some old child records to be "revived". For instance, consider the example of Figure 4.6. Under the depicted database state, if record $Gallery('G1')$ were to be inserted, the record $Artist('Leo')$ would be revived because of its parent state.

To counter this problem we introduce a *record version*, an additional hidden column, represented by the name "*#vr*", that contains an integer that represents the version of a record within the table. This number increases after insert operations and is used to bind child records to specific versions of parent records. Binding the records to parent versions basically consists in appending an additional column to the table that stores pairs *value-version* for each record. Figure 4.7 depicts this new feature over the example of Figure 4.6, where parent versions are represented next to referenced primary keys, for simplicity. Now that record $Artist('Leo')$ references version 1 from $Gallery('G1')$, when the latter is inserted again, its version will increase to 2 and $Artist('Leo')$ no longer points to the most recent version of its parent. Note also that, in this case, the token D in the state of record $Gallery('G1')$ prevails over token T.

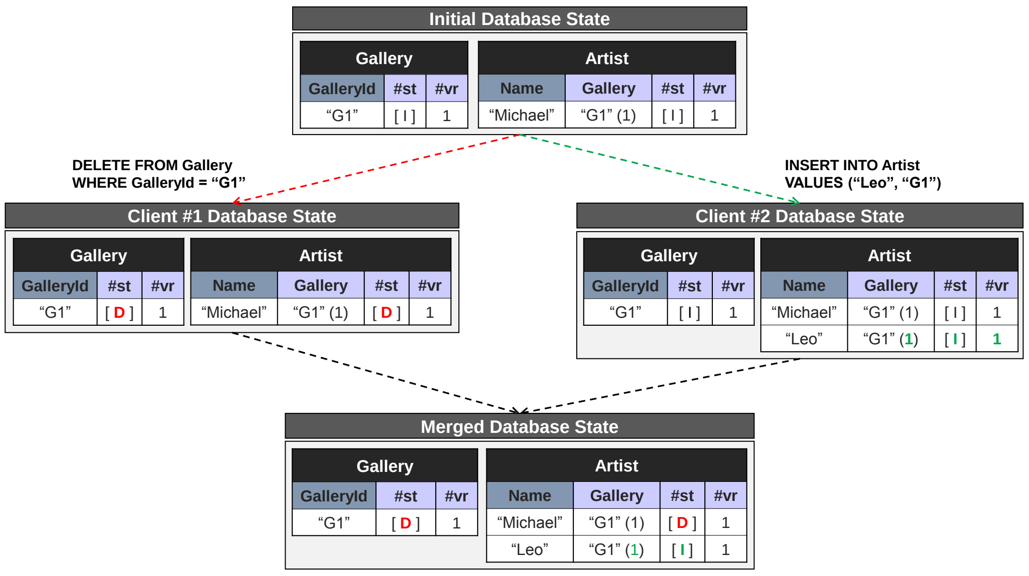


Figure 4.7: Example of the use of record versions on a *delete-wins* approach. In the foreign key columns, in parentheses, are represented the versions of the parent records.

Visibility calculation. Given the presented solution, the formula for computing the visibility of records changes for the *delete-wins* approach. As previously said, to determine a record's visibility one must take into account the own record's state and also its parent's visibility. Asserting the parent's visibility consists in first verifying that its version equals to the one the first record points to, and then verifying the parent record's state. All these conditions must be true to assert a record visibility which translate into a conjunction. Hence, given a record *row*, a parent record *prow*, a parent table name *ptable*, and the referenced column (from the parent table) *pcol*, the following condition must be satisfied to assert a record visibility:

$$row[\text{'}\#st\text{'}] <> \text{'}D\text{'} \wedge row[\text{'}\{ptable, pcol\}\text{'}] = prow[\text{'}\#vr\text{'}] \wedge prow[\text{'}\#st\text{'}] <> \text{'}D\text{'}$$

This formula hides some additional work, including recursive validations to be done in order to validate a record's visibility correctly. That is, the formula above concerns a single parent; the complete formula must take into account all parents of a given row, including parents of the parents. Specially, if at least one of the parents contains a *delete-wins* foreign key policy, it is necessary to apply this formula to this parent.

As the *update-wins* approach, the following paragraphs describe the algorithms for the common database operations, with the exception of the delete operation, because its algorithm is the same as the one presented in algorithm 5.

Select. The SELECT algorithm for the *delete-wins* approach is the same as the one from the *update-wins*, represented in algorithm 2: the algorithm consists in the SELECT query itself followed by the *is_visible* function, joined together by a conjunction. This time, the *is_visible* function that is used to fetch the rows that are visible to the user is the one

Algorithm 6 Function that asserts the visibility of a record (*delete-wins* version).

```
1:  FUNCTION is_visible ($row$, $table$) RETURNS boolean
2:  BEGIN
3:      IF $row$[‘#st’] <> ‘D’ THEN
4:          DECLARE $fkeys$ := (SELECT column_name, ref_table, ref_column_name FROM
        schema_info.fkey_column WHERE table_name = $table$);
5:          IF $fkeys$ IS NOT EMPTY THEN
6:              FOREACH ($col$, $ref\_table$, $ref\_col$) IN $fkeys$ DO
7:                  DECLARE $ref\_vcol$ := ‘{$ref\_table$,$ref\_col$}’;
8:                  DECLARE $ref\_val$ := $row[col]$;
9:                  DECLARE $ref\_vrs$ := $row[ref\_vcol]$;
10:                 DECLARE $cnt$ INT;
11:                 SELECT count(*) INTO $cnt$ FROM $ref\_table$ WHERE $ref\_col$ = $ref\_val$ AND
        ‘#vr’ = $ref\_vrs$;
12:                 IF $cnt = 0$ THEN
13:                     RETURN false;
14:         RETURN true;
15:     ELSE
16:         RETURN false;
17: END;
```

represented in algorithm 6. This function differs from the previous version (algorithm 1). Its functionality is based on the visibility calculation already stated in this section: the algorithm starts by checking the row's state and asserting its visibility, returning false in case it is not visible (line 3, algorithm 6); if the row is visible, in the next step the algorithm fetches all parent rows of the first row, validates the row's parent versions with its parents, and if the validation satisfies, the parents visibilities are asserted (lines 6 to 13). If any of these verifications fail, the function returns false; otherwise, the function returns true.

Insert. Performing an insert operation is slightly different from its previous mode of operation (algorithm 3). Unlike the alternative version, the *delete-wins* approach focuses on *record versions* and to use them to bind records with their parents, instead of touching explicitly the parents states, as represented in algorithm 7. The algorithm begins by collecting all versions from the parent rows of the row to be inserted; those versions are locally stored for later use (lines 6 to 14). If one of the parent rows does not exist (i.e. is not visible), the statement's execution fails. Otherwise, the algorithm proceeds with the row insertion: first, the row is inserted along with its new state I (line 15); then, it fetches any current version of the inserting row, increments that version, and updates the row's version with this new version (lines 16 to 21); at last, the newly added row is updated with its parents versions, previously collected (lines 22 and 23).

Update. At last, the UPDATE statement on a *delete-wins* approach has its mode of operation reflected in algorithm 8. Unlike its alternative version, the presented algorithm is pretty simple: the relevant action performed by an update operation consists in the

Algorithm 7 Insert operation on a *delete-wins* approach.

```
    Requires: table : table name, values : list of pairs (col, value) to insert.
 1: BEGIN
 2:     DECLARE pk := (SELECT column_name FROM schema_info.key_column WHERE table_name = table);
 3:     DECLARE fkeys := (SELECT column_name, ref_table, ref_column_name FROM schema_info.fkey_column WHERE table_name = table);
 4:     DECLARE fk_versions;
 5:     IF fkeys IS NOT EMPTY THEN
 6:         FOREACH (col, ref_table, ref_col) IN fkeys DO
 7:             DECLARE cnt INT;
 8:             DECLARE ref_vrs INT;
 9:             SELECT count(*) INTO cnt, '#vr' INTO ref_vrs FROM ref_table WHERE ref_col = values[col] AND is_visible(*);
10:             IF cnt = 1 THEN
11:                 DECLARE ref_vcol := '{ref_table,ref_col}';
12:                 fk_versions := fk_versions UNION (ref_vcol, ref_vrs);
13:             ELSE
14:                 ABORT;
15:     INSERT INTO table(values.keys, '#st') VALUES (values.values, 'I');
16:     DECLARE curr_vr INT;
17:     SELECT '#vr' INTO curr_vr FROM table WHERE pk = values[pk];
18:     IF curr_vr IS NULL THEN
19:         UPDATE table SET '#vr' = 1 WHERE pk = values[pk];
20:     ELSE
21:         UPDATE table SET '#vr' = (curr_vr + 1) WHERE pk = values[pk];
22:     FOREACH (ref_vcol, ref_vrs) IN fk_versions DO
23:         UPDATE table SET 'ref_vcol' = ref_vrs WHERE pk = values[pk];
24: END;
```

Algorithm 8 Update operation on a *delete-wins* approach.

```
    Requires: table : table name, values : list of pairs (col, value) to update, pkey : primary key to update.
    Pre-conditions: pkey's row must be visible within table table.
 1: BEGIN
 2:     DECLARE pk := (SELECT column_name FROM schema_info.key_column WHERE ref_table_name = table);
 3:     UPDATE table SET values AND '#st' = 'I' WHERE pk = pkey;
 4: END;
```

update itself, where a row is updated with a given list of values, and the update of the row's state to *I*, for asserting its visibility (lines 2 and 3).

4.2.3.4 *No-concurrency* approach

The *no-concurrency* foreign key policy differs from the previous two policies in the way that prevents concurrent conflicting operations to be executed over table rows.

More specifically, a *no-concurrency* approach precludes the concurrent execution of an update and a delete on a parent row. In the context of the referential integrity functionality, this is analogous to state that a parent row cannot be deleted concurrently with the insert or update of a row that references it. Note that when a row is inserted, it is determined which rows from other tables are referenced by the first row through their respective primary keys. As such, an invisible *pointer* is made for each parent row at the moment of the insertion, therefore the need to restrict the access to the parent rows.

Our approach to control concurrent accesses to parent rows is based on the use of *multi-level locks* (MLL) [47]. This approach was inspired by the invariant-repair techniques presented in the Indigo system [17]. In our solution, MLLs are used to enforce mutual exclusion on the deletion of rows. Mutual exclusion is achieved through the use of *exclusive locks* that are acquired by a transaction at the moment of the deletion of a row and maintained until the end of the transaction. On the other hand, a transaction that updates a row that references external parent row acquires a *shared lock* for each parent row, which are released at the end of the transaction. Acquiring a *shared lock* makes it possible for several transactions to make updates simultaneously on rows that reference the same parent row.

Visibility calculation. The *no-concurrency* semantics uses visibility tokens in a very simple manner. Since this semantics presents stronger consistency properties, it does not allow concurrency anomalies when performing operations over related tables. Hence, it only relies on visibility tokens I and D (with behaviors already described at Section 4.2.3.1) to assert record visibility. As such, a record is visible if it satisfies the following condition:

$$row[\text{'}\#st\text{'}] <> \text{'}D\text{'}$$

In summary, in *no-concurrency* semantics asserting the visibility of a record only consists in certifying that the token computed from the record's state is not D.

The following paragraphs present the algorithms for the database operations when following *no-concurrency* semantics. In general, the algorithms for this semantics are very similar to the algorithms for the *update-wins* approach. For that reason, we give a brief explanation about only what changed in each algorithm to support the *no-concurrency* semantics, given each respective pseudo-code. The SELECT operation and the *is_visible* function are the same as the ones from the *update-wins* semantics.

Insert. Based on algorithm 3, the INSERT operation's pseudo-code is presented in algorithm 9. The major changes in this algorithm are found in the code inside the *for each* loop and consist in acquiring a *shared lock* and validating the referenced rows (lines 5 to 14, from algorithm 9). Acquiring a *shared lock* is done in a protected environment, in a *try-catch* scope, to prevent an exception to be thrown by the database when a lock is not available (lines 7 to 10); if this is the case, because other transaction is using the lock, the execution of the algorithm aborts. This is a straightforward approach to prevent

Algorithm 9 Insert operation on a *no-concurrency* approach.

```
Requires: $table$ : table name, $values$ : list of pairs ($col, value$) to insert.
1:  BEGIN
2:      DECLARE $pk$ := (SELECT column_name FROM schema_info.key_column WHERE table_name = $table$);
3:      DECLARE $fkeys$ := (SELECT column_name, ref_table, ref_column_name FROM schema_info.fkey_column WHERE table_name = $table$);
4:      IF $fkeys$ IS NOT EMPTY THEN
5:          FOREACH ($col$, $ref\_table$, $ref\_col$) IN $fkeys$ DO
6:              DECLARE $ref\_pk$ := $values[col]$;
7:              TRY
8:                  ACQUIRE SHARED LOCK FOR $ref\_pk$;
9:              CATCH lock not available
10:                 ABORT;
11:             DECLARE $cnt$ INT;
12:             SELECT count(*) INTO $cnt$ FROM $ref\_table$ WHERE $ref\_col$ = $ref\_pk$ AND is_visible(*);
13:             IF $cnt = 0$ THEN
14:                 ABORT;
15:     RELEASE SHARED LOCKS;
16:     INSERT INTO $table$($values$.keys, '#st') VALUES ($values$.values, 'I');
17: END;
```

Algorithm 10 Update operation on a *no-concurrency* approach.

```
Requires: $table$ : table name, $values$ : list of pairs ($col, value$) to update, $pkey$ : primary key to update.
Pre-conditions: $pkey$'s row must be visible within table $table$.
1:  BEGIN
2:      DECLARE $pk$ := (SELECT column_name FROM schema_info.key_column WHERE ref_table_name = $table$);
3:      DECLARE $fkeys$ := (SELECT column_name, ref_table, ref_column_name FROM schema_info.fkey_column WHERE table_name = $table$);
4:      IF $fkeys$ IS NOT EMPTY THEN
5:          FOREACH ($col$, $ref\_table$, $ref\_col$) IN $fkeys$ DO
6:              IF $values[col]$ IS NOT NULL THEN
7:                  DECLARE $ref\_val$ := $values[col]$;
8:              ELSE
9:                  DECLARE $ref\_val$ := (SELECT $ref\_col$ FROM $table$ WHERE $pk = pkey$);
10:             TRY
11:                 ACQUIRE SHARED LOCK FOR $ref\_val$;
12:             CATCH lock not available
13:                 ABORT;
14:             DECLARE $cnt$ INT;
15:             SELECT count(*) INTO $cnt$ FROM $ref\_table$ WHERE $ref\_col$ = $ref\_val$ AND is_visible(*));
16:             IF $cnt = 0$ THEN
17:                 ABORT;
18:     RELEASE SHARED LOCKS;
19:     UPDATE $table$ SET $values$ AND '#st' = 'I' WHERE $pk = pkey$;
20: END;
```

deadlocks from occurring. On the other hand, validating the referenced rows simply consists in asserting their existence. When all referenced rows are validated, the algorithm releases all locks (line 15) and creates the row into the database (line 16).

Update. The pseudo-code for the algorithm of the UPDATE operation, under *no-concurrency* semantics, is introduced in algorithm 10. Similarly to the INSERT operation, this operation has changed on the foreign key validation, requiring now to acquire a *shared lock* for each parent row the updating row references to (lines 10 to 13), and to assert the existence of each parent row (lines 14 to 17). At the end, all *shared locks* are released and the row is updated (lines 18 and 19, respectively).

Delete. At last, the algorithm for the DELETE statement is depicted in algorithm 11. The first aspect to take into account from this algorithm is that it starts by acquiring an *exclusive lock* for the primary key of the row to be deleted (line 2, from algorithm 11). To

Algorithm 11 Delete operation on a *no-concurrency* approach.

```
    Requires: table : table name, pkey : primary key to delete.
 1: BEGIN
 2:     acquire_elock(pkey);
 3:     DECLARE pk_col := (SELECT column_name FROM schema_info.key_column WHERE table_name = table);
 4:     DECLARE deps := (SELECT table_name, column_name, constraint, policy FROM schema_info.fkey_column WHERE ref_table = table);
 5:     IF deps IS NOT EMPTY THEN
 6:         FOREACH (dep_table, dep_col, dep_const, dep_policy) IN deps DO
 7:             IF dep_const CONTAINS 'DELETE CASCADE' THEN
 8:                 DECLARE dep_pk := (SELECT column_name FROM schema_info.key_column WHERE table_name = dep_table);
 9:                 DECLARE dep_keys := (SELECT dep_pk FROM dep_table WHERE dep_col = pkey AND is_visible(*));
10:                 FOREACH key IN dep_keys DO
11:                     IF dep_policy CONTAINS 'NO-CONCURRENCY' THEN
12:                         acquire_elock(key);
13:                     UPDATE dep_table SET '#st' = 'D' WHERE dep_pk = key;
14:                     -- Recursive calls of the delete operation for this row...
15:             ELSE
16:                 ABORT;
17:     RELEASE EXCLUSIVE LOCKS;
18:     UPDATE table SET '#st' = 'D' WHERE pk_col = pkey;
19: END;

20: FUNCTION acquire_elock (lock) RETURNS void
21: BEGIN
22:     TRY
23:         ACQUIRE EXCLUSIVE LOCK FOR lock;
24:     CATCH lock not available
25:         ABORT;
26: END;
```

acquire a lock, the function *acquire_elock* is called with the lock identifier as an argument (lines 20 to 26). From this point, the algorithm resumes its normal operation until it updates the child rows of the deleting row (lines 6 to 16). As an additional condition for the algorithm, before a set of child rows have their states changed to D, the algorithm acquires as many *exclusive locks* as the number of child rows of the deleting row, provided that the foreign key policy of these rows is the *no-concurrency* policy (lines 11 and 12). When all child rows are deleted, all *exclusive locks* previously obtained are released and the row is deleted (lines 18 and 19).

4.2.3.5 Visibility rules

This section finishes the explanation about the referential integrity mechanism and focuses on defining visibility rules.

Both table policies and foreign key policies establish an order among the visibility tokens when calculating the visibility of a record. Starting by analyzing the table policies, the *update-wins* table policy states that insert/update operations prevail over delete operations considering the same record. Analogous to visibility tokens, this means that the I token prevails over the D token, such that:

$$D < I$$

, where $<$ is a prevalence relation between tokens, where $a < b$ states that b prevails over a. On the other hand, a *delete-wins* table policy states that delete operations prevail over insert or update operations, that is, the visibility token D prevails over the I token, such that:

$$I < D$$

Taking a look into the foreign key policies, the *update-wins* foreign key policy states that a parent record prevails on the database after being *touched* by the execution of an insert operation on its child record. Analogous to visibility tokens, this statement means that token T must prevail over any other visibility token. That is, for any visibility token t other than T, t has less prevalence than T, such that:

$$\forall t \in \{I, D\}, t < T$$

This prevalence is necessary to prevent referential integrity from being broken.

On the contrary, the *delete-wins* foreign key policy states that deleting a parent record forces the deletion of its child records as well, which keeps preventing a parent record from being revived. In this case, in what concerns visibility tokens, the *delete-wins* foreign key policy only relies on the table policies *update-wins* and *delete-wins* when establishing prevalence rules.

Similarly, and as stated on Section 4.2.3.4, the *no-concurrency* foreign key policy only relies on table policies to calculate the state of a row. Therefore, it applies the token prevalence rules from table policies.

Table 4.3: All combinations of AQL's visibility tokens, given table and foreign key policies.

Table policy	F.K. policy from child table(s)			
	UW(*)	DW	NC	None
UW	$D < I < T$	$D < I$	$D < I$	$D < I$
DW	$I < D < T$	$I < D$	$I < D$	$I < D$

Given these precedence rules, we identified eight ways of ordering visibility tokens, given several combinations of table policies and foreign-key policies. The eight combinations are represented in Table 4.3. Note that only four combinations exist, as two rules repeat in the table.

Once more, the definition of the visibility rules approaches the possible ways of using table policies on a table, when its child tables define one or none of the three possible foreign key policies at a time. Marked on the table with an asterisk (*) is the *update-wins* foreign key policy defined on child tables, that when combined with a *delete-wins* foreign key policy (at the source table), violates referential integrity.

4.2.3.6 Previous solutions and discussion

In the previous work [75], AQL resorted to cascading techniques to not allow database inconsistencies in referential integrity. These cascading techniques consisted in indirectly marking all records as inserted or as deleted after a parent record's visibility was asserted (i.e. changed to T) or not asserted (i.e. changed to D), respectively. These two techniques corresponded to the *touch-cascade* and *delete-cascade* techniques, respectively.

The *touch-cascade* technique was an action triggered by the *touch* action on a parent record and consisted in touching back all child records that referenced that parent record, as means of asserting their visibility. Touching back all child records consisted in *touching* the records at the table level, by means of a hidden table created to embrace all records that reference a specific parent record, even the concurrently added child records. The goal of the hidden table was to store the *touch-cascade* state, the token TC, and to later use it for the visibility calculation.

On the other hand, the *delete-cascade* technique presented the inverse behavior of the *touch-cascade*: this technique was triggered whenever a parent record was deleted, i.e. when its state changed to D, resulting in the posterior deletion of its child records. Similar to *touch-cascade*, the child records were deleted indirectly via a hidden table that relates them with their parent, by changing the table state to DC instead of TC.

Despite its intuitiveness and ease of programming, opting for this solution to deal with concurrency on referential integrity later turned out to present some problems.

Because of its semantics, the *touch-cascade* would revive some records that, in theory, should not exist. Consider the example of Figure 4.8, that illustrates a similar scenario to the one shown in Figure 4.5, with the same database schema, same initial state and same operations. Additionally, the example introduces hidden tables (depicted in orange) and

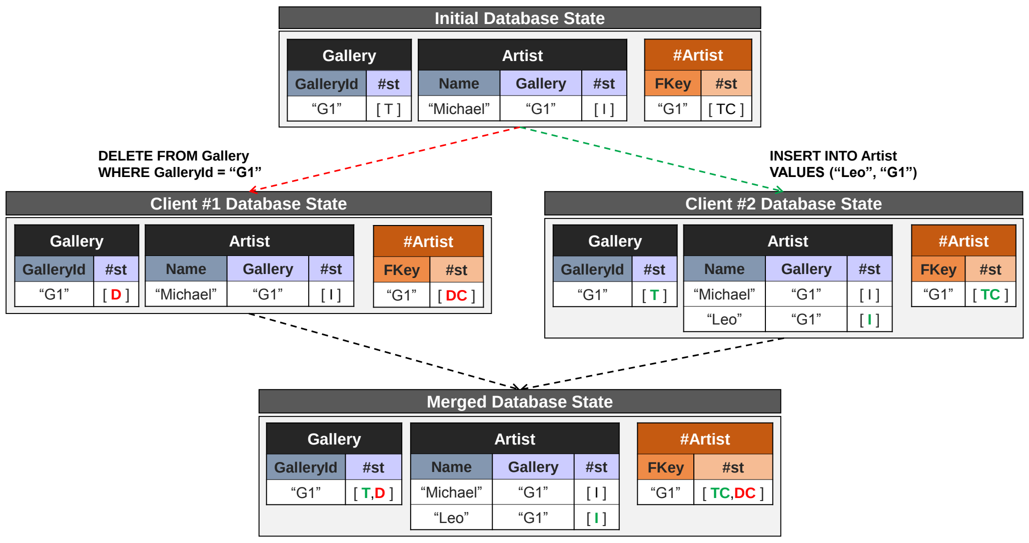


Figure 4.8: Example of the *touch-cascade* technique, including the use of visibility tokens. Hidden columns and tables are prefixed with a '#' character.

tokens TC and DC. Given the final merged state of the database, after the execution of the two operations, the problem of the *touch-cascade* technique arises at the visibility checking for the records on this point: after the $delete(\text{'}G1\text{'})$ operation takes effect, the record $Artist(\text{'}Michael\text{'})$ should continue to be deleted, i.e. should not be visible; instead, the *touch-cascade* action turns this record into visible (because at the hidden table $\#Artist$, TC prevails over DC), which is not correct since its visibility was influenced by an operation on another record, more specifically, the $insert(\text{'}Leo\text{'}, \text{'}G1\text{'})$ operation.

Updating a record in this solution does not incur on a latency cost when compared to our solution. This is because the former solution necessarily does fewer write operations to update record states on a cascade scenario, since updating a record's state on this scenario consists in updating a hidden table (that could cover a very large set of records on a table), which corresponds to a single write operation.

However, despite its write efficiency, we decided to abandon this solution for the fact that we aim for faster read operations (i.e. SELECT statements). For this purpose, we are willing to sacrifice the performance of delete cascading operations. On a database system where read operations prevail over write operations (which typically occurs in most current geo-replicated databases), this is seen as a good trade-off.

4.2.4 Table Partitioning

Partitioning a table in AQL consists in splitting it into smaller sub-tables, such that each sub-table contains a set of rows with the value of a column in common. This partitioning technique is known as horizontal partitioning (explained in Section 2.2.1.2), which groups rows from a particular table by attribute/column.

In our database system, splitting a table is easily done by taking advantage of the bucket abstraction of AntidoteDB. Recalling Section 3.3, AQL uses database keys to map

AQL records to database objects, where a database key is composed by a primary key, an object type, and the table name (Figure 3.3). The third parameter of the database key is a bucket name (analogous to a table name), where a bucket is a collection of keys that can be partially replicated in the database.

Therefore, by using buckets to identify which table a row belongs to, partitioning a table in AntidoteDB only consists in creating several different buckets, such that each bucket identifies the original table (i.e. the topmost table to be partitioned) and a value from the partition column to which a group of rows points to after the partitioning. That way, a unique bucket name composed of these two columns is created and rows are spread to the most adequate bucket names. Note that, with this design, the partition column becomes immutable, since changing this column's value on a row would lead to generate a new key in the database for that row, which could lead to inconsistencies and to increase the database size even more (because AntidoteDB does not support the permanent deletion of objects).

This design of table partitioning allows to prepare AntidoteDB for future implementations of partial replication strategies, by allowing objects (rows, in this case) to be placed on specific locations within the geo-replicated system. For instance, an idea for implementing partial replication would be having data objects to be replicated in small subsets of data centers, in strategic places around the globe, by following a certain criteria.

4.2.5 Indexing System

Like some popular NoSQL databases [27, 52, 69], the AntidoteDB database uses indexing mechanisms to ease the querying processing and to get performance gains. To that end, we present a new indexing system for the AntidoteDB, including a brief overview on this system and the specifications of index data structures.

4.2.5.1 Overview

The indexing system is a separated module in the AntidoteDB architecture that is responsible for managing indexes and handling data requests to index data structures. By managing indexes, we mean data insertion/deletion in the indexes data structures induced by actions on the database tables. By data requests, we mean requests made by upper layers in the database architecture, including from the query optimizer layer (Figure 4.2a).

Each table in the database has its corresponding primary index and data is inserted in the index upon record insertion in the table. The former indexing solution stated that the AQL system was responsible for managing primary indexes and for sending instructions to AntidoteDB for writing and reading index data in the form of database objects. We decided to pass this functionality to the database and now all the index management is performed in AntidoteDB, more properly in the indexing layer.

Similarly to primary indexes, secondary indexes are managed within the database, including their creation and update (upon record creation) on the indexing layer. However, unlike primary indexes, secondary indexes are created on demand by the database user according to her needs. This kind of index presents a cost on write operations, and thus must be used carefully.

In both cases, the indexing layer receives requests for reading or writing data and is responsible for interpreting those requests and converting then to database requests. As represented in Figure 4.2a, this layer may communicate to either the transaction manager or directly to the materializer layer. The indexing layer chooses the most adequate layer to read/write data, depending on whether the data request is received from the query optimizer or the transaction manager, respectively.

4.2.5.2 Index specifications

Primary and secondary indexes are the main tools of the indexing system, being important data structures that organize data in an efficient manner, with the intention of improving the query optimizer.

Since AntidoteDB is a *key-value* data store, we take advantage of its semantics and organize index data in native database objects, i.e. in operation-based Conflict-free Replicated Data Type (CRDT) objects [72].

Both specifications follow the same logic when storing data, by using data structures that bind indexed column values with primary keys. The idea consists in mapping raw column values to database keys to overcome scalability issues (due to predetermined key naming) and also to take into account the table partitioning (explained in Section 4.2.1.1), that makes necessary for keeping track of the keys of partitioned rows.

The former AQL indexing solution supported only primary indexes built on CRDT *grow-only map* objects. A *grow-only map* is a data structure whose payload is a *key-value* map, where the *key* can be a value of any type, and *value* is an embedded CRDT object. Our index specifications follow an identical design by using map alike data structures that keep raw column values and store database keys in CRDT objects. The CRDT objects are of the same type as the column type from the indexed table, in order to keep the index data consistent with the table rows; index consistency is achieved by satisfying the same concurrent semantics of the table.

The following two paragraphs refer to and explain each index data structure in terms of pseudo-code. The pseudo-code follow the syntax presented in [72].

Primary indexes. In order to store primary index data in the database, AntidoteDB uses CRDT objects, more specifically primary index CRDT objects that follow the specification presented in algorithm 12. This new CRDT has a simple payload: a sorted set, named *index tree*, with the behavior of a map that stores (*pkey*, *state*) pairs, where *pkey* is a primary key value and *state* is an embedded CRDT object of type T that stores the database

Algorithm 12 Operation-based primary index.

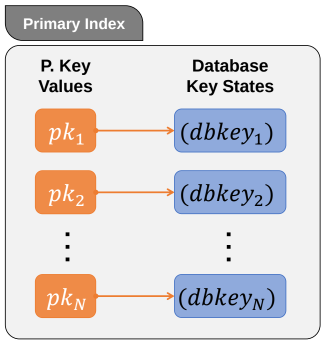
```
 1: payload sorted set T $index$                ▷ T is the CRDT type of the elements of the set
 2: initial $\emptyset$
 3: query $value()$ : sorted set $ret$
 4:     let $ret = \{(pk', newState) \mid (pk', state') \in entries \wedge (pk', computeQuery(\text{T}, \textbf{value}, [], state'))\}$
 5: query $get(\text{T } pkey)$ : entry $ret$
 6:     let $entry = \{(pk, state) \in index \mid pk = pkey\}$
 7:     let $ret = \{(pk, computeQuery(\text{T}, \textbf{value}, [], state))\}$
 8: query $range(\text{T } lbound, \text{T } ubound)$ : sorted set $ret$
 9:     let $entries = \{(pk, state) \in index \mid pk \geq lbound \wedge pk \leq ubound\}$
10:     let $ret = \{(pk', newState) \mid (pk', state') \in entries, (pk', computeQuery(\text{T}, \textbf{value}, [], state'))\}$
11: update $update(\text{T } pkey$, function $op$, element $dbkey)$
12:     AtSource($pkey$, $op$, $dbkey$) : $downstreamOp$
13:         if $\exists (pk, currState) \in index : pk = pkey$ then
14:             let $downstreamOp = computeDownstream(\text{T}, \textbf{update}, [op, dbkey], currState)$
15:         else
16:             let $emptyState = getNew(\text{T})$
17:             let $downstreamOp = computeDownstream(\text{T}, \textbf{update}, [op, dbkey], emptyState)$
18:     Downstream($downstreamOp$, $pkey$)
19:         if $\exists (pk, currState) \in index : pk = pkey$ then
20:             let $newState = performUpdate(\text{T}, \textbf{update}, [downstreamOp], currState)$
21:             $index := (index \setminus \{(pk, currState)\}) \cup \{(pk, newState)\}$
22:         else
23:             let $emptyState = getNew(\text{T})$
24:             let $newState = performUpdate(\text{T}, \textbf{update}, [downstreamOp], emptyState)$
25:             $index := index \cup \{(pk, newState)\}$
26: update $remove(\text{T } pkey)$
27:     AtSource($pkey$) : $downstreamOp$
28:         let $downstreamOp = computeDownstream(this.\text{type}, \textbf{update}, [pkey, \textbf{reset}, \bot], this)$
29:     Downstream($downstreamOp$, $pkey$)
30:         $computeUpdate(this.\text{type}, \textbf{update}, [downstreamOp, pkey], this)$
```

key correspondent to *pkey*; the set is initialized as an empty set and is sorted by the key in lexicographic order. A graphical representation of this data structure is presented in Figure 4.9a. As the traditional operation-based CRDTs, this specification includes **query** and **update** functions. The **query** functions are the *value*, *get*, and *range* functions.

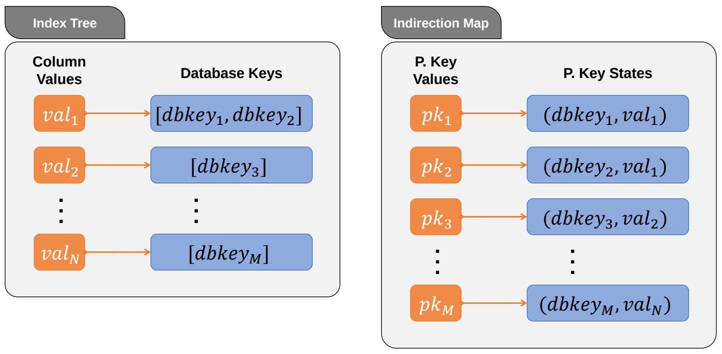
The *value* function (lines 3 and 4, from algorithm 12) reads the CRDT index and returns the elements of the payload set. This function uses an auxiliary function, *computeQuery*, that receives an embedded CRDT object (*state'*) and a query function (**value**), and returns the result of computing the function on the object. This function is not represented in the pseudo-code for brevity.

The *get* function (lines 5 to 7) receives a primary key and returns the respective set entry. An empty entry is returned if the key is not present in the set. At last, the *range* function (lines 8 to 10) allows to perform a range search on the set given a lower bound key and an upper bound key, and returns all entries whose primary keys comprehend between the lower and upper bounds.

On the other hand, the **update** functions are composed by the *update* and *remove*



(a) Primary index data structure.



(b) Secondary index data structures.

Figure 4.9: Graphical representation of the primary and secondary index data structures: (a) index sorted set; (b) *index tree* (on the left) and *indirection map* (on the right).

functions. The *update* function (lines 11 to 25) has the goal of updating a specific entry in the set identified by a given primary key. The update itself consists in applying a function on the embedded object of an entry, within two steps. The first step represents the calculations to be performed at the source replica, which in this case consists in reading an entry and computing the downstream operation of its embedded CRDT[5] (lines 12 to 17). To compute the downstream operation is used the function *computeDownstream* with similar behavior as the function *computeQuery*. Note that the downstream operation from the embedded object uses the *update* function from the primary index CRDT.

The second step reads the downstream operation calculated on the previous step and applies it to the set (lines 18 and 25). This last step uses the auxiliary function *performUpdate* to perform the received downstream operation in the corresponding entry, and then updates the set accordingly.

At last, the *remove* function receives a primary key and "removes" an entry from the set (lines 26 to 30). In fact, no entry is deleted, but instead a **reset** operation is issued to the entry's embedded object, using the *update* function to that end. That way, the **reset** operation will belong to the observed events of this CRDT, avoiding inconsistencies from concurrent updates on the entry. In other words, if the entry were to be literally deleted, the index could merge updates for that entry that happened causally before to the delete. And since the entry would not exist, the merge would occur instantaneously.

Secondary indexes. The specification for the secondary index CRDT is shown in algorithm 13. Its payload is composed by a sorted set, the *index tree*, which contains index entries (sorted by the lexicographic order of their keys), and a second set, the *indirection map*. The *index tree* stores $(colval, pkeys)$ pairs, in which a $colval$ is analogous to an

[5]Each node from the data center owns a copy of a CRDT object, which we call replica. Following this terminology, the outcome of the piece of code performed in this step is known as the *downstream operation* and is asynchronously sent from the source replica to the remaining replicas, so that the latter can apply the results locally on their copies of the object. More details in [72] and [73].

indexed column value, and *pkeys* is a list of simple database keys from the rows that reference the indexed value. On the other hand, the *indirection map* stores (*pkey*,*state*) pairs, where *pkey* is a primary key value, and *state* is an embedded CRDT object that stores the indexed column value pointed by the row identified by *pkey* and the database key of the same row.

The idea behind the *indirection map* is to have the same concurrency semantics as of the indexed column, such that the outcome of concurrent updates on the indexed column on the table is reflected in the state of index. Figure 4.9b represents graphically both data structures. The elements of the *index tree* have no specific type and the *indirection map* holds objects of type T.

At the **query** level, this specification includes functions *value*, *get*, and *range*, and additionally *lookup* (lines 3 to 10, algorithm 13). The first three functions present the behaviors already mentioned, except that they no longer need to read the values from CRDT objects since they can access the *index tree* for that purpose. The novelty comes from the function *lookup* that searches for an entry with a given key within the *index tree* and returns true if it exists, or false otherwise.

The **update** typed functions present the major changes from the previous design. The *update* function (lines 11 to 28) receives a primary key and a column value to be inserted in the index, and an operation that inversely binds the column value with the primary key. When computing the downstream operation, the entry whose key is the primary key value is retrieved from the *indirection map*. This entry contains an embedded object that is used to compute the downstream operation (lines 12 to 17).

Later on, the downstream operation is applied to the corresponding entry at the *indirection map* and the *index tree* is updated accordingly with the new indexed value (lines 18 to 28). If necessary, the primary key to be inserted may need to be removed from an entry where it belonged before. The index is updated using the auxiliary function *updateIndex* (lines 44 to 50).

Lastly, the *remove* function removes a primary key from the index by issuing a **reset** operation to the respective entry on the *indirection map*. After the reset operation is applied to the *indirection map*, the changes are reflected on the *index tree*. If the embedded object's new state from the *indirection map* is a bottom state, i.e. an initial CRDT state[6], the primary key ceases to exist in the index (lines 42 and 43). This indicates that it is possible to explicitly delete the primary key from the index (and perhaps the whole index entry where it belonged, if the entry list is empty), provided that the respective *indirection map* entry is not explicitly deleted.

[6]An initial object state may present some variations depending on the CRDT type, for example, an empty set for a CRDT *set* or an empty value for a CRDT *register*.

Algorithm 13 Operation-based secondary index.

```
payload sorted set T $index$, set $indMap$          ▷ T is the CRDT type of the elements of the set
initial $\emptyset, \emptyset$
query $value()$ : sorted set $ret$
    let $ret = index$
query $lookup$(T $colval$) : boolean $b$
    let $b = \exists pks\colon (colv, pks) \in index \land colv = colval$
query $get$(T $colval$) : entry $ret$
    let $ret = pks\colon (colv, pks) \in index \land colv = colval$
query $range$(T $lbound$, T $ubound$) : sorted set $ret$
    let $ret = \{(colv, pks) \in index \mid colv \geq lbound \land colv \leq ubound\}$
update $update$(element $pkey$, function $op$, T $colval$)
    AtSource($pkey, op, colval$) : $downstreamOp$
        if $\exists currState\colon (pk, currState) \in indMap \land pk = pkey$ then
            let $downstreamOp = computeDownstream(\mathrm{T}, \textbf{update}, [op, colval], currState)$
        else
            let $emptyState = getNew(\mathrm{T})$
            let $downstreamOp = computeDownstream(\mathrm{T}, \textbf{update}, [op, colval], emptyState)$
    Downstream($downstreamOp, colval, pkey$)
        if $\exists currState\colon (pk, currState) \in indMap\colon pk = pkey$ then
            let $newState = performUpdate(\mathrm{T}, \textbf{update}, [downstreamOp], currState)$
            $indMap := indMap \setminus \{(pkey, currState)\}$
        else
            let $emptyState = getNew(\mathrm{T})$
            let $newState = performUpdate(\mathrm{T}, \textbf{update}, [downstreamOp], emptyState)$
        $indMap := indMap \cup \{(pkey, newState)\}$
        if $\exists (colval', pkeys) \in index\colon pkey \in pkeys$ then
            $index := (index \setminus \{(colval', pkeys)\}) \cup \{(colval', pkeys \setminus \{pkey\})\}$
        $updateIndex(colval, pkey)$
update $remove$(element $pkey$)
    AtSource($pkey$) : $downstreamOp$
        let $downstreamOp = computeDownstream(this.\mathrm{type}, \textbf{update}, [pkey, \textbf{reset}, \bot])$
    Downstream($downstreamOp, pkey$)
        if $\exists currState\colon (pk, currState) \in indMap\colon pk = pkey$ then
            let $newState = performUpdate(\mathrm{T}, \textbf{update}, [downstreamOp], currState)$
            $indMap := indMap \setminus \{(pkey, currState)\}$
        else
            let $emptyState = getNew(\mathrm{T})$
            let $newState = performUpdate(\mathrm{T}, \textbf{update}, [downstreamOp], emptyState)$
        $indMap := indMap \cup \{(pkey, newState)\}$
        if $\exists (colval', pkeys) \in index\colon pkey \in pkey$ then
            $index := (index \setminus \{(colval', pkeys)\}) \cup \{(colval', pkeys \setminus \{pkey\})\}$
        if $\neg isBottom(newState)$ then
            $updateIndex(colval', pkey)$
function $updateIndex$(element $colval$, element $pkey$)
    let $pkeys = get(colval)$
    if $pkeys \neq \emptyset$ then
        if $pkey \notin pkeys$ then
            $index := (index \setminus \{(colval, pkeys)\}) \cup \{(colval, pkeys \cup \{pkey\})\}$
    else
        $index := index \cup \{(colval, \{pkey\})\}$
```

Table 4.4: AntidoteDB's API.

Operation	Description
$get(key)$	Reads an object identified by key.
$put(key, op, args)$	Writes an object identified by key, given an update operation op and arguments $args$.
$read(key, op, args)$	Reads an object identified by key, given a read operation op and arguments $args$.
$query(filter)$	Performs a query on the database, given a filter statement $filter$.

4.2.6 Query Optimizer

The query optimizer is a new element of the AntidoteDB architecture (Figure 4.2a) with the goal of interpreting and optimizing user queries, and computing their results. In addition, it allows to perform partial data calculations on server-side and transfer the results to the client.

This system contributes with an extension to the current AntidoteDB's API, by providing two new ways for issuing database queries beyond the traditional *put* and *get* operations from the *key-value* data store.

First, SQL-like queries can now be issued to the database in a well formed statement, through a *query* API function. This statement contains traditional SELECT statement fields, such as projection columns, table name, and WHERE clause conditions. The query optimizer receives this statement and elaborates the best plan to retrieve the result in a fast manner.

At last, the AntidoteDB's API now supports issuing read functions to database objects, through a *read* operation. Unlike the traditional *get* operation that reads the whole state from a database object, the *read* operation extends the previous database operation by receiving the CRDT database key to read and the name and arguments of a CRDT function. In turn, the CRDT type should export the function in a similar way to the CRDT index specifications presented in Section 4.2.5.2, using the **query** annotation for that purpose. Intuitively, the new *read* operation allows to retrieve partial objects from the database, by applying custom functions on CRDT snapshots within the data server. This solution addresses the limitation from the *get* operation that transfers a full CRDT snapshot from the database to the client, which in turn may present a cost on the transfer overhead of the object, but also forces the client to execute its queries locally on this snapshot.

Table 4.4 presents the new AntidoteDB's API, including the new API operations *query* and *read*. Although omitted from this table, each operation is executed under a transaction.

Query algorithm. A well designed query optimizer must take into account several aspects and some database information to achieve the best performance and low execution latency. As previously mentioned in Section 4.2.1.4, the former SELECT statement supported very simple conditions such as equality conditions over primary key columns, which merely consisted in reading directly an object from the database (since a primary key value and a table compose a database key). However, the SELECT statement we propose allows more complex queries, specifically more complex conditions at the WHERE clause. Thus, reading directly a database object through its key does not suffice.

The query optimizer makes sure to use aforementioned features such as indexes and query precedence on the query processing. Therefore, we present now the query algorithm to process a query given a list of projection columns *proj*, a table name *tname*, and a list of conditions *conds*. The algorithm proceeds with the following steps:

Step 1: The algorithm validates the projection columns in *proj* within table *tname*. If some columns are invalid, an exception is thrown;

Step 2: The algorithm evaluates the conditions in *conds*: if the *conds* list is empty, the primary index from table *tname* and all its records are read and the algorithm proceeds to step 5; else, the algorithm proceeds to step 3;

Step 3: The algorithm traverses the conditions in *conds* and identifies sub-conditions, i.e. conditions that are enclosed between parenthesis. The whole WHERE clause is considered as a sub-condition;

Step 4: The algorithm initializes a *subPartialRes* set to hold partial results from sub-conditions, and for each sub-condition, proceeds with the following steps:

a) The algorithm traverses the sub-condition and identifies *conjunction groups*, i.e. groups of one or more conditions that form a conjunction. The goal is to process groups of conjunctions individually and join each of their results;

b) The algorithm initializes a new *conjPartialRes* set to hold partial results from *conjunction groups*, and for each *conjunction group*, calculates a partial result to be merged with *conjPartialRes* according to the following:

   i. Traverse the *conjunction group* and find sub-conditions;

   ii. If sub-conditions are found, for each one perform only the whole step 4 and intersect each intermediate *subPartialRes* set with the current computed *conjPartialRes*;

   iii. Group remaining conditions by column and find a *range group* intersection between conditions on the same column. If no intersection is found, this condition group has no solution, and the algorithm proceeds to the next conjunction group (step 4b);

   iv. If the current *conjPartialRes* is not empty, filter this set with the conditions from the *range group* and proceed to step 4(b)vi;

   v. Else, given the *range group*, identify the usable indexes and read data from the indexes:
      A. If no index is found, read the primary index to make a full table scan and intersect the result with *conjPartialRes*;
      B. Else, find the index that returns the least number of rows concerning the *range group*, read the index data, retrieve the database keys, and read the corresponding rows. These rows are then filtered with the remaining conditions from the group (i.e. the ones which do not point to any index) and the result is intersected with *conjPartialRes*.
   vi. Perform the union of the *conjPartialRes* with the current partial result *subPartialRes* and proceed to the next *conjunction group*.

Step 5: The algorithm performs the projection over the latest *subPartialRes* through the list of columns of *proj* and returns the result.

This algorithm hides some details, for simplicity of presentation. We now discuss these details.

When processing groups of conjunctions, on step 4(b)iii, the algorithm collects all conditions and groups them by column, such that it creates a mapping between columns and ranges of values, i.e. a *range group*. When a column appears on a group of conjunctions, it automatically creates a range of all possible values it can attain on this group, such that the range has a lower and an upper bound. For instance, consider the following group of conjunctions from a WHERE clause:

```
Artist = 'Michael' AND ArtYear > 2010 AND ArtYear <= 2019
```

For this group of conjunctions, the algorithm collects the following ranges for each column, that constitute a *range group*:

$$\texttt{Artist} \longrightarrow [\texttt{`Michael'},\texttt{`Michael'}]$$
$$\texttt{ArtYear} \longrightarrow]2010, 2019]$$

Creating ranges out of AQL conditions presents two benefits for our solution. First, it prevents the algorithm from reading and evaluating records unnecessarily from the database; knowing in advance that some conditions represent impossible solutions (because they do not intersect, and thus, represent impossible intervals of values) is more efficient than evaluating those conditions on the data. For example, in the group of conditions above, if instead of being `ArtYear <= 2019` the condition was `ArtYear <= 2009`, the whole group would be invalidated and no evaluation was needed for these conjunctions.

Second, it takes advantage of the new read semantics when reading database objects that, in our solution, consists in reading partial index data by calling index read functions for that purpose, such as the *get* and *range* functions (present in algorithms 12 and 13). Step 4(b)v of the algorithm demonstrates the use of indexes on query processing.

4.3 Summary

In this chapter, we described the system model of our solution that meets the goals for this work.

Section 4.1 details the system's architecture on the basis of the two systems AQL and AntidoteDB, and explains how these two systems interact with each other to form the main system. For each individual system the main architectural modules and their interconnections are presented, including the communication between clients and system.

Section 4.2 details at the high-level the functionalities employed in this work. This section starts by defining the AQL's SQL-like syntax, including high-level user and administrative statements, data-types supported, and the supported database invariants, such as primary key, check, and foreign key constraints. In a proper section, the referential integrity mechanism is detailed, presenting the algorithms proposed for improving this common SQL feature in the system. The remaining of the main section explains the table partitioning, the indexing system, including formal index CRDT specifications, and the algorithm of the query optimizer, all employed in the proposed system.

Chapter 5

Implementation

This chapter focuses on the implementation aspects of our solution, concerning the two systems AQL and AntidoteDB. In particular, we present the main tools used to implement the features introduced in the previous chapter and explain how some functionalities were mapped on a programming platform. In addition, we mention some testing techniques and the deployment of our solution.

5.1 Programming Environment

Our system was mainly implemented using the Erlang programming language. Erlang [32, 48] is a functional programming language initially created by the Ericsson company [31], but that rapidly increased its popularity on other programming fields.

Its most well known characteristics include high concurrency, high reliability, fault-tolerance, and scalability, which are desirable characteristics in distributed systems. As a distributed programming language, it follows the *actor* model, where an *actor* is responsible for executing a limited (and perhaps small) number of tasks within a system, including communicating with other actors. Among its supported data structures, includes lists, tuples, atoms, variables, maps, records, and many others.

Erlang is used by well-known distributed systems, such as the NoSQL database Riak [68], the message broker RabbitMQ [64], and the Amazon's SimpleDB database system [6].

Erlang contains a framework known as Open Telecom Platform (OTP), a framework composed of tools, libraries, and other software that allow programmers to easily build their own concurrent distributed applications. An OTP distributed application is composed by several individual processes that communicate with each other (within the application) and with outside processes through messages and predefined configurations.

Because of its importance in the Erlang language, sometimes the name Erlang is interchangeable with the name Erlang/OTP. OTP offers fundamental tools for developing our solution, specially on building the major application of our system, as well as the necessary servers to meet our proposed architecture.

Besides the OTP framework, our solution also relies on the Rebar3 [65] tool to build our Erlang projects. Rebar3 allows the programmer to create, compile, test, and manage Erlang applications and their dependencies. Rebar3 allows to build releases of an application, by packaging all dependencies and source binaries into a single and exportable unit, through simple commands and configuration files. This tool is used to manage our solution, in terms of building and testing our application.

5.1.1 Development Tools

Both AQL and AntidoteDB code is maintained in the version control platform GitHub [45]. GitHub is a software based on Git [44], a system that allows to define work-flows for a project (e.g. through branches), to follow its development, and to manage its resources in a distributed and cooperative manner.

In our work, the AQL [12] and AntidoteDB [11] systems represent two distinct Erlang projects hosted on two distinct public GitHub repositories. Each repository contains a master branch that presents a stable solution. In addition, each repository contains a number of branches that correspond each to a different functionality of the respective system.

5.2 System's Architecture

As mentioned in the previous section, AQL and AntidoteDB are implemented in two separate Erlang projects, each representing an OTP application. The AntidoteDB presents the implementation of a data node (as depicted in Figure 4.2a), where each component of the system represents either an Erlang generic server [38] or a finite state machine (FSM) process [36] that communicates with the other components through direct calls or message passing [48]. The proposed components on AntidoteDB's architecture – the query optimizer and the index manager components – follow the same approach.

On the other side, the AQL implementation is pretty straightforward, by directly implementing an OTP application with two Erlang servers, the AQL module and the Web Server module (depicted in Figure 4.2b). The AQL module is a server responsible for receiving and parsing client operations through the Erlang's Remote Procedure Calls (RPCs), which is the classic way for clients to communicate with the AQL system. In order to parse operations, AQL resorts to the Erlang modules Leex [39] and Yecc [40], where Leex is a lexical analyzer responsible for transforming a sequence of strings into a token sequence (the tokens must be defined in advance), and Yecc is a parser library

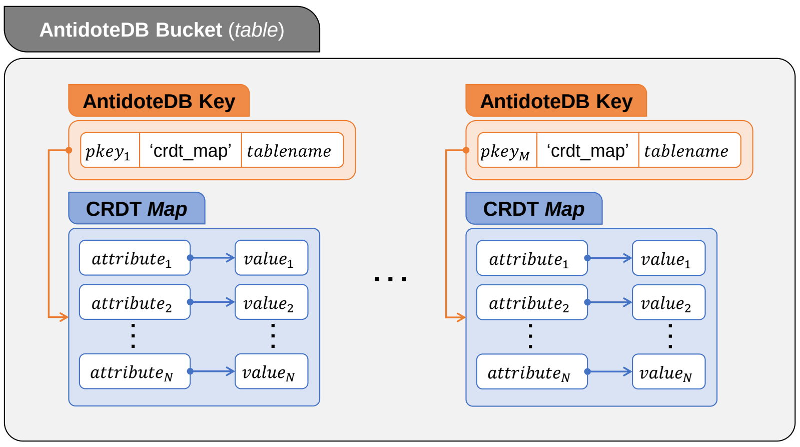


Figure 5.1: Representation of the table and record mapping inside AntidoteDB.

that interprets a list of grammar rules to transform a token sequence into an Erlang specification, composed by some of the language's primitives (e.g. tuples and atoms).

On the other hand, the Web Server module is an Erlang server implemented using the Elli [30] library, a web server suitable to export HTTP APIs through a well defined interface. A client who wishes to communicate with AQL via the HTTP server must send her queries in the body of a HTTP message using the verb 'POST'.

Although AQL and AntidoteDB systems are implemented in two separate projects, the AQL OTP application assumes the AntidoteDB project as a direct dependency. That way, an AntidoteDB node starts its operation in the background, simultaneously with an AQL instance, and makes it possible for AQL to communicate directly with the AntidoteDB through local calls.

5.3 Database Schema

In this section we explain how the proposed database schema is mapped into AQL and AntidoteDB specifications. As stated in the previous chapter, the AQL's database schema is similar to the schema from relational databases, including tables and indexes.

The AQL is responsible for building this schema within AntidoteDB, and thus highly relies on the AntidoteDB's built-in objects to support a SQL-like schema. For that purpose, we take advantage of the AntidoteDB CRDT library [7].

5.3.1 Tables/Rows Mapping

In AQL, a table is a set of database objects, each one representing a row. By mapping rows to individual database objects, rows become scattered among the data nodes on a data center, which simplifies scalability through *sharding*.

Given the rows are mapped into database objects, we take database keys to identify those rows within the database. AntidoteDB defines a database key as an Erlang tuple

Table 5.1: AQL data types and corresponding CRDT types.

AQL data type	CRDT type
VARCHAR	*Last-Writer-Wins Register*
INTEGER/INT	*Last-Writer-Wins Register*
BOOLEAN	*Flag-EW*
COUNTER_INT	*PN-Counter* (without check constraint) *Bounded Counter* (with check constraint)

containing the object key, type, and bucket, such as:

```
{object_key, crdt_type, bucket}
```

For a row, this tuple consists in the primary key value, the CRDT *Map* type, and the table name, respectively, all represented as Erlang atoms. Figure 5.1 illustrates the use of the AntidoteDB analogy and internal objects to represent a table and its records.

Each row is mapped into a CRDT *Map* object within the database. A CRDT *Map* is used to store row data as a mapping between column name and value, which suggests that each entry from the map consists in a column name and an embedded CRDT object representing the value on the column. Each embedded object is defined according to the column data type and its concurrency semantics, as represented in Table 5.1, and briefly explained below.

VARCHAR and INTEGER data types. For the VARCHAR and the INTEGER data types, a column uses a CRDT *Last-Writer-Wins Register* [7, 72], or *LWW-Register* for short. As the name suggests, this CRDT deploys a *last-writer-wins* behavior on concurrent updates over a column.

BOOLEAN data type. For the BOOLEAN data type, the *Flag-EW* CRDT [7] is used. This CRDT supports a concurrency semantics that states that an enable operation prevails over a disable operation on a concurrent scenario.

COUNTER_INT data type. For the COUNTER_INT data type, the corresponding CRDT type is chosen according to whether or not the column specifies a check constraint. If the column does not specify a check constraint, the *PN-Counter* CRDT [7, 72] is used. This CRDT supports increment and decrement operations that commute, which allows convergence for the column. On the other hand, if the column specifies a check constraint, the *Bounded Counter* CRDT [7, 16] is used instead. A more detailed explanation of the use of this CRDT in check constraints is stated in Section 5.4.2.

5.3.2 Table Metadata

For each table of the database, AQL maintains the necessary information for table management. The table metadata includes:

- The **table name**;
- The **conflict-resolution policies** employed by the table, which include the table policy, the foreign-key policy, and the child foreign-key policy;
- The **columns specifications**, including the column names, types, and constraints (if applicable);
- The **foreign key specifications**, where a specification includes the foreign key's column and type (from the referencing table), the referenced table and the referenced column, and the *delete-cascade* behavior (if applicable);
- The list of **indexes** created for the table, including the index name, and the indexed column;
- The **partition column** (when applicable).

In order to store the metadata for the database tables, AQL resorts to a database object consisting in a CRDT *Map*, under a predefined database key. The database key is defined with the tuple `{'#tables', crdt_map, aql_metadata}`. All tables metadata is stored in a single database object, more specifically in a CRDT *Map*, where an entry of the map corresponds to the metadata of a specific table stored on an embedded CRDT *LWW-Register*.

To guarantee fast accesses to the metadata, each AntidoteDB's data node caches the metadata of all tables in an in-memory and segmented cache, using an open-source cache implementation in Erlang [42].

5.3.3 Table Partitioning

To support table partitioning, AQL uses database keys to aggregate rows that have a column value in common. The rows are identified accordingly to support direct access given the column value. Therefore, the database keys use bucket names that encode not only the table name, but also the value of the partition column, as follows:

```
{primary_key, crdt_map, {table, hash}}
```

The `hash` represents the hash value of the original value from the partition column. We use the Erlang's native Crypto library [34] to generate the hashed value in binary form, and then we decode the binary to a sequence of characters.

A table with partitioning enabled causes the system (both AQL and AntidoteDB) to need to follow an indirection to discover the database key for a given primary key. To avoid recalculating database keys in most cases, AntidoteDB directly stores keys at the index level. This is mainly useful to accelerate query processing, given index data contains the database keys for direct access to the rows.

The idea behind the presented partitioning mechanism, giving particular attention to the spreading of rows among different partitions, is to allow the AntidoteDB to deploy (in a future release) a replication mechanism such that a data center only receives updates for a given set of buckets (or tables) it owns. This replication mechanism is not yet developed in AntidoteDB, limiting our solution to use the current replication architecture of the database.

5.3.4 Indexing System

The implementation of the proposed indexing system is divided into the implementation of the index data structures and the implementation of the index manager.

5.3.4.1 Data structures

The implementation of the index data structures is pretty straightforward and intuitive. Section 4.2.5.2 defines the algorithms for both primary index and secondary index CRDT specifications, which are the building blocks for their respective implementations.

Primary index. Concerning the implementation of the primary index, the payload of this CRDT consists in a *general balanced tree* (*GB-Tree*, for short) [37], a balanced *key-value* data structure where one *key-value* pair represents a tree node. Following the primary index specification, the key of a node is a primary key value and the value of a node consists in an embedded CRDT *LWW-Register* that stores a database key that points to the row identified by the primary key.

Secondary index. At the implementation level, the secondary index specification uses two data structures as payload, an *index tree* and an *indirection map*. The *index tree* is represented as a *GB-Tree* where a node's key is an indexed value and a node's value is an ordered set of database keys that point to the indexed value. On the other hand, the *indirection map* is implemented as an Erlang dictionary that maps a raw primary key value to a tuple `{DBKey, IValue}`, where `DBKey` is a database key and `IValue` is the indexed value pointed by the row identified by the primary key. Internally, both `DBKey` and `IValue` are embedded CRDT objects that store their respective values: the database key is stored on a CRDT *LWW-Register* and the index value is stored on a CRDT of the same type as the column specifies on the respective table.

The implementations of each index are at the *indexes* branch of the AntidoteDB's CRDT library [7]. This library also constitutes a dependency of the AntidoteDB's Erlang application [11].

5.3.4.2 Index management

As for the management of the indexes themselves, AntidoteDB deploys the index manager component (depicted in Figure 4.2a) to issue index updates to the database whenever necessary. As previously mentioned in Section 5.2, the index manager constitutes an internal server within AntidoteDB that receives row updates and generates index updates from these. Additionally, the index manager employs additional functions to read full or partial index data.

In order to catch row updates, the index manager registers a *pre-commit hook* for every table created on the database. A *commit hook* [10] is an additional piece of code to be executed over a database key, before or after a transaction has committed. That is, for every key that is updated in a single transaction, the *pre-commit hook* will be executed[1], such that *k* keys will trigger the *commit-hook* *k* times. In a SQL context, the *hook* functionality can be seen as a SQL trigger that activates after the update or deletion of a row.

Given that, we implement a *hook* to be executed for every updated row, before a transaction commits, that consists in gathering the updates inferred for that row within the transaction and generate index updates properly. The index updates generated are issued alongside the row updates, to be validated and executed in the context of the same transaction.

5.4 Database Invariants

This section summarizes the implementation techniques used to maintain database invariants. As stated in Section 4.2.2, we identified three table constraints that are addressed in our solution, the primary key, check, and foreign key constraints.

5.4.1 Primary Key Constraint

AQL resorts to AntidoteDB keys to guarantee that a single row exists for a given primary key, by merging the values of the row on concurrent updates, as previously mentioned in Section 5.3.1. In order to guarantee that a primary key is not null, AQL's internal implementation validates the non-null primary key on insertion. Additionally, on update, AQL validates the primary key is not changed.

5.4.2 Check Constraint

The check constraint defines a lower or an upper bound on a numeric column. As previously stated in Section 5.3.1, this constraint is only suitable for the AQL's COUNTER_INT typed columns. At the implementation level, a COUNTER_INT column defined with a check constraint makes use of a *bounded counter* CRDT [16]. This CRDT supports a default bound that states the value of the counter must be greater than or equal to zero, despite concurrent increments and decrements.

Since the AQL syntax allows the user to define customized check conditions (explained in Section 4.2.1.1), in order to establish a custom bound for this counter, and

[1]This is analogous to say that, for every row insertion/update/deletion, the *pre-commit hook* will trigger and execute its corresponding function.

Table 5.2: Mapping between bound comparators and their correspondent *btype* and *isEq* values (adapted from [75]).

Bound comparator	***btype***	***isEq***
Lesser (<)	-1	1
Lesser Eq. (≤)	-1	0
Greater (>)	1	1
Greater Eq. (≥)	1	0

to essentially calculate the original value of a column, AQL follows the formula below (adapted from [75]):

$$value_{AQL} = btype \times (value_{AntidoteDB} + bvalue \times btype + isEq)$$

where $value_{AntidoteDB}$ is the value of the counter stored in AntidoteDB, *bvalue* is the bound value determined for the column (i.e. the integer number on the check condition), and *btype* and *isEq* are determined according to the bound comparator, as stated in Table 5.2.

The formula above adapts from the built-in bound of "greater than or equal to 0" of the *bounded counter*, to a custom bound through the formula's variables:

- When the bound is "greater than or equal to *bvalue*", *bvalue* is added to the database counter value $value_{AntidoteDB}$ to obtain the desired result.
- When the bound is "greater than *bvalue*", *bvalue* is added to the counter value $value_{AntidoteDB}$ and 1 is added to the result (activated by the variable *isEq*), since "greater than *bvalue*" is equivalent to "greater than $bvalue + 1$".
- When the bound is "lesser than or equal to *bvalue*", *bvalue* is subtracted from the database value $value_{AntidoteDB}$ through the *btype* variable.
- When the bound is "lesser than *bvalue*", the *bvalue* is subtracted from $value_{AntidoteDB}$ and the calculated result adds 1, due to the *isEq* variable.

The presented formula is used to compute the value of a counter column when AQL receives the value from AntidoteDB. On the other hand, when AQL needs to send the counter value to the database, the formula is solved for $value_{AntidoteDB}$ and is used to compute the AntidoteDB counter value.

5.4.3 Referential Integrity Constraint

Referential integrity allows to establish relationships among the tables of a database. AQL employs this common SQL functionality in the geo-replicated, NoSQL database AntidoteDB, by designing strategies that allow concurrent updates to be executed on rows that maintain a parent-child relationship without breaking referential integrity.

5.4.3.1 Validation mechanisms

Some violation-avoidance mechanisms were implemented in AQL, in order to prevent database inconsistencies. First, when a CREATE TABLE operation is issued, AQL validates that all foreign key specifications point to existing tables and respective primary key columns, given the causality of events at the point in time the table is created. Besides, AQL also validates the foreign key semantics defined on this statement do not break referential integrity, by taking into consideration incompatible semantics (see Table 4.3).

Second, AQL validates row updates (e.g. INSERT and UPDATE operations) on foreign key columns, to prevent rows from pointing to *non-existent* primary key values from other tables. The term *non-existent* refers to not visible rows, given our visibility system.

At last, AQL prevents a parent row from being deleted if some of the child tables (or child from the child tables, and so on) did not specify the *delete-cascade* behavior for at least one of their foreign keys. This validation is made on-the-fly while the child rows are deleted, which means that the transaction (that performs the deletion) aborts if AQL finds a child table with foreign keys with this characteristic.

These validations, as other operations in the database, are performed using the table metadata.

5.4.3.2 Visibility mapping

Recalling Section 4.2.3.1, the solution designed to support referential integrity in AQL is characterized by a visibility system composed by concurrent resolution semantics, but above all, by visibility tokens and version numbers that help to determine if a row is visible or not visible within the database.

Each row has two additional columns, a column for the row's state and another for the row's version. At the implementation level, this corresponds to each row object containing two additional entries on the CRDT *Map*. The state column is represented as a CRDT *MV-Register* [7, 72] within the row map, in order to support multiple values (i.e. tokens) from concurrent updates on a row. On the other hand, the version column is represented by a CRDT *LWW-Register* [7, 72] that holds the last seen version number for a row.

In addition to these two columns, a row may contain as many columns for foreign keys it points to as the table specifies. Each one of these columns represents also an entry on the row's CRDT *Map*, where the entry's key is the textual representation of the parent table's name and column, and the entry's value is a CRDT *LWW-Register* storing an Erlang tuple `{fk_value, fk_version}` that concerns a parent row's version.

Although the aforementioned columns are necessarily present in the row's database object, both are hidden to the final user and are used only for visibility purposes. Figure 5.2 illustrates the mapping of the row $ArtWork('W0')$ into an AntidoteDB object, with the state, version, and foreign key hidden columns, represented by the names "`#st`", "`#vrs`", and "`{Artist, Name}`", respectively.

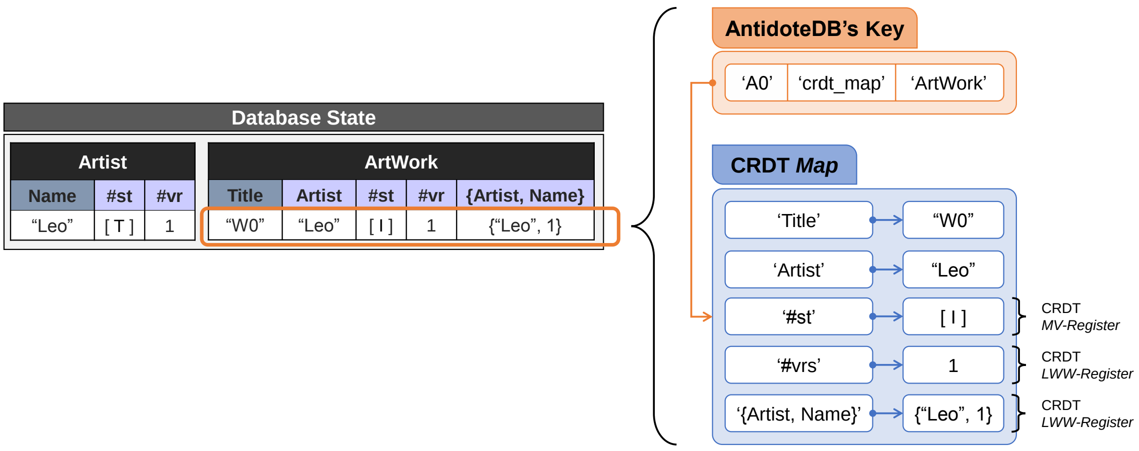


Figure 5.2: An illustration of a row mapping into an AntidoteDB object. Erlang atoms are enclosed in single quotes.

It is important to state that the version column and the posterior foreign key columns, though present in a row's object, are exclusively used when a *delete-wins* foreign key policy is used for the respective table, as explained in Section 4.2.3.3.

5.4.3.3 Visibility calculation

The visibility of a row is performed using three types of hidden columns mentioned at the previous section. Both AQL and AntidoteDB resort to the values of these columns whenever it is necessary to calculate the state of a row.

The algorithm for checking a row's state may vary depending on the foreign key semantics applied to a table, as explained in Sections 4.2.3.2 and 4.2.3.3. For instance, computing the visibility of a row on a table whose foreign keys semantics is *update-wins* may only consist on checking the row's state, while in *delete-wins* semantics it may be necessary to check the parent row's visibility as well.

In either case, validating a row's existence consists in comparing the list of visibility tokens (from the state column) with the rule of the table where the row resides. A rule is determined through table and foreign key policies, as demonstrated on Table 4.3. Given the table rule, the comparison will determine which token from the list will prevail, and that token will determine the row existence.

In a more comprehensive way, the execution plan to assert the visibility of a row r from table t is defined by the following algorithm:

1. Build the visibility rule for table t;

2. Fetch the prevailing token from r's state, by comparing the list of tokens from r with the table rule;

3. If the fetched token is D, the row r is *not visible* in table t; ■

4. Else, for each foreign key $\{p, v\}$ of r, where p is the parent primary key and v is the version of p that r points to:
   a) If the foreign key policy of t is *delete-wins*:
      i. Fetch the parent row p and calculate its visibility resorting to this algorithm;
      ii. If both p's version and v do not match *or* p is not visible, the row r is *not visible* in table t; ■
      iii. Otherwise, continue the foreign key loop from Step 4.
   b) Else, if the parent's foreign key policy of t is *delete-wins*:
      i. Fetch the parent row p and calculate its visibility resorting to this algorithm;
      ii. If p is not visible, row r is *not visible* in table t; ■
      iii. Otherwise, continue the foreign key loop from Step 4.
   c) Otherwise, continue the foreign key loop from Step 4.
5. If the previous step validated, row r is guaranteed to be *visible* in table t. ■

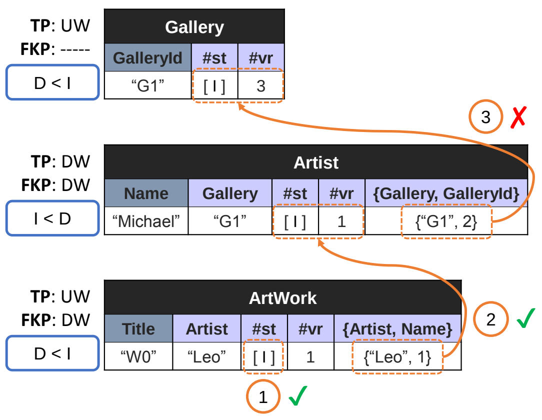
As represented in the algorithm above, when necessary recursive calls are performed upwards in the hierarchy in order to assert visibility of parent rows. Typically, this scenario only occurs when the table itself or the parent table specify the *delete-wins* foreign key semantics. In addition, whenever needed, information concerning a table and its parent tables is retrieved from the table metadata object.

Figure 5.3 illustrates two examples that calculate the visibility of the row $ArtWork('W0')$, by resorting to the execution plan proposed. Next to each table are represented the table policy (TP), the foreign key policy (FKP), and the visibility rule generated from these policies.

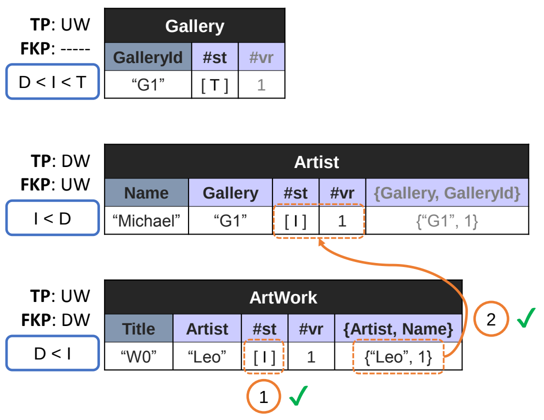
The first example uses three tables that specify the *delete-wins* foreign key policy, depicted in Figure 5.3a. As stated in this example, three steps are needed in order to determine that row $ArtWork('W0')$ is not visible due to row $Artist('Michael')$ being not visible. The second example, represented in Figure 5.3b, uses the same database state, but instead uses the foreign key policies *update-wins* and *delete-wins* for the $Artist$ and the $ArtWork$ tables, respectively. In this case, it only takes two steps to assert row $ArtWork('W0')$ existence. Because table $Artist$ employs *update-wins* semantics, it does not make use of the version columns and hence they are printed in grey text in the figure.

5.4.3.4 Locking system

The locking system is a sub-solution for the referential integrity functionality and is employed when the access to database rows needs to be restricted. The urge to resort to a locking mechanism comes from the specification of the *no-concurrency* semantics on foreign keys.



(a) *Delete-wins* scenario.



(b) Mixed scenario.

Figure 5.3: Example of the visibility calculation of record $ArtWork('W0')$ in two different scenarios: (a) all foreign key policies are *delete-wins*; (b) the foreign key policies are *update-wins* and *delete-wins* approaches.

When used as a conflict-resolution policy, the *no-concurrency* semantics restricts the access to a parent row when it is being deleted. The access is restricted to prevent child rows from being created concurrently with the parent row's deletion, thus precluding a concurrent conflict.

AntidoteDB with strong semantics. In order to implement this semantics, AQL resorts to an already developed AntidoteDB version [9] that implements a locking system with strong semantics. In summary, this version deploys a lock manager that exports an interface for acquiring and releasing *exclusive* or *shared locks* in the database. Besides, the locking manager allows for transactions to acquire locks at the beginning of their operation, which are released at the commit or rollback. This system is also characterized by deploying a locking ownership within a data center and among data centers, and by serializing all lock requests within one data center.

To take advantage of the interface provided by the locking system, AQL performs requests to AntidoteDB whenever locks need to be acquired or released in the context of a transaction. Recalling Section 4.2.3.4, a transaction may acquire two types of locks, and in specific occasions:

- An ***exclusive lock*** is acquired by a transaction for a row that is being deleted. If this operation involves the deletion of child rows, the transaction may acquire *exclusive locks* for child rows that specify *no-concurrency* semantics on their foreign keys;

- A ***shared lock*** is acquired by a transaction for *all parent rows* of a row that is being inserted or updated. "All parent rows" covers parents of parents, i.e. all rows involved in the referential integrity hierarchy whose foreign key semantics are *no-concurrency*.

Acquiring a lock for a row, at the implementation level, consists in reserving the database key of the row. At the transaction's commit or rollback, all locks are released.

5.5 Read Operations

AntidoteDB was modified to support efficient read operations in AQL. These modifications, such as the implementation of a query optimizer and read operations for reading partial objects, are described in the remaining of this section.

5.5.1 Query Optimizer

The query optimizer is a novel feature of the AntidoteDB database that allows to read database objects by interpreting a set of rules that resemble an AQL query. Therefore, the query optimizer is an intermediate component between the AQL interface and the database, in which AQL queries are performed internally within AntidoteDB.

In order to perform a query in the database, the AQL instance sends a set of information to AntidoteDB, required to perform the query. This set of information is the internal representation of an AQL query, composed by:

- The **table** where the query is performed;
- The list of **projection columns** that allows to filter table rows by columns;
- The list of **conditions** that allows to filter the rows from a table, given that their column values satisfy all conditions.

Internally, the table name is represented as an Erlang atom. Currently, only a table is allowed, but we aim to support queries with more than one table in the future. On the other side, the projection columns are a list of atoms, each atom representing a column name. Only valid column names are allowed for the specified table. At last, the conditions are represented as a list of several sub-lists and tuples, such that:

- **Conjunctions** are grouped together in a list, where a **conjunction** is a tuple {*column*, *comparator*, *value*};
- **Disjunctions** represent groups of conjunctions, which are analogous to lists of lists;
- **Sub-conditions** are tuples {`sub`, *disjunctions*}, where *disjunctions* is a list of disjunctions.

To illustrate this syntax, consider the following AQL query:

```
SELECT Name, Country
FROM Artist
WHERE Name = 'Michael' OR (Age > 2010 AND Age <= 2019 OR Country = 'PT')
```

The set of tuples below represents the query transformed into its internal form:

```
{table, `Artist'},
{projection, [`Name', `Country']},
{conditions, [
  [{`Name', equality, "Michael"}],
  {sub, [
    [{`Age', greater, 2010}, {`Age', lessereq, 2019}],
    [{`Country', equality, "PT"}]
  ]}
]}
```

After receiving a tuple with these characteristics, AntidoteDB resorts to the query optimizer to interpret the tuple and retrieve the results. The query optimizer implements the algorithm described in Section 4.2.6 to process a query and calculate the final result, by using the index data structures and by manipulating record data.

5.5.2 Partial Object Reads

Partial object reads is a feature implemented in AntidoteDB that makes it possible for the clients to retrieve smaller portions of database objects, given a filtering function exported by the object (i.e. the CRDT type).

In order to implement the partial object reading feature, few changes occurred inside the AntidoteDB's implementation.

First, the API of the database needed to be extended so that the clients could take advantage of the partial object reading. To that end, we added a new database operation to the API, that receives a database key, a function and its arguments to be issued to the respective object, and a transaction identifier.

Second, the transaction manager was modified to support functions to be issued to the objects. The first modification in this context was to allow a transaction to read object snapshots from its *read set*. Only then, the partial reads were implemented, consisting in the following: given a database key, the CRDT type, and a read function, the implementation reads a snapshot of the object identified by the key, from the materializer layer or the transaction's *read set*, and applies the function on the local snapshot. The outcome of executing the read function on the snapshot is then sent to upper layers, such as the AntidoteDB's API, the query optimizer, or the index manager.

5.6 Testing

While both systems were being developed, AQL and AntidoteDB underwent continuous tests to verify the correctness of their code. For that purpose, some testing tools, most of them built in Erlang, were used in order to achieve correct solutions for our systems, that we present in this section.

5.6.1 Unit Testing

Erlang allows to execute simple pieces of data that test the functionality of a unit of code. A unit can be a function or an entire module of programming code, and is tested resorting to the Erlang framework EUnit. The EUnit framework [35] allows to define functions where each may represent the execution of a certain feature within the module in an off-line mode. The goal is to assert functionality and find bugs in the code.

In order to specify unit tests, a module must include a valid path to the EUnit library through a `TEST` macro at the header of the module. Thereafter, each unit test is specified through a simple function, whose name must necessarily end with "`_test`" and does not take any arguments. Additionally the function must be within a `TEST` scope in order to be recognized by the testing framework. The content of a test function may contain specific macros from the EUnit framework, such as `?assertEqual` (to test if an expected result equals the obtained result), `?assertMatch` (to test if the obtained result pattern matches the expected result), and so on. Whenever one of these macros does not validate, the test fails and may throw an exception; a test may also fail by runtime errors. Otherwise, the test succeeds and that information is printed to the programmer.

To run all unit tests declared for a module at once, the EUnit automatically exports the function *module*`:test()` from the given module; on the other hand, to run a given set of tests, a *module*`:test(`*test_set*`)` function is exported from the module and called for this purpose.

AQL and AntidoteDB follow this semantics and only some (and relevant) Erlang modules implemented in these systems are programmed to run unit tests. For executing the tests, both systems do not depend on the execution of each other. Additionally, Rebar3 is used as part of the testing execution process, providing a specific command for this purpose.

5.6.2 Common Testing

Similarly to unit tests, Common Tests (CT) [33] is an Erlang framework that allows to perform integration tests on the code, but in a broader spectrum of use. Unlike the EUnit framework (described in the previous section), this framework allows to test whole OTP applications and libraries, which represent larger environments than the ones usually tested under EUnit. Therefore, CTs allow to test the functionalities of a whole system, by taking advantage of several modules and libraries from that system to perform the tests.

The semantics used by CTs for testing are based on test suites. A test suite is a module that contains a set of test cases that are preceded by the execution of the initial setup for the suite and followed by the final setup. Like the EUnit specification for unit tests, a test case is a function that tests certain functionality of a system and whose possible outcomes are succeeding or failing. The initial and final setups for a test suite are instructions to be performed before and after the evaluation of the test cases, respectively, and may consist in data initialization or deleting data from the side-effects of the test suite, for instance.

After the execution of a set of test suites, the CT framework logs the results into HTML files with some statistics regarding the execution of the test suites, such as the total execution time of the test suites, including the execution time of each test case, the number of succeeded and failed test suites, etc.

Both AQL and AntidoteDB systems resort to Common Tests in order to test their OTP applications on a black-box fashion. Each project contains a *test* directory that contains suitable test suites for the given system. For instance, AQL creates test suites regarding the featured queries and the referential integrity mechanism, while AntidoteDB specifies test suites regarding the transactional mechanism and the inter-data center replication.

5.6.3 Continuous Integration

Continuous Integration (CI) is the act of continuously merging small changes of code to a single line of development. This development practice enables to perform incremental testing on the code, including unit tests and integration tests, and building and deploying code automatically on the moment new code is submitted for the project. Nevertheless, a development project should never be exempt of local testing.

In order to employ a CI mechanism, AQL and AntidoteDB resort to TravisCI [82], a continuous integration tool that is mostly used for the GitHub platform. TravisCI guarantees testing, building, and deploying of GitHub repositories at commit time in an automatic fashion. For building a project, TravisCI requires the project to specify a *travis.yml* file with all instructions needed for this tool to perform the building correctly. For our systems' projects, the instructions include code compilation, and the execution of EUnit tests and Common Tests. With this information, TravisCI performs the building on a virtual environment and notifies the repository's owner if the building fails.

5.7 Deployment

Our system is intended to be used by an external client, a user who wishes to perform modifications or read at the AntidoteDB database. To reach this demand, the system was built to represent a fully specified and developed system that later could be deployed and exported to the outside world.

Deploying our system consists in exporting the AQL's functionality to the clients, which automatically includes the AntidoteDB's API as well since these two represent a whole. Therefore, in this section we present the deployment solutions for our system, among them the release mode, development mode, the shell mode, and the Docker mode.

5.7.1 Release Mode

As mentioned at the beginning of Section 5.1, our deployment solutions highly rely on Rebar3 to build the system's OTP application. In the context of deployment, Rebar3 allows to build an application release, which represents an aggregated of compiled code, including the source code from the dependencies, and Erlang libraries necessary for the application to be exported and executed in a standalone mode.

Besides the binary files, the release also includes scripts for running the OTP application in an Erlang node given a pre-defined configuration (which includes parametrization for the application's internal sub-applications and servers, or exportable environment variables), for attaching a console to the node, and finally for stopping the whole application.

Creating a release application has the advantage of allowing the application to be exportable to any machine, provided that this machine complies with the Erlang compilation requirements.

5.7.2 Development Mode

As mentioned in the previous section, the system can be initialized through a console that runs over an Erlang node that represents the OTP application itself.

On this console, a user may write native Erlang commands for calling directly exported functions from the modules supported by AQL and AntidoteDB applications, including AQL modules and functions that parse and process queries.

In addition, the execution of this node allows for clients to communicate with the system via *Remote Procedure Calls* (RPCs) or via HTTP requests. In fact, communication between external clients and the database is only possible through RPCs or HTTP messages.

To start a console for the release node, the AQL's project directory includes a *Makefile* with the command `make shell`, to be run on a terminal. The Erlang node that supports the console is identified by a well formatted name, which by default is '*aql*@127.0.0.1'. For using the public IP address for the node (i.e. to make the node public), the command `make shell_public` is issued and the node is initialized under the public address.

5.7.3 Shell Mode

The shell mode represents an alternative way for using the system and allows the user to input directly AQL statements on their raw form (as presented in Section 4.2) for accessing or modifying the database.

This shell is more suitable for database administrators that have permissions to access the server where the system resides and need to perform configurations to the database; it also may be used for debugging. On the other hand, it cannot be directly used by external clients (e.g. database users) since it does not provide client-side services. Nevertheless, even when the system is started in shell mode, external clients can issue queries via RPC or HTTP to the shell, using the AQL's API.

This deployment mode is based on the console provided by the release version mentioned in the previous sections. Thus, to start in shell mode the user inputs the `make` `aqlshell` command (from the *Makefile* on AQL's project directory) on a terminal. Internally, this command runs a script that starts the release node and starts the shell from within the node. By default, the shell runs under the name '*aql*@127.0.0.1'.

5.7.4 Docker Mode

At last, we alternatively deployed our system using the Docker engine. Docker [28] is an engine that allows to deploy systems on a virtualization layer characterized by allowing these systems to run on any machine with any operating system.

In order to proceed with the deployment on Docker, an image containing the AQL and AntidoteDB systems was created. This image is stored on Docker Hub at the address `https://hub.docker.com/r/pedromslopes/aql/`, from where the image must be pulled to use the system. When executed, this image creates a container with all the necessary information for using the previous deploying modes, including the exposure of TCP ports for communication with the system.

5.8 Summary

This chapter maps the designed solution presented in the previous chapter into an implementation environment.

Section 5.1 introduces the main tools used for implementing the proposed solution, including the Erlang functional programming language and its most known framework, the Open Telecom Platform (OTP), important for the development process for this work.

Section 5.2 describes the mapping between the proposed architecture and a programming environment, including the tools used for generating the AQL's syntax in Erlang. In addition, this section explains how AQL and AntidoteDB are coupled together in a single OTP application. Moreover, a HTTP server API is specified for AQL, for remote communication with this interface.

Section 5.3 defines how AQL data is mapped into AntidoteDB abstractions, such as *buckets* and CRDT objects for representing tables and rows, respectively, and embedded CRDT objects for the data types. Table partitioning and index mappings are also explained at this section.

Section 5.4 describes the processing to implement database invariants on the database, resorting to the unique properties of CRDTs.

Section 5.5 details the internal representation of the query optimizer and how partial objects are retrieved from the database.

Section 5.6 states the main techniques for testing our solution, in terms of unit testing, common testing, and continuous integration.

At last, Section 5.7 presents four deployment modes of our solution.

Chapter 6
Evaluation

In this chapter, we present an evaluation of the prototype of our system. The evaluation tries to answer the following questions:

- What is the cost of the proposed solutions for enforcing referential integrity in the system?
- What is the cost of performing updates on indexes?
- How efficient is it to use indexes and partial object readings in range queries?
- What is the overhead of the partitioning mechanism on the database?

Each one of these questions is addressed in one section in this chapter, where the results of our experiments will be presented and analyzed. Before analyzing the results, the first section briefly explains the experimental setup and the benchmarking configurations used for running the experiments.

6.1 Experimental Setup

The experiments were performed in a cluster at the Department of Informatics, FCT-NOVA. The cluster consisted in 12 nodes interconnected by a local network.

In order to run the experiments, 6 out of the 12 machines were used, where 3 machines were servers and the other 3 were clients. In the architectural model of the system, the 3 server machines represent each a data center where resides a deployed Docker version of AQL together with an AntidoteDB data node. The machines have a Quad-Core Intel Xeon X3450, 2.67 GHz processor, 8 GB RAM memory, and two Gigabit Ethernet NIC Broadcom Corporation NetXtreme II BCM5716 network card. Moreover, the machines are configured and prepared with the Docker engine.

The 3 clients submit operations during a limited period of time, in a close loop with no think time between consecutive operations. Each client sticks to a single data center, which means that during the execution of the test, the client issues operations to the same data center. The idea behind this setup is to simulate a geo-replicated setup where a client communicates with a single data center, which is usually the closest geographically. The clients run on machines composed by two Quad-Core AMD Opteron 2376, 2.3 GHz processors, 16 GB RAM memory, and two Gigabit Ethernet NIC Broadcom Corporation NetXtreme BCM5721 network card. Internally, each client runs a deployed Docker image containing a Basho Bench [19] instance.

Given the servers and the clients execute in Docker containers, in order for clients to communicate with servers and for servers to communicate with servers, the 6 machines were set up in a *swarm* cluster [29]. A *swarm* cluster allows a group of Docker containers to belong to an overlay network, where operates a *manager* responsible for joining *workers* into the *swarm*. In our case, it was indifferent which machine would become the manager, for which we choose randomly one machine among the six available for this role.

6.2 Benchmarking

As mentioned in the previous section, each client consists in an instance of Basho Bench [19]. Basho Bench is a benchmarking tool written in Erlang that allows to perform stress tests on a system. This tool has a very intuitive architecture: a test is performed on a Basho Bench instance composed by a main server responsible for spawning N worker[1] processes, which in turn are responsible for issuing operations to a system. In our case, each client runs a Basho Bench instance that issues operations to a single AntidoteDB data center. That instance was deployed into a Docker image, under the tag "aqlapp", that can be found at the DockerHub's address: `https://hub.docker.com/r/pedromslopes/basho-bench-aql/`.

Additionally to the previous architecture, an instance maintains a statistics server that keeps track of throughput and latency data collected throughout the test; the throughput calculates the number of operations performed per second, and the latency calculates the time elapsed from issuing an operation until its response.

A Basho Bench instance performs a test run given a configuration file and a driver. The configuration file determines the duration of the run, the number of workers, the operations to be executed by each worker, and other additional parameters. A driver specifies the operations code to be executed by the workers. For our case, we establish a constant duration of 2 minutes per run, while the number of workers and operations vary depending on the test case. We developed drivers that allow to test the important functionalities of the system, and that allow to execute four types of operations in total:

[1]The term "worker" in Basho Bench cannot be confused with the term "worker" from the Docker's *swarm* engine.

Table 6.1: Common-use workloads, with the three operations and their respective frequencies.

Workload #	*PUT* (%)	*GET* (%)	*DELETE* (%)
1	10	89	1
2	10	90	0
3	45	50	5
4	50	50	0

Table 6.2: Index-oriented workloads, with three operations and respective frequencies.

Workload #	*PUT* (%)	*GET* (%)	*RANGE* (%)
5	10	0	90
6	10	45	45

- ***PUT***: a *put* operation is translated to an INSERT statement over a table and issued to AQL;
- ***GET***: a *get* operation is analogous to perform an AQL's SELECT operation with the WHERE clause consisting in an equality on a generated primary key;
- ***DELETE***: a *delete* operation issues a DELETE statement to AQL for a specified primary key;
- ***RANGE***: a *range* operation performs an AQL's SELECT statement with a WHERE clause consisting in a range query on some given column of a table.

A workload defines the frequency of each operation in an experiment. We identify a total of six workloads to perform the tests that, for convenience, were split into two groups, a group of common-use workloads and a group of index-oriented workloads, represented in tables 6.1 and 6.2, respectively. Throughout the next sections we identify which workloads (or groups of workloads) are more adequate for each test scenario.

Initial setup. To start the 3 Basho Bench clients and the 3 AntidoteDB servers/data centers, a script is executed in one of the client machines. This script is responsible for initiating and connecting the 3 AntidoteDB data centers (through an Erlang script) and, later, for initiating the 3 clients and to assign each to a data center. The three clients start issuing operations at the same time for more precise results. Before the clients can perform the stress test, they are configured with the intended workload for the run, including the number of repetitions per run. We repeated each run by 3 times and performed the mean of the obtained results on each repetition.

Benchmark execution. The experiments use a database schema, known in advance by all instances. That database schema is composed by the tables *Gallery*, *Artist*, and *ArtWork*, with their columns being depicted in Figure 6.1. Depending on the benchmark,

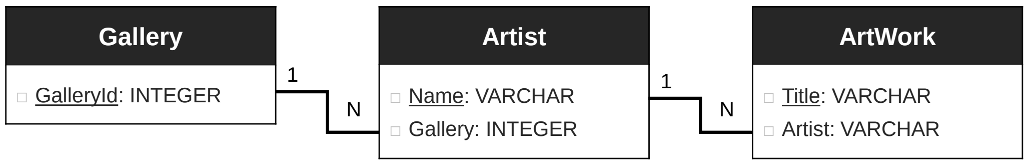


Figure 6.1: Database schema used by common benchmarking configurations.

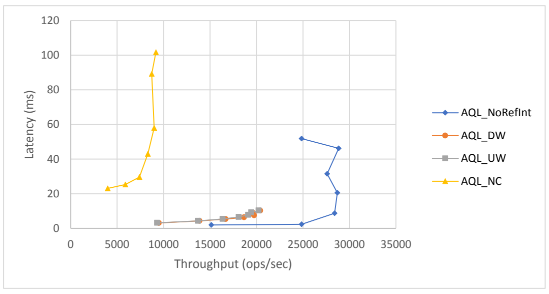
some of the tables or columns may not be used, given the scope of the test. For operating on the database, each worker running on an instance issues an operation enclosed on a new transaction, which means that the transaction is committed and changes are reflected on the database. Additionally, each worker keeps track of the primary keys that it acknowledges from each table, at a point in time. This builds the internal state of the worker and allows to perform operations in the database in a more realistic manner. For instance, when a worker inserts a row in the database, it stores the inserted primary key in its knowledge base. Thereafter, the worker can try to perform a SELECT or DELETE operation on the primary key. At the beginning of the execution, the internal state of a worker is filled with primary keys inserted as a consequence of a population mechanism, that allows the worker to insert a predefined number of rows on the database. In the following benchmarks, we use a pre-run population of 1000 rows in total on the database (covering the three tables), distributed evenly among the workers.

Data generation. All instances are configured to automatically generate primary keys on the insertion of rows. The primary keys are created on a Pareto distribution [13] basis, where 20% of the available keys get selected 80% of the time. For instance, from a maximum of 1000 keys, the keys below 200 will be chosen 80% of the time. We defined a maximum number of keys of five hundred million for our tests. In very rare occasions, it may be necessary to generate a primary key for a DELETE or a SELECT operation in case the worker's internal state is empty. When testing scenarios with referential integrity, the worker uses its internal state to build foreign key references when inserting rows. For instance, when inserting an artist, the worker searches for a random gallery primary key within its stored set of galleries, and use it to relate the artist to the gallery.

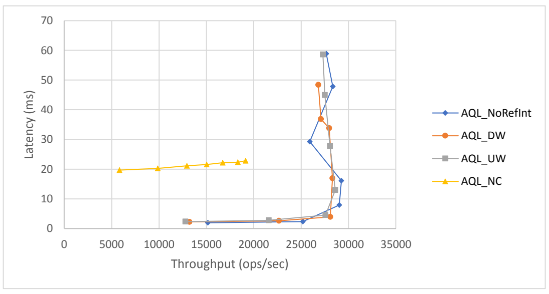
6.3 Referential Integrity Overhead

In this section we present and analyze the results of running the system under the different referential integrity semantics. Recalling Section 4.2.3, we present three semantics for specifying foreign key concurrent behaviors: *update-wins*, *delete-wins*, and *no-concurrency* semantics.

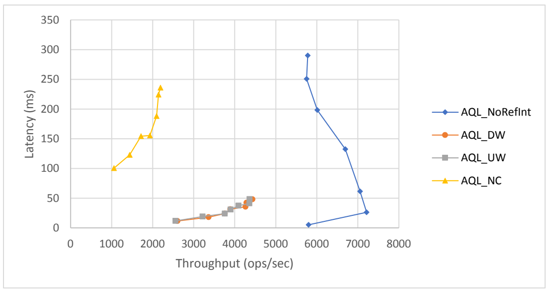
In order to measure the cost of the referential integrity mechanism, the tests will be performed on the standard version of the AQL/AntidoteDB system (*AQL_Base*) and on a modified version of the system specially designed to not consider foreign keys (*AQL_NoRefInt*). In summary, there are 4 initial test cases: one case regarding the



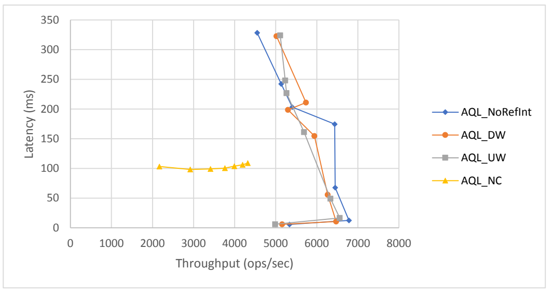
(a) Workload 10-89-1.



(b) Workload 10-90-0.



(c) Workload 45-50-5.



(d) Workload 50-50-0.

Figure 6.2: Scalability of the different mechanisms used for enforcing referential integrity, without cascading behaviors.

use of no referential integrity, and three cases regarding each of the concurrent semantics, the *update-wins* (AQL_UW), *delete-wins* (AQL_DW), and *no-concurrency* (AQL_NC).

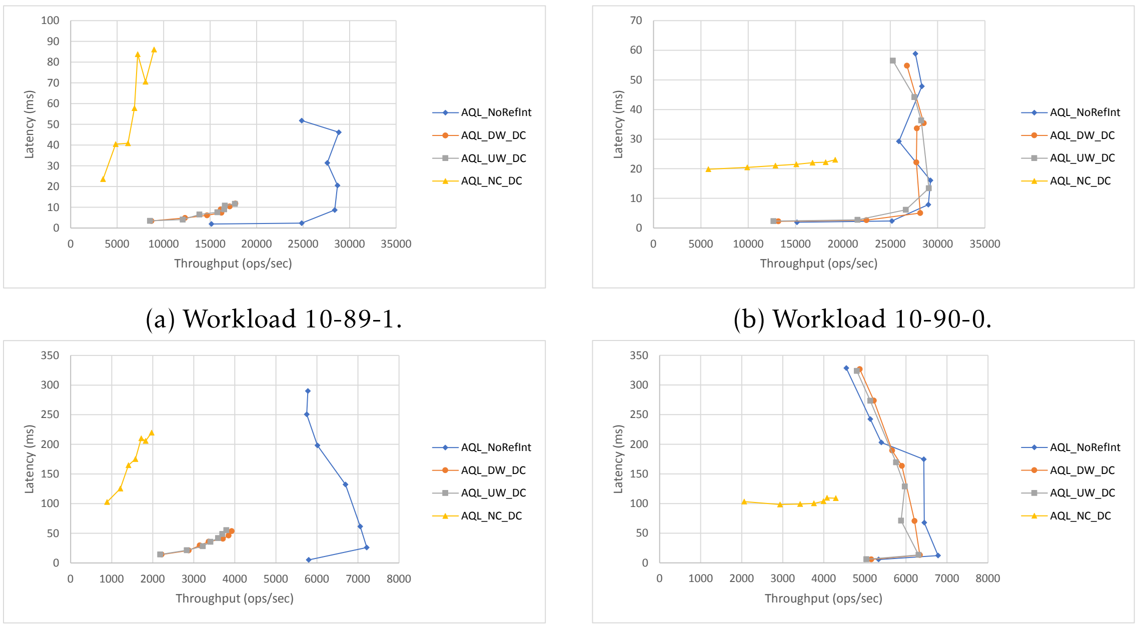
Furthermore, these test cases were performed taking into account two scenarios: the first scenario with no cascade on delete, and the second scenario with cascading behavior (AQL_UW_DC, AQL_DW_DC, AQL_NC_DC, for *update-wins*, *delete-wins* and *no-concurrency*, respectively). We also consider a scenario, similar to the latter, where we test the deployment of secondary indexes on referential integrity with *delete-cascade*.

6.3.1 Cost of Referential Integrity

For measuring the cost of running the system with referential integrity, with and without cascading, we use Basho Bench configuration files that allow to run different versions of our system, given some parameters that concern the several test cases.

The following sections provide the analysis of running those configurations on the four workloads from Table 6.1. The figures presented in those sections show results that correspond to several runs using specific workloads, where each run is represented by a line on a graph. Each run concerns the two aforementioned system's versions, $AQL\_NoRefInt$ and *AQL_Base*, with the variations of the latter version defined on the previous section.

Each result expresses throughput and mean latency values, and each marker represents the execution of a run with a specific number of clients. The number of clients was increased for each experience.



(a) Workload 10-89-1.

(b) Workload 10-90-0.

(c) Workload 45-50-5.

(d) Workload 50-50-0.

Figure 6.3: Scalability of the different mechanisms used for enforcing referential integrity, with cascading behaviors.

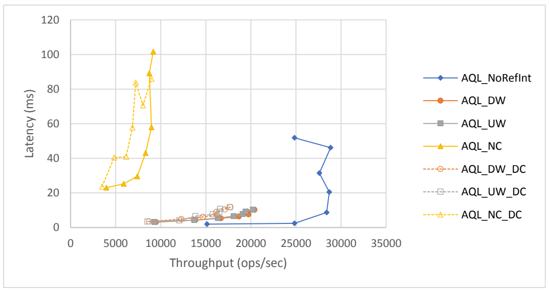
6.3.1.1 Foreign keys without cascading

Figure 6.2 depicts the results of running experiments on the different versions of the system concerning the referential integrity mechanism without cascading behaviors.

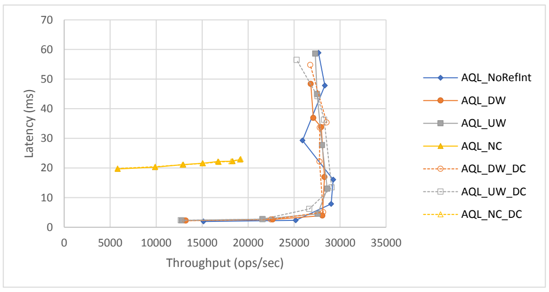
By observing the figure, we note that, in most cases, the *AQL_NoRefInt* solution has the best throughput among all the workloads as expected, since it does not introduce additional write/read overhead on foreign key management. However, it exhibits very high latencies mainly due to *PUT* operations, which increase infinitely the size of primary indexes and consequently the size of the database, making the database slower to process requests.

Concerning solutions with referential integrity employed, the worst results in terms of throughput and latency arise with the *no-concurrency* semantics (the yellow lines from Figure 6.2). It was expected that this semantics would incur in poor performance when compared to the other implementations, given its strong properties (as discussed in the following sections).

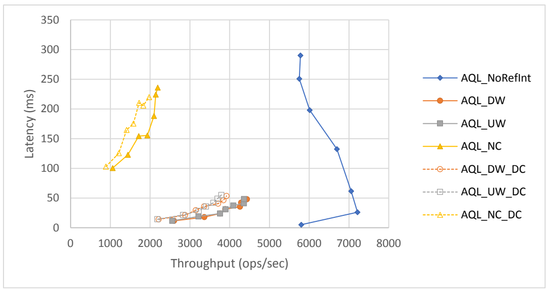
On the contrary, when using the *update-wins* and *delete-wins* approaches with restrict foreign keys, the system achieves a good balance between throughput and latency, being approximately 2 times slower on throughput when delete operations are issued (compared with the *AQL_NoRefInt* solution) and presenting lower throughput with a tiny margin of 0.1% (on worst cases) when no delete operations are issued to the system. This small percentage of the difference between throughputs (without delete operations) proves that our system, even when implementing a referential integrity mechanism, can



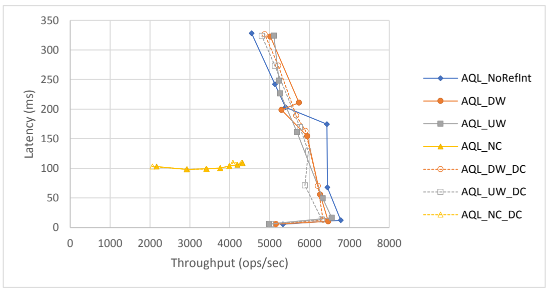
(a) Workload 10-89-1.



(b) Workload 10-90-0.



(c) Workload 45-50-5.



(d) Workload 50-50-0.

Figure 6.4: Comparison on scalability of the proposed mechanisms used for enforcing referential integrity, including mechanisms without and with cascading.

reach as good performance as when running the system without it, representing a very important achievement for this work.

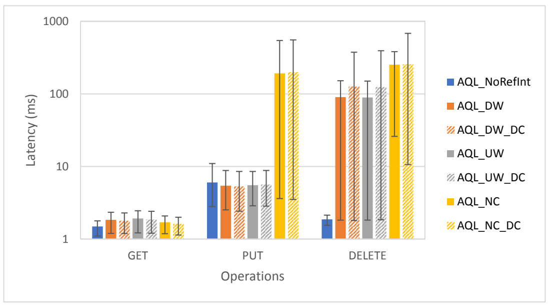
6.3.1.2 Foreign keys with cascading

When using the *delete-cascade* behavior on foreign keys, the system may present slight performance drops, namely on the amount of operations per second. Figure 6.3 shows the computed results for running four versions of the system with cascading semantics.

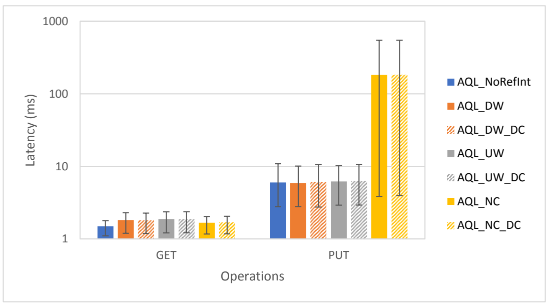
Figures 6.3a (workload 10-89-1) and 6.3c (workload 45-50-5) show more explicitly how applying the *delete-cascade* mechanism incurs some overhead on the system when configured with a referential integrity hierarchy with 3 levels. This overhead is due to the need of updating several levels of the foreign key hierarchies.

Meanwhile, on workloads when no delete operations are issued (workloads 10-90-0 and 50-50-0) the results with *delete-cascade* are similar among each other and very close to the $AQL\_NoRefInt$ solution, which enforces the optimality of our system without delete operations, as already demonstrated in Section 6.3.1.1. These results also enforce the overhead caused by delete operations on the system. Comparing the results shown in Figures 6.3a and 6.3b, for instance, we see that delete operations may decrease throughput by approximately 32% with only 1 percent of delete operations[2].

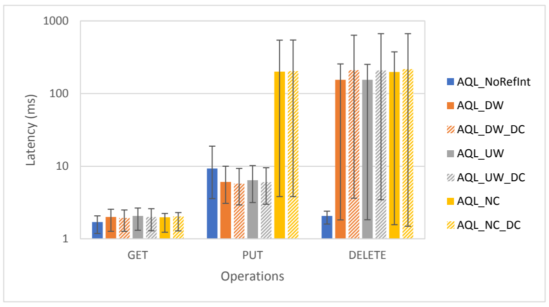
[2]The reader may note that the latency of the first scenario here is lower than that of the second scenario, when it would expect the opposite, given the delete operations are more harmful to the system. As explained in the beginning of Section 6.3.1.1, this is justified by the effects of *PUT* operations when updating the indexes.



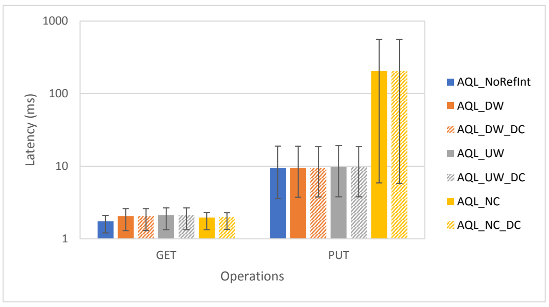
(a) Workload 10-89-1.



(b) Workload 10-90-0.



(c) Workload 45-50-5.



(d) Workload 50-50-0.

Figure 6.5: Comparison of mean latencies, on runs using 3 clients for four workloads, that concern the different versions of the system that enforce referential integrity. The vertical error bars represent the minimum (bottom end of the bar) and 95th percentile (top end of the bar) latencies for each common bar.

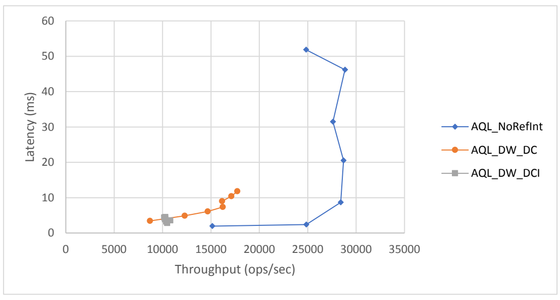
6.3.1.3 Comparison of results on foreign keys

At last, Figure 6.4 compares the results of both solutions without and with cascading behaviors, denoted with solid and dotted lines on each graph of the figure, respectively.

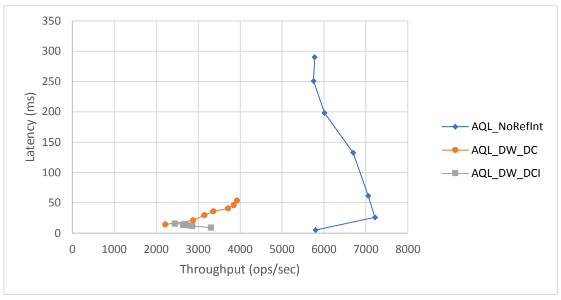
With this figure, we can conclude that using the concurrent foreign key semantics with cascading is more expensive than its inverse, due to the overhead induced by the delete cascade operation (Figures 6.4a and 6.4c). Changing for scenarios where no delete operations are issued (Figures 6.4b and 6.4d), the overhead caused by cascading is practically null, which enforces our previous conclusions about the system's efficiency on these scenarios. With the overall results, we expect performance losses on the system when the table hierarchy increases.

Figure 6.5 complements the analysis of the two previous sections and shows in a better resolution the mean latencies of each operation, from the four workloads tested on this experience, on a test case with only 3 clients issuing operations to the database. The graphs depict the two main test cases, without and with cascading, identified by filled bars and bars with a diagonal pattern, respectively. For each bar, the minimum and 95th latencies are represented on vertical error bars, where one can see the variation for each operation in each scenario.

In the four workload scenarios a pattern is established by the poor performance of the system's version concerning the *no-concurrency* semantics compared to the other versions,

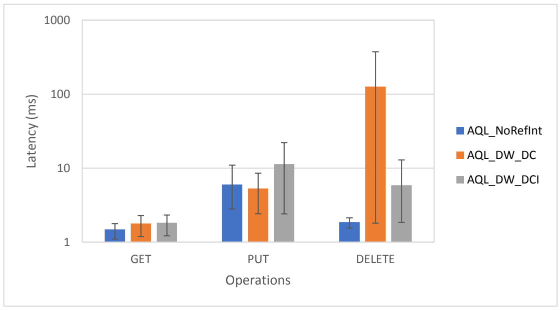


(a) Workload 10-89-1.

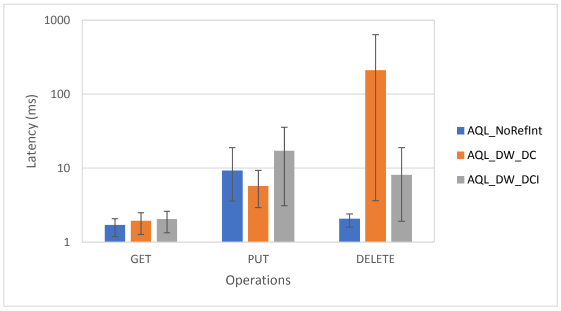


(b) Workload 45-50-5.

Figure 6.6: Comparison on scalability of mechanisms that use and do not use secondary indexes, when enforcing referential integrity.



(a) Workload 10-89-1.



(b) Workload 45-50-5.

Figure 6.7: Comparison of latency values, on runs using 3 clients, concerning versions of the system without referential integrity, with referential integrity and cascading, and with referential integrity and indexes.

which indicates that the coordination necessary between replicas for acquiring and/or releasing locks is still significant for the system (even with a small amount of clients). The low minimum latencies may indicate instants of possible usage of the cache sub-system (briefly mentioned in Section 5.3.2) for reading table metadata, usually used on insert or delete operations, which is faster than reading the whole metadata from the database.

6.3.2 Cost of Indexing on Foreign Keys

This section addresses a more specific test case where we study the cost of foreign keys when deployed with secondary indexes. The idea is to understand how adding secondary indexes to the database schema influences the execution of the referential integrity functionality.

To perform this test we compared the performance of three versions of the system: a version without referential integrity (*AQL_NoRefInt*), a version with foreign keys specifying *delete-cascade* behaviors and *delete-wins* semantics (*AQL_DW_DC*), and a last version where two secondary indexes are created for tables *Artist* and *ArtWork* (*AQL_DW_DCI*). We evaluate this test case on workloads 1 and 3 (from Table 6.1) to address all the main operations.

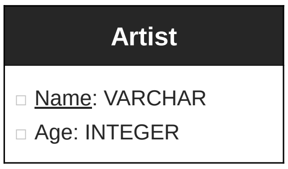


Figure 6.8: Database schema used by index and partition oriented benchmarking configurations.

Figure 6.6 shows the results obtained with the above configurations. The results show an improvement in latency, compared to the base version without secondary indexes (*AQL_DW_DC*), where the numbers do not go beyond 5 milliseconds and 15 milliseconds at the scenarios from Figures 6.6a and 6.6b, respectively. On the other hand, although not much perceptible by the figures, the throughput of this solution tends to be lower and tends to decrease over time, given the workload and the number of clients issuing operations in the system. We justify these values with the burden caused by updating the indexes throughout the execution of the system. Next sections present a more detailed analysis of the indexing system in isolation.

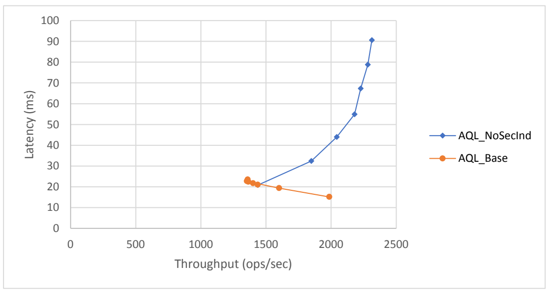
In addition to the performance values, Figure 6.7 depicts the mean latencies, together with minimum and 95th percentile latencies, for this test case. The figure clearly shows the superiority of using indexing on delete operations, compared to base versions. Since deleting a row due to cascading requires to collect dependent rows, querying the database for these rows and taking advantage of indexes for that purpose is much cheaper and presents performance gains. This figure also shows that the latency variations when using indexes are minimal.

6.4 Index Usage Overhead and Efficiency

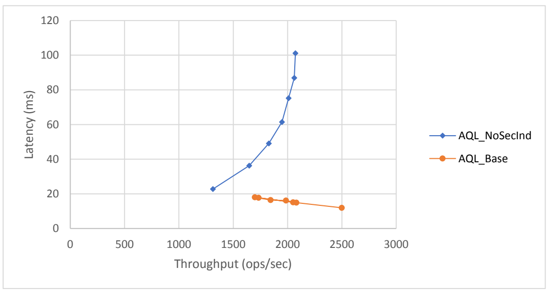
For evaluating the indexing systems proposed in this work, we use two AQL/AntidoteDB versions: the first version is the base version of the system, the *AQL_Base*, used to test the whole functionality of the indexes, and the second version is a modified version of the system, the *AQL_NoSecInd*, that does not make use of indexes on its operation. This means that no secondary indexes are used when processing the queries neither secondary indexes are updated on table updates[3].

Given the focus on this test is the performance of the indexes, the database schema comprises only a table *Artist* with two columns, the *Name* as the primary key column, and the *Age* column, as depicted on Figure 6.8. The column *Age* is the indexed column for this example, i.e. the column for which a secondary index is created. Moreover, the results were obtained from the execution of a set of runs under the workloads from Table 6.2 and workload 3 from Table 6.1.

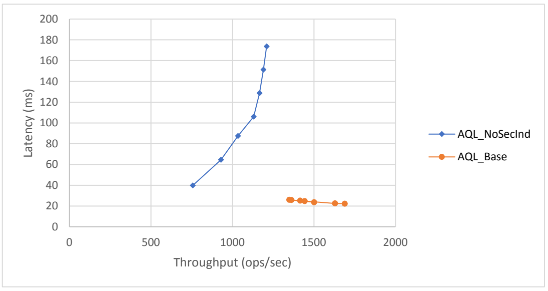
[3]Although secondary indexes are most of the time mentioned in this section, primary indexes functionalities, such their capabilities on processing range queries, are also evaluated on this section.



(a) Workload 10-0-90.



(b) Workload 10-45-45.



(c) Workload 45-50-5 (from Table 6.1).

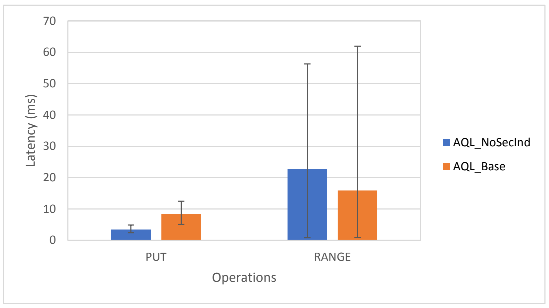
Figure 6.9: Scalability of the system's versions that concern the use and no use of secondary indexes, for reading and writing in the database.

The results are presented in Figure 6.9. For making the figure more clear, the orange lines regarding version *AQL_Base* on each sub-figure are actually decreasing in throughput and increasing in latency, and not decreasing in latency and increasing in throughput. This means that the performance of the system drops as the number of clients increases. This is an indication that making use of secondary indexes may become harmful for the system due to their heavy management and handling. This heaviness is caused by the fact that the database objects, in this case the index objects, only grow in size and never diminish. In our system, indexes grow continuously due to inserting rows on the database, leaving a large amount of data to be processed when reading these objects.

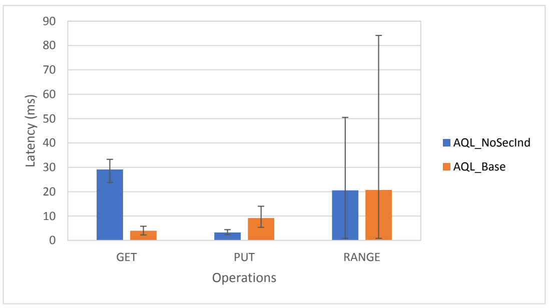
At last, Figure 6.10 concludes this analysis by presenting the mean latency for a run with 3 clients. The presented latency results allow to express the final thoughts on the use of secondary indexes on query processing:

- For read operations, the base system surpasses the version *AQL_NoSecInd*, i.e. without secondary indexes, leading to improvements in the order of the 89 percentage points for GET operations and 15 percent on RANGE operations, and the indexes show their usefulness in scenarios where read operations prevail over write operations;
- The write operations (i.e. PUT and DELETE operations) are much expensive due to the burden of updating the indexes on row updates.

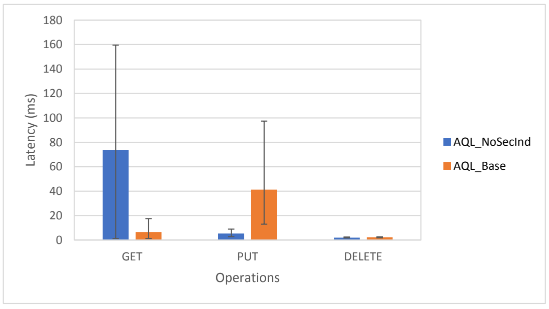
It is important to note as well that range queries present the highest variation among all the operations. The main reason for that variation in our evaluation is that some range



(a) Workload 10-0-90.



(b) Workload 10-45-45.



(c) Workload 45-50-5 (from Table 6.1).

Figure 6.10: Comparison of mean latencies for index-oriented operations, on runs using 3 clients, concerning versions of the system with and without indexing.

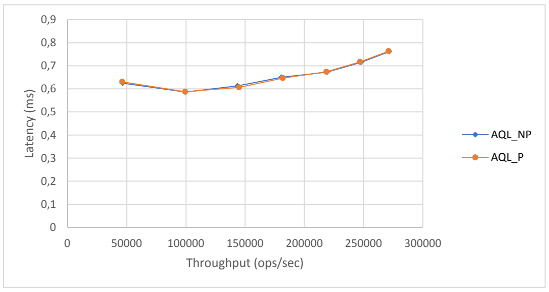
queries require more rows to be filtered than others, which highly depends on the range generated by our benchmarking tool.

6.5 Partitioning Overhead

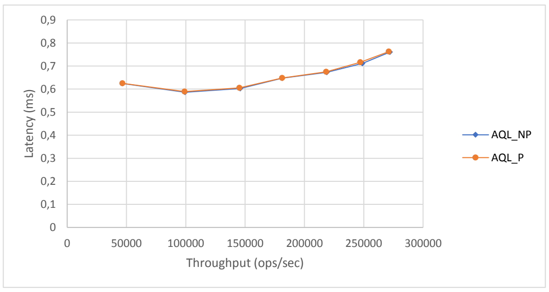
This section evaluates the overhead caused by the use of partitioning on a table. As mentioned in Section 5.3.3, reading a row from a partitioned table often leads to read the primary index of the table in question, because primary indexes store the database keys of rows that, in partitioning, point to values of a partition column.

With the realization of benchmarking tests in this scenario, we aim to analyze the overhead caused by partitioning when it is necessary to resort to indexing to perform read operations. For this purpose we use the *AQL_Base* deployed version of the system with two distinct configurations, where in the first one a table is partitioned by one of its columns (version *AQL_P*), and in the second one the table is presented in its base form, i.e. is not partitioned (version *AQL_NP*). For this benchmark, one table suffices to compare the overhead on both cases, and for that end, table *Artist* is used, where the column *Age* is defined as the partition column for this case. For this experimentation, we used workloads 1 and 3 from Table 6.1.

Figures 6.11 and 6.12 show the performance of the two versions by means of a throughput/latency relation, and the mean latencies of each operation, respectively. The conclusion for the results presented on these two figures is straightforward: the impact of

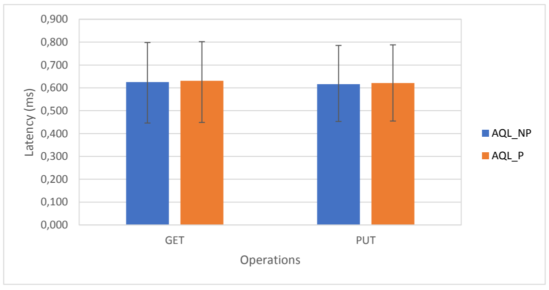


(a) Workload 10-90-0.

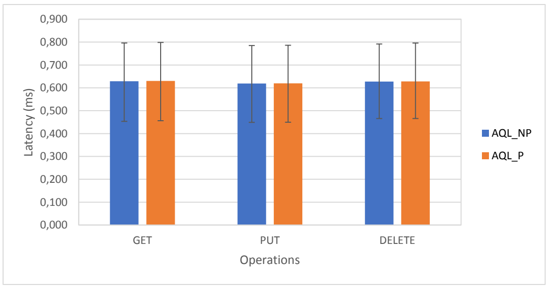


(b) Workload 45-50-5.

Figure 6.11: Scalability of system's versions concerning the use and no use of partitioning on the database.



(a) Workload 10-90-0.



(b) Workload 45-50-5.

Figure 6.12: Comparison of mean latencies, on runs using 3 clients, concerning versions of the system with and without partitioning on the database.

applying partitioning on a table is insignificant in our system. As concluded on the previous section, even when an index is accessed (in this case, the primary index) we expect the access to be fast. Therefore, when the primary index is read to obtain the database key of a row, to posteriorly read that row, the achieved latency is the same as reading directly the row from the database without reading the index in the first place. This proves the efficiency and the good performance of reading keys from the indexes.

6.6 Summary

This chapter reflected the testing procedures performed for the sake of evaluating our system, by following predefined configurations and setups.

Section 6.1 described briefly the setup of machines used in order to test the system, covering an explanation on the communication between machines and how the Docker engine is used for this purpose.

Section 6.2 defined the benchmarking tools and configurations for performing a predefined set of runs, in order to evaluate the system. It also defined the workloads for running experiments and database configurations that allow the system to run in a distributed environment.

Sections 6.3, 6.4, and 6.5 evaluated the AQL/AntidoteDB system regarding the overhead of enforcing referential integrity, without and with cascading behaviors, on the database; the efficiency and cost of secondary indexes when querying and updating the database; and the cost of partitioning in the system, respectively.

The obtained results allowed to observe a noticeable inferiority of using referential integrity with different concurrency semantics over a version of the system without this functionality, mainly when delete operations are issued to the system. This functionality can further aggravate performance when cascading semantics are employed, but can be improved if secondary indexes are created on foreign key columns. When no delete operations are issued to the database, the system has identical performance compared as to not using referential integrity.

While evaluating the indexes overhead, we concluded that the use of secondary indexes can improve the latency of read operations, but can worsen the performance of write operations on the other hand. This verifies due to index objects tending to grow bigger over time.

At last, the results of performing experiments with partitioning revealed that partitioning is not harmful for the system and neither the accesses to the primary indexes for fetching database keys.

Chapter 7

Conclusion

The goal of this work was to continue the development of a SQL interface for the AntidoteDB NoSQL database. To this end, this work proposed new features to both the existing SQL interface, the Antidote Query Language, and to the AntidoteDB database.

This work contributed with the design and implementation of the following main functionalities:

- a referential integrity mechanism, that provides both optimistic and pessimistic approaches for handling concurrency;
- the design of indexes for optimizing the query processing of AntidoteDB. For implementing indexes more efficiently, we introduced a new feature in AntidoteDB: allowing partial read operations to be executed in the server;
- a partitioning mechanism for supporting future extensions that adopt partial replication in the database.

The results from our evaluation show that the referential integrity is still a source of overhead, namely when rows specify the cascading behavior on foreign keys, due to the number of writes this solution performs on foreign key hierarchies. However, the results also showed that our referential integrity solution is optimal without delete operations, presenting a performance very similar to a solution without this mechanism on the database. With the use of a query optimizer on the database side, we obtain good results when querying a database that includes secondary indexes, when compared with versions without the use of indexes. However, the system's performance drops when the indexes are updated, given their complex characteristics. We consider this as a performance trade-off between the choice of better read or write operations on the database.

Future Work. Although AQL is more complete at this stage, we identify a set of additional functionalities for this system as main features to support in the future:

- Support for built-in functions embedded in SELECT statements, such as SUM, AVG (stands for average), and COUNT;
- Support for more complex queries such as joins, unions, and intersections, and other clauses such as LIKE and BETWEEN for comparisons, and ORDER BY for choosing the order of the rows;
- Support for *null* values on the columns of a row, and more complex data types;
- Support for materialized views and composite indexes.

Supporting *null* values as column values highly depends on the AntidoteDB implementation that must support the permanent deletion of database objects, which we consider as future work. Alongside with this feature, it should be considered for future work new behaviors for foreign keys on the delete action, such as "ON DELETE SET NULL" or "ON DELETE SET DEFAULT", for complementing the cascading and restrict behaviors.

For improving performance, we consider the support of a garbage collector for AntidoteDB to delete unnecessary data from the database (which in our system is translated into rows that are not visible) whenever necessary. We also consider to revisit the replication technique used by AntidoteDB, at the bucket level, in order to take better advantage of the partitioning functionality provided by AQL.

A paper for this work has been previously submitted in the context of the Conference on Innovative Data Systems Research (CIDR) [25]. In the future we aim to submit a new and more complete paper concerning this work.

Bibliography

[1] M. K. Aguilera, W. Golab, and M. A. Shah. "A Practical Scalable Distributed B-tree." In: *Proc. VLDB Endow.* 1.1 (Aug. 2008), pp. 598–609. ISSN: 2150-8097. DOI: 10.14778/1453856.1453922. URL: http://dx.doi.org/10.14778/1453856.1453922.

[2] M. K. Aguilera, J. B. Leners, and M. Walfish. "Yesquel: Scalable Sql Storage for Web Applications." In: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP '15. Monterey, California: ACM, 2015, pp. 245–262. ISBN: 978-1-4503-3834-9. DOI: 10.1145/2815400.2815413. URL: http://doi.acm.org/10.1145/2815400.2815413.

[3] D. D. Akkoorath and A. Bieniusa. *Antidote: the highly-available geo-replicated database with strongest guarantees*. Tech. U. Kaiserslautern.

[4] D. D. Akkoorath, A. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro. "Cure: Strong semantics meets high availability and low latency." In: *Int. Conf. on Distributed Computing Systems (ICDCS 2016)*. Nara, Japan: IEEE, June 2016, pp. 405–414. DOI: 10.1109/ICDCS.2016.98. URL: https://hal.inria.fr/hal-01350558.

[5] P. S. Almeida, A. Shoker, and C. Baquero. "Efficient State-based CRDTs by Delta-Mutation." In: *CoRR* abs/1410.2803 (2014). arXiv: 1410.2803. URL: http://arxiv.org/abs/1410.2803.

[6] *Amazon SimpleDB*. https://aws.amazon.com/simpledb/. Accessed: 2018-08-31.

[7] *Antidote CRDT's GitHub Repository*. https://github.com/pedromslopes/antidote_crdt/tree/indexes. Accessed: 2018-09-02.

[8] *AntidoteDB documentation*. http://syncfree.github.io/antidote. Accessed: 2018-01-18.

[9] *AntidoteDB with strong semantics*. https://github.com/SyncFree/antidote/tree/strong_consistency. Accessed: 2018-09-04.

[10] *AntidoteDB's Commit Hooks*. http://syncfree.github.io/antidote/commit_hooks.html. Accessed: 2018-09-02.

[11] *AntidoteDB's GitHub Repository*. https://github.com/pedromslopes/antidote. Accessed: 2018-08-31.

[12] *AQL's GitHub Repository*. https://github.com/pedromslopes/AQL. Accessed: 2018-08-31.

[13] B. C. Arnold. *Pareto Distributions Second Edition*. Chapman and Hall/CRC, 2015.

[14] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. "Highly Available Transactions: Virtues and Limitations." In: *Proc. VLDB Endow.* 7.3 (Nov. 2013), pp. 181–192. ISSN: 2150-8097. DOI: 10.14778/2732232.2732237. URL: http://dx.doi.org/10.14778/2732232.2732237.

[15] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. "Megastore: Providing Scalable, Highly Available Storage for Interactive Services." In: *Proceedings of the Conference on Innovative Data system Research (CIDR)*. 2011, pp. 223–234. URL: http://www.cidrdb.org/cidr2011/Papers/CIDR11_Paper32.pdf.

[16] V. Balegas, D. Serra, S. Duarte, C. Ferreira, M. Shapiro, R. Rodrigues, and N. Preguiça. "Extending Eventually Consistent Cloud Databases for Enforcing Numeric Invariants." In: *Proceedings of the 2015 IEEE 34th Symposium on Reliable Distributed Systems (SRDS)*. SRDS '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 31–36. ISBN: 978-1-4673-9302-7. DOI: 10.1109/SRDS.2015.32. URL: http://dx.doi.org/10.1109/SRDS.2015.32.

[17] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. Preguiça, M. Najafzadeh, and M. Shapiro. "Putting Consistency Back into Eventual Consistency." In: *Proceedings of the Tenth European Conference on Computer Systems*. EuroSys '15. Bordeaux, France: ACM, 2015, 6:1–6:16. ISBN: 978-1-4503-3238-5. DOI: 10.1145/2741948.2741972. URL: http://doi.acm.org/10.1145/2741948.2741972.

[18] V. Balegas, N. M. Preguiça, S. Duarte, C. Ferreira, and R. Rodrigues. "IPA: Invariant-preserving Applications for Weakly-consistent Replicated Databases." In: *CoRR* abs/1802.08474 (2018). arXiv: 1802.08474. URL: http://arxiv.org/abs/1802.08474.

[19] *Basho Bench*. https://github.com/pedromslopes/basho_bench/tree/aql. Accessed: 2018-09-14.

[20] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. "A Critique of ANSI SQL Isolation Levels." In: *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*. SIGMOD '95. San Jose, California, USA: ACM, 1995, pp. 1–10. ISBN: 0-89791-731-6. DOI: 10.1145/223784.223785. URL: http://doi.acm.org/10.1145/223784.223785.

[21] S. Burckhardt, A. Gotsman, and H. Yang. "Understanding Eventual Consistency." In: *Microsoft Research Technical Report MSR-TR-2013-39* (2013).

[22] *Cassandra documentation*. http://cassandra.apache.org/doc/latest/. Accessed: 2018-01-27.

[23] R. Cattell. "Scalable SQL and NoSQL Data Stores." In: *SIGMOD Rec.* 39.4 (May 2011), pp. 12–27. ISSN: 0163-5808. DOI: 10.1145/1978915.1978919. URL: http://doi.acm.org/10.1145/1978915.1978919.

[24] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. "Bigtable: A Distributed Storage System for Structured Data." In: *ACM Trans. Comput. Syst.* 26.2 (June 2008), 4:1–4:26. ISSN: 0734-2071. DOI: 10.1145/1365815.1365816. URL: http://doi.acm.org/10.1145/1365815.1365816.

[25] *CIDR 2019*. http://cidrdb.org/cidr2019/. Accessed: 2018-12-06.

[26] *CQL documentation*. https://docs.datastax.com/en/cql/3.3/index.html. Accessed: 2018-01-28.

[27] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. "Dynamo: Amazon's Highly Available Key-value Store." In: *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*. SOSP '07. Stevenson, Washington, USA: ACM, 2007, pp. 205–220. ISBN: 978-1-59593-591-5. DOI: 10.1145/1294261.1294281. URL: http://doi.acm.org/10.1145/1294261.1294281.

[28] *Docker*. https://www.docker.com/. Accessed: 2018-09-11.

[29] *Docker's Swarm engine*. https://docs.docker.com/get-started/part4/. Accessed: 2018-09-13.

[30] *Elli Erlang web server*. https://github.com/elli-lib/elli. Accessed: 2018-08-31.

[31] *Ericsson Computer Science Laboratory*. https://www.ericsson.com/en. Accessed: 2018-08-31.

[32] *Erlang Programming Language*. https://www.erlang.org/. Accessed: 2018-08-31.

[33] *Erlang's Common Tests (CT) framework*. http://erlang.org/doc/man/common_test.html. Accessed: 2018-09-11.

[34] *Erlang's Crypto module*. http://erlang.org/doc/man/crypto.html. Accessed: 2018-09-02.

[35] *Erlang's EUnit framework*. http://erlang.org/doc/apps/eunit/chapter.html. Accessed: 2018-09-10.

[36] *Erlang's Finite State Machines*. http://erlang.org/doc/man/gen_statem.html. Accessed: 2018-08-31.

[37] *Erlang's General Balanced Trees*. http://erlang.org/doc/man/gb_trees.html. Accessed: 2018-09-04.

[38] *Erlang's Generic Servers*. http://erlang.org/doc/man/gen_server.html. Accessed: 2018-08-31.

[39] *Erlang's Leex module*. http://erlang.org/doc/man/leex.html. Accessed: 2018-08-31.

[40] *Erlang's Yecc module*. http://erlang.org/doc/man/yecc.html. Accessed: 2018-08-31.

[41] C. J. Fidge. "Timestamps in message-passing systems that preserve the partial ordering." In: *Proceedings of the 11th Australian Computer Science Conference* 10.1 (1988), 56–66. URL: http://fileadmin.cs.lth.se/cs/Personal/Amr_Ergawy/dist-algos-papers/4.pdf.

[42] *Fogfish's Segmented In-Memory Cache*. https://github.com/fogfish/cache. Accessed: 2018-09-02.

[43] S. Gilbert and N. Lynch. "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services." In: *SIGACT News* 33.2 (June 2002), pp. 51–59. ISSN: 0163-5700. DOI: 10.1145/564585.564601. URL: http://doi.acm.org/10.1145/564585.564601.

[44] *Git version control software*. https://git-scm.com/. Accessed: 2018-08-31.

[45] *GitHub*. https://github.com/. Accessed: 2018-08-31.

[46] *Graph Databases - Neo4j*. https://neo4j.com/why-graph-databases/. Accessed: 2018-02-15.

[47] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992. ISBN: 1558601902.

[48] F. Hebert. *Learn You Some Erlang for Great Good! A Beginner's Guide*. No Starch Press, Inc, 2013. ISBN: 978-1-59327-435-1.

[49] M. P. Herlihy and J. M. Wing. "Linearizability: A Correctness Condition for Concurrent Objects." In: *ACM Trans. Program. Lang. Syst.* 12.3 (July 1990), pp. 463–492. ISSN: 0164-0925. DOI: 10.1145/78969.78972. URL: http://doi.acm.org/10.1145/78969.78972.

[50] *Intelligent Replication*. http://basho.com/products/riak-kv/intelligent-replication. Accessed: 2018-01-10.

[51] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web." In: *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*. STOC '97. El Paso, Texas, USA: ACM, 1997, pp. 654–663. ISBN: 0-89791-888-6. DOI: 10.1145/258533.258660. URL: http://doi.acm.org/10.1145/258533.258660.

[52] A. Lakshman and P. Malik. "Cassandra: A Decentralized Structured Storage System." In: *SIGOPS Oper. Syst. Rev.* 44.2 (Apr. 2010), pp. 35–40. ISSN: 0163-5980. DOI: 10.1145/1773912.1773922. URL: http://doi.acm.org/10.1145/1773912.1773922.

[53] L. Lamport. “Time, Clocks, and the Ordering of Events in a Distributed System.” In: *Commun. ACM* 21.7 (July 1978), pp. 558–565. ISSN: 0001-0782. DOI: 10.1145/359545.359563. URL: http://doi.acm.org/10.1145/359545.359563.

[54] L. Lamport. “Paxos made simple.” In: (2001).

[55] C. Li, J. Leitão, A. Clement, N. Preguiça, R. Rodrigues, and V. Vafeiadis. “Automating the Choice of Consistency Levels in Replicated Systems.” In: *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*. USENIX ATC’14. Philadelphia, PA: USENIX Association, 2014, pp. 281–292. ISBN: 978-1-931971-10-2. URL: http://dl.acm.org/citation.cfm?id=2643634.2643664.

[56] 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: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP ’11. Cascais, Portugal: ACM, 2011, pp. 401–416. ISBN: 978-1-4503-0977-6. DOI: 10.1145/2043556.2043593. URL: http://doi.acm.org/10.1145/2043556.2043593.

[57] 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. Lombard, IL: USENIX Association, 2013, pp. 313–328. URL: http://dl.acm.org/citation.cfm?id=2482626.2482657.

[58] H. Lu, C. Hodsdon, K. Ngo, S. Mu, and W. Lloyd. “The SNOW Theorem and Latency-optimal Read-only Transactions.” In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI’16. Savannah, GA, USA: USENIX Association, 2016, pp. 135–150. ISBN: 978-1-931971-33-1. URL: http://dl.acm.org/citation.cfm?id=3026877.3026889.

[59] M. Mehta and D. J. DeWitt. “Data Placement in Shared-nothing Parallel Database Systems.” In: *The VLDB Journal* 6.1 (Feb. 1997), pp. 53–72. ISSN: 1066-8888. DOI: 10.1007/s007780050033. URL: http://dx.doi.org/10.1007/s007780050033.

[60] *MySQL Triggers syntax*. https://dev.mysql.com/doc/refman/5.5/en/trigger-syntax.html. Accessed: 2018-08-13.

[61] C. H. Papadimitriou. “The Serializability of Concurrent Database Updates.” In: *J. ACM* 26.4 (Oct. 1979), pp. 631–653. ISSN: 0004-5411. DOI: 10.1145/322154.322158. URL: http://doi.acm.org/10.1145/322154.322158.

[62] A. Pavlo, C. Curino, and S. Zdonik. “Skew-aware Automatic Database Partitioning in Shared-nothing, Parallel OLTP Systems.” In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’12. Scottsdale, Arizona, USA: ACM, 2012, pp. 61–72. ISBN: 978-1-4503-1247-9. DOI: 10.1145/2213836.2213844. URL: http://doi.acm.org/10.1145/2213836.2213844.

[63] N. Preguiça, J. L. Martins, M. Cunha, and H. Domingos. "Reservations for Conflict Avoidance in a Mobile Database System." In: *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services*. MobiSys '03. San Francisco, California: ACM, 2003, pp. 43–56. DOI: 10.1145/1066116.1189038. URL: http://doi.acm.org/10.1145/1066116.1189038.

[64] *RabbitMQ*. http://www.rabbitmq.com/. Accessed: 2018-08-31.

[65] *Rebar3*. http://www.rebar3.org/. Accessed: 2018-08-31.

[66] *Redis documentation*. https://redis.io/documentation. Accessed: 2018-01-08.

[67] *RediSQL online code repository*. https://github.com/RedBeardLab/rediSQL. Accessed: 2018-01-29.

[68] *RiakKV features*. http://basho.com/products/riak-kv. Accessed: 2018-01-20.

[69] *RiakKV Secondary Indexes*. http://docs.basho.com/riak/kv/2.2.3/developing/usage/secondary-indexes/. Accessed: 2018-08-23.

[70] E. Schurman and J. Brutlag. *Performance Related Changes and their User Impact*. Talk at Velocity '09.

[71] O. Shacham, F. Perez-Sorrosal, E. Bortnikov, E. Hillel, I. Keidar, I. Kelly, M. Morel, and S. Paranjpye. "Omid, Reloaded: Scalable and Highly-available Transaction Processing." In: *Proceedings of the 15th Usenix Conference on File and Storage Technologies*. FAST'17. Santa clara, CA, USA: USENIX Association, 2017, pp. 167–180. ISBN: 978-1-931971-36-2. URL: http://dl.acm.org/citation.cfm?id=3129633.3129649.

[72] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. *A comprehensive study of Convergent and Commutative Replicated Data Types*. Research Report RR-7506. Inria – Centre Paris-Rocquencourt ; INRIA, Jan. 2011, p. 50. URL: https://hal.inria.fr/inria-00555588.

[73] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. "Conflict-free Replicated Data Types." In: *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*. SSS'11. Grenoble, France: Springer-Verlag, 2011, pp. 386–400. ISBN: 978-3-642-24549-7. URL: http://dl.acm.org/citation.cfm?id=2050613.2050642.

[74] A. Silberschatz, H. Korth, and S. Sudarshan. *Database Systems Concepts*. 6th ed. New York, NY, USA: McGraw-Hill, Inc., 2010. ISBN: 0073523321, 9780073523323.

[75] J. P. Sousa. "SQL-like Interface for Antidote Geo-replicated Key-Value Store." Master's thesis. Faculdade de Ciências e Tecnologia, Universidade NOVA de Lisboa, 2017.

[76] 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*. SOSP '11. Cascais, Portugal: ACM, 2011, pp. 385–400. ISBN: 978-1-4503-0977-6. DOI: 10.1145/2043556.2043592. URL: http://doi.acm.org/10.1145/2043556.2043592.

[77] B. Sowell, W. Golab, and M. A. Shah. "Minuet: A Scalable Distributed Multiversion B-tree." In: *Proc. VLDB Endow.* 5.9 (May 2012), pp. 884–895. ISSN: 2150-8097. DOI: 10.14778/2311906.2311915. URL: http://dx.doi.org/10.14778/2311906.2311915.

[78] *SQL Foreign Keys syntax*. https://docs.oracle.com/cd/E17952_01/mysql-5.6-en/create-table-foreign-keys.html. Accessed: 2018-06-30.

[79] *SQLite website*. https://www.sqlite.org/. Accessed: 2018-01-29.

[80] *SyncFree Project*. https://pages.lip6.fr/syncfree/index.html. Accessed: 2018-01-18.

[81] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. "Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System." In: *SIGOPS Oper. Syst. Rev.* 29.5 (Dec. 1995), pp. 172–182. ISSN: 0163-5980. DOI: 10.1145/224057.224070. URL: http://doi.acm.org/10.1145/224057.224070.

[82] *TravisCI - Continuous Integration tool*. https://travis-ci.org/. Accessed: 2018-09-11.

[83] R. Vilaça, F. Cruz, J. Pereira, and R. Oliveira. "An effective scalable SQL engine for NoSQL databases." In: *IFIP International Conference on Distributed Applications and Interoperable Systems*. Springer. 2013, pp. 155–168.