Geo-Replicated Buckets

An Optimistic Geo-Replicated Shim for Key-Value Store

João Miguel Louro Neto

Master of Science Thesis
Stockholm, Sweden 2015

TRITA-ICT-EX-2015:226
Geo-Replicated Buckets

An Optimistic Geo-Replication Shim for Key-Value Stores

JOÃO MIGUEL LOURO NETO

Master’s Thesis
Supervisor: Vianney Rancurel (Scality)
Supervisor: Marc Shapiro (INRIA, LIP6)
Examiner: Johan Montelius (KTH)

2015-11-12

TRITA-ICT-EX-2015:226
Abstract

This work introduces GeoD and VersionD. GeoD is a causally-consistent geo-replication shim for key-value stores. GeoD enables the separation of concerns regarding replication and convergence from key-value stores, while preserving all of their read-only functionality. VersionD extends the feature set of GeoD by providing a versioning API, allowing applications to store multiple values for a given object, and to use context-specific conflict-resolution rules. We also discuss an efficient architecture for optimistically replicated systems, where all the storage replicas persist data in a common shared DHT. Since most key-value stores don’t support versioning and GeoD relies on versioning support from the key-value store, we developed a shim layer that also augments the traditional core API with extra versioning operations and logic.
## Contents

1 Introduction 1

2 Background 3
   2.1 Replication 3
      2.1.1 Objects, Replicas and Sites 3
      2.1.2 Geo-Replication 3
      2.1.3 Optimistic Replication 4
   2.2 Time and Ordering of Events 4
   2.3 Consistency and Conflicts 4
      2.3.1 Strong Consistency 5
      2.3.2 Weak Consistency 5
      2.3.3 Hybrid Consistency Models 6
   2.4 Distributed Hash Table 6
   2.5 Range Queries 7
   2.6 Key-Value Stores 7
      2.6.1 Bucket-Key-Value Store 8
      2.6.2 Versioned Key-Value Store 8
   2.7 Existing NoSQL Databases 8

3 Optimistic Replication Shim 11
   3.1 Causality Tracking 12
   3.2 Conflict Handling and Merging Semantics 12
      3.2.1 Delta Mutators 14
   3.3 Recovering Deleted Data 15
   3.4 Dissemination and Garbage Collection 15
   3.5 Extensions 16

4 GeoD Implementation 17
   4.1 Overview 17
   4.2 GeoD Protocol 18
   4.3 API Translation 19
   4.4 Delta Management 20
   4.5 Propagator 21
Chapter 1

Introduction

Running large Key-Value Stores out in the real world is a challenging task: processing thousands of requests per second on a petabyte-scale data set while keeping response times to a minimum in order to provide excellent user experience is no easy task. Geo-Replication is one of the key features of such systems – it allows the masking of failures while providing lower latency for operations, resulting in increased uptime and a better experience for users all around the globe.

However, not all key-value stores support this feature and, when available, it is usually not flexible enough to suit the user’s needs – they usually employ classical forms of replication (synchronous or single-master), which are unsuited for large-scale applications.

A more scalable alternative to the traditional geo-replication mechanisms is optimistic replication (also known as lazy replication), a strategy in which replicas are allowed to commit operations locally and operate asynchronously. The main principle behind it is to do away with the total order of events, relaxing it into partial ordering. While it increases the complexity for programmers due to the introduction of concurrent events, it allows replicas to accept write operations at any time, including during network partition scenarios.

To this end we designed and developed GeoD, a shim that tackles the problem of providing optimistic replication to key-value stores. GeoD runs on top of independent key-value store instances and replicates all data asynchronously without any modification. It does so by intercepting requests from clients to the datastore, detecting the resulting internal state transitions and propagating them to the other sites. To save bandwidth, these transitions are buffered and aggregated before being sent. GeoD increases slightly the latency of requests, as it adds some processing delay to the critical path. Furthermore, the implementation described here is very general and covers only a small set of the common operations in the key-value stores.

The built-in design for conflict resolution follows two principles: it should be as familiar to the user as possible (i.e. it should behave similarly as if it was non-replicated), and should prevent unwanted loss of information – any deletion that is concurrent with a write to the same object will be ignored. Causality is tracked
by resorting to version vectors. In case two values are written concurrently, the one with the largest physical timestamp wins (last-write-wins policy).

A typical desired feature of optimistically replicated systems is versioning – instead of storing a single value for each object, we may want to store multiple versions for logging or, more importantly, to support application-specific merge semantics. These allow for applications with complex merge semantics to have full control on how conflicting values are merged. To this end, we also developed VersionD, an upgrade to the basic GeoD that extends the client API with a versioning API.

VersionD enables legacy applications to make use of the built-in conflict resolution of GeoD while simultaneously allowing applications with specific merge semantics to enforce them. The downside to VersionD is the increased number of round-trips to the data store, which can impact negatively the latency of requests. The VersionD presented here also requires the data store to support prefix queries in order to provide some of its functionality (e.g. retrieving all the versions of a key), but simple workarounds can be designed.

Finally, we present an architecture for optimistically replicated data stores, by using two distinct storage components – a common data repository, shared by all the sites, and a private metadata repository, optimistically replicated at all sites. The main advantage of such an architecture is the massive reduction in synchronization overhead – instead of synchronizing a large file between all the sites, one just needs to broadcast it’s metadata (typically a few bytes). This allows for synchronization messages to be very small, thus reducing bandwidth costs and propagation latency and, consequently the number of conflicting operations.

We discuss how to perform collision-free writes to the shared data repository without coordination. Efficient access to data is considered a sub-problem and is not covered in this work.

The contributions of this thesis are GeoD and VersionD, which allow to extend the feature set of key-value stores with optimistic replication. While GeoD only stores a single value per key, VersionD stores a set of values (called versions) to allow greater expressiveness in conflict resolution. We finish by presenting an architecture that allows for low-bandwidth low-latency propagation of updates between the different geo-locations.
Chapter 2

Background

This chapter introduces optimistic replication alongside with other relevant concepts.

2.1 Replication

Replication is a strategy in which multiple copies of an object are stored in multiple machines for increased availability, durability and either performance or reliability. By storing copies of data in separate machines, if one fails, a system can continue operating using the replicated data.

2.1.1 Objects, Replicas and Sites

An object is the fundamental entity in the system. They are the minimum granule of data in the system, and the minimal unit of replication. A replica is a copy of an object stored in a site, or in a computer. A site may store replicas of multiple objects, but I will use the terms replica and site interchangeably, as the algorithms discussed here manage each object independently.

2.1.2 Geo-Replication

Geo-replication is a specific use-case of replication in which the replicas of an object are scattered over different geographic sites. In this scenario we assume network characteristics poorer than in the single-site scenario: the latency between the replicas is a few orders of magnitude higher, the bandwidth is much more limited and link failures are more unpredictable.

By having a copy close to clients, they will experience improved system uptime and lower access latency [78]. One of the most popular use-cases is in the Domain Name System [26], where having multiple copies of the records of a name results in near-perfect uptime, even in the presence of denial-of-service attacks.
2.1.3 Optimistic Replication

Optimistic replication \[71, 72\] is a group of techniques for efficient data-sharing in wide-area or mobile environments. They differ from traditional pessimistic database replication techniques \[42\] in the way they handle concurrency. While pessimistic algorithms synchronously coordinate replicas during accesses and block other users during an update, optimistic replication allows data to be accessed without prior synchronization. Updates are propagated in the background, and conflicts are handled after they are detected. The advantages include improved availability, as applications continue to make progress even under the presence of arbitrary network failures, and scaling linearly to a large number of replicas due to the reduced synchronization required between the sites.

2.2 Time and Ordering of Events

In a local environment, time and ordering is trivial. For time, we can rely on the local physical clocks, and use them to provide ordering of events in the system. However, in a replicated environment, clocks are unreliable \[62, 75\]. Logical clocks \[38, 59\] were developed to tackle the problem, but only provide a partial ordering of events. Vector clocks \[23, 51\] provide a mechanism to figure out a partial ordering of events based on their causality. Version vectors \[14, 17, 46, 58, 61, 68\] are similar to vector clocks; however, their purpose is not to order the events in a distributed system, but to track the causality between the events in replicas. These have been shown to be optimal \[31\] in size – to unequivocally track causality there must be one entry per source of concurrency. More recently, interval tree clocks \[16\] were introduced, which generalize the above mechanisms for dynamic systems, where processes can be created or retired in a decentralized fashion.

2.3 Consistency and Conflicts

The definition of conflict is inseparable from that of consistency. A conflict occurs when a set of concurrent operations violate consistency. For example, if two replicas concurrently execute two distinct put operations to the same key with different values, the one executed the latest will replace the value of the earliest. If the operations are not applied in the same order, the replicas will diverge indefinitely. The system is said to be consistent if all the replicas are in the same logical state.

In a single-replica environment, its state management and storage is centralized. Independently of the number of operations, concurrent or not, there is at all times a consistent and unique state – each user will see a consistent view of the data. In single-master replicated environment (e.g. DNS \[26\]), replicas are kept consistent (except for brief periods due to message propagation delay), but remain consistent in the long run.
However, on multi-master systems, replicas may diverge without bound if operations conflict. Let us take a replicated key-value data store as a concrete example. If two replicas concurrently set the value of an object to different values, what should be the final outcome? If two replicas execute the operations in the order they receive them (i.e., first the local update, and later the one from the remote replica), they will diverge into different states.

There are two ways to overcome this: either synchronize the replicas (by ordering the conflicting updates), or designing the operations in such way that conflicts do not occur. We call these two families of models strong consistency and weak consistency, respectively.

Under a network partition scenario, the system can be either completely available, or kept strongly consistent. This has been popularized as the CAP theorem [27, 39, 41, 60, 80, 81].

### 2.3.1 Strong Consistency

Strong consistency requires some form of global agreement on the order of updates, and the user will have a similar experience to that of a single-master replicated system. All replicas return the same value when queried (except for the message propagation delay mentioned before). However, operations must perform synchronization in order to decide on the outcome of concurrent conflicting operations.

### 2.3.2 Weak Consistency

Weak consistency allows for replicas to execute updates without the constant synchronization imposed by strong consistency – we call it disconnected operation. Weak consistency, however, imposes some limitations in the outcome of concurrent sets of operations. In order to provide disconnected operation, all the operations must be designed in a way so that they are associative, commutative and idempotent [74].

In addition to this limitation, some of the system’s invariants may not be respected if we allow all operations to be executed without synchronization [24, 52]. One such example of an invariant is ‘the balance of an account must be non-negative’ in a bank application: different replicas can withdraw amounts whose sum is greater than the total balance, causing it to be negative after synchronization.

We briefly describe relevant weak consistency models.

### Eventual Consistency

Eventual consistency simply states informally that if no updates are made to an object, all replicas will eventually converge. It does not define the procedure that achieves convergence, nor any methodology. It is the weakest consistency model. The other models here discussed can be seen as strengthenings of eventual consistency.
Strong Eventual Consistency and CRDTs

Strong eventual consistency states that replicas that have received the same set of updates are in the same state.

A Conflict-free Replicated Data Type (CRDT [74]) is a principled approach into designing a replicated data type providing strong eventual consistency semantics. CRDTs can be designed using two frameworks – operation-based and state-based. In the operation-based framework, concurrent operations are made commutative. In the state-based framework, states are merged in such a way they form a monotonic semi-lattice. The two approaches have been shown to be equivalent, as one can emulate the other. Delta-CRDTs [18] are an approach for deriving efficient state-based CRDTs, by computing how the state changes for each of the operations executed by the user.

Causal Consistency

Causal consistency employs causal (happens-before [13]) order [73] between the updates, for example through the use of Version Vectors [46]. Updates that are ordered do not conflict – all replicas will apply them in the same order (similar to what is accomplished by strong forms of consistency). However, two concurrent operations may still cause a conflict (as they are not causally ordered). In this case, the resolution of the conflict is not defined.

Causal+ Consistency

Causal+ consistency is a hybrid of Causal Consistency and Strong Eventual Consistency – uses version vectors to capture the causal order of updates, but in case of a conflict, employs automatic conflict resolution. One such example of conflict resolution is in GentleRain [36], where in addition to having one version vector per operation, a physical timestamp is also attached as a tie-breaker to decide on a total ordering of operations.

2.3.3 Hybrid Consistency Models

In order to satisfy global invariants, hybrid consistency models have been developed. Red-Blue Consistency [24, 52] splits the set of operations into red operations (those that may violate global invariants if executed concurrently and therefore must be executed with coordination between all replicas) and blue operations (that can be executed locally before being later asynchronously propagated). Naturally, red operations are unavailable under network partitioning scenarios, the system being able to execute blue operations only.

2.4 Distributed Hash Table

A Distributed Hash Table (DHT) is a peer-to-peer (P2P) systems that are scalable, robust and self-organizing. They provide a lookup service similar to that of a hash
table. Nodes are arranged in a ring, each taking responsibility for $1/n$ of the keyspace on average. Nodes are assigned a position in the ring (e.g. by hashing their identifier) and take responsibility for the interval between its predecessor’s position (the node with the position before in the ring) and its own. Responsibility of mapping the keys to values is distributed among the nodes in such a way that if the set of participants changes, the system recovers with minimal disruption.

Chord [77] is among of the first implementations of a DHT. Chord employs consistent hashing [47] to partition its 160-bit key space and to balance the load between the participating nodes.

## 2.5 Range Queries

A range query is a common database operation that retrieves all records where some key is between an upper and a lower boundary. In order to these queries efficiently, we require an efficient indexing of the keys.

Some storage systems opt to store all the records sorted. For example, in BigTable [29] the key space is partitioned into tablets, each storing a consecutive subset of the possible keys, and each tablet is mapped onto a single machine. This technique may cause hot-spots, as the more popular regions of data will fall on the same set of machines.

Other systems have a separate search structure called an index. Indices are a copy of the list of keys that can be searched efficiently. A classic data-structure used in the implementation of indices is the B-tree [25]. These, however, are unsuited for concurrent operation – for correctness, no more than a single access at a time can be done to a given subtree. No variant of these structures is known to work well in a multi-master replication environment.

DHTs are excellent candidates to be at the core of a key-value store. However, due to the lack of locality, they require a separate index to be built in order to support efficient range queries [19]. For this purpose, specialized data structures like the Prefix Hash Tree [69] were developed. However, all of them have the same limitations of the classical counterparts – concurrent updates to the same subtree may violate the structure’s invariants and require updates to be serialized.

## 2.6 Key-Value Stores

A Key-value store is the simplest type of NoSQL database [63]. It is a simple map: store a set of objects, each uniquely identified by a key. We will use oid (Object Identifier) to refer to the key of an object. The core API provided by a Key-Value store can be seen in table 2.1.
Operation | Description
---|---
addKey(oid, value) | Sets the contents of the object oid
deleteKey(oid) | Deletes the object oid
ggetKey(oid) | Retrieves the contents of object oid

Table 2.1. Core operations on Key-Value data stores

2.6.1 Bucket-Key-Value Store

The Bucket-Key-Value Store (or Bucket Store) model augments the Key-Value Store by introducing the concept of buckets. A bucket is a logical container for storing objects that organizes the namespace at the highest level, each with a unique identifier. In practical terms, each can be seen as an independent instance of a key-value store. This is the model followed by Amazon S3. The core API provided by Bucket-Key-value stores can be seen in table 2.2. Throughout the report we will use bid (Bucket Identifier) to refer to the identifier of a bucket.

2.6.2 Versioned Key-Value Store

A Versioned Key-Value Store not only stores and exposes the latest version of the value associated with a given key, but keeps a history of values that have been assigned to it, called versions. Each version encodes the state of the object at a point in time and space. Versioning support is explicitly part of the data model and exposed by the API. The versioning API can be seen in table 2.3.

2.7 Existing NoSQL Databases

Key-Value stores are the simplest of NoSQL databases. We briefly discuss relevant details of some of the current implementations.

Operation | Description
---|---
createBucket(bid) | Creates a new bucket with the identifier bid
dropBucket(bid) | Drops the bucket with the identifier bid
addKey(bid, oid, value) | Sets the contents of the object oid in the bucket bid
deleteKey(bid, oid) | Deletes the object oid in the bucket bid
ggetKey(bid, oid) | Retrieves the contents of the object oid in the bucket bid

Table 2.2. Core operations on Bucket-Key-Value data stores
2.7. EXISTING NOSQL DATABASES

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addVersion(bid, oid, vid, value)</td>
<td>Adds a version vid of object oid in bucket bid</td>
</tr>
<tr>
<td>deleteVersion(bid, oid, vid)</td>
<td>Deletes version vid of object oid in bucket bid</td>
</tr>
<tr>
<td>getVersion(bid, oid, vid)</td>
<td>Retrieves version vid of object oid in bucket bid</td>
</tr>
<tr>
<td>getLatest(bid, oid)</td>
<td>Retrieves the set of latest versions of object oid in the bucket bid</td>
</tr>
</tbody>
</table>

Table 2.3. Versioning API for Bucket-Key-Value Stores

Amazon Dynamo

Dynamo [33] is Amazon’s highly-available key-value store. Amazon uses a highly
decentralized, loosely coupled service-oriented architecture consisting of hundreds of
services. For the shopping cart service, they require storage technologies that are
always available: “customers should be able to view and add items to their shopping
cart even if disks are failing, network routes are flapping, or data centers are being
destroyed by tornadoes.” Therefore, this system requires a data-store that is always
both readable and writable, and that it’s data needs to be available across multiple
data centers.

Dynamo, however, targets applications that require only key/value access with
primary focus on high availability. No powerful features such as prefix queries are
required by the applications, nor are they provided by Dynamo.

Many traditional data stores execute conflict resolution during writes and keep
the read complexity simple. However, in such systems, writes may be rejected if the
data store cannot reach all (or a quorum) of the replicas at a given time. Dynamo
targets the design space of an “always writable” data store. This requirement forces
the pushing of complexity of conflict resolution from the writes to the reads in
order to ensure that writes are never rejected. In Dynamo, conflict resolution can be
done either by the data store or by the application. If done by the data store, only
simple policies can be used (such as last-writer-wins). However, if delegated to the
application, more complex merge policies can be used with the added complexity of
managing this merging.

Voldemort

Voldemort [9] is a distributed key-value storage system, used at LinkedIn [5] by
numerous critical services powering a large portion of the site, based on Amazon
Dynamo.

COPS

COPS [54] is a key-value store providing Causal+ Consistency. COPS can enforce
causal dependencies between keys stored across an entire cluster, rather than a single
server as previous systems. COPS checks whether causal dependencies between keys are satisfied in the local cluster before exposing writes. Further, in COPS-GT (an extension of COPS) they introduce read-only transactions in order to obtain consistent views of multiple keys without locking.

Netflix Dynomite

Dynomite [8] is an open-source project by Netflix that provides replication for key-value stores. It is a generic Dynamo implementation that can be used with many different key-value pair storage engines. It supports multi-site replication and is designed for high-availability. It uses an operation-based approach: changes broadcasted to other replicas take the form of a log of operations. However, it provides no consistency guarantees whatsoever – operations may be reordered, or executed more than once, and no efforts are made to change their semantics to make them associative, commutative nor idempotent. Only applications that do not require any consistency guarantees may be used with Dynomite, such as publishing time series data.

RiaK

RiaK [10] is a distributed key-value store offering high scalability, fault tolerance and scalability. Its design is very influenced by Amazon Dynamo’s. In addition to storing binary objects, RiaK allows users to choose among several types of CRDT objects, whose convergence is then automatically handled by RiaK.

Amazon S3

Amazon Simple Storage Service (S3) [1] is a bucket-key-value cloud storage service. Amazon S3 supports retrieval of an object by using the full object key or using a prefix.

GentleRain

GentleRain [36] is a causally consistent replicated data-store, providing throughput comparable to that of an eventually-consistent data-store. GentleRain

SwiftCloud

SwiftCloud [82] is a data storage system for cloud platforms that spans both client nodes (that cache a subset of the objects) and datacenter servers (that replicate every object).
Chapter 3

Optimistic Replication Shim

In this chapter we present and detail the design of GeoD, an optimistic geo-replication shim for versioned bucket-key-value stores. It provides the following:

**Availability** Even in the event of node or network failures, the system must remain available. Any distributed system in the presence of network failures faces a trade-off between availability and consistency \cite{27, 39, 41, 66, 80, 81}. With this in mind, for writes to succeed under an arbitrary inter-DC network partition scenario, the different replicas must be allowed to diverge, albeit temporarily. For this reason, we employ a weak consistency model.

**Perseverance of read-only functionality** The set of read-only operations (i.e. those not causing any changes in the internal state) that can be provided by the key-value must remain available after applying the shim. As an example, if a key-value store can support prefix queries, it must also be able to provide them when geo-replicated.

**No mechanism interference** Some systems fulfill the previous requirements, but only at the expense of some other features. For example, Cassandra fulfills the availability requirement but, in order to provide range queries, it requires swapping the consistent hashing \cite{47} function for an order-preserving hash function in the mapping of objects into storage nodes, which may cause hotspots during data placement.

**Separation of Concerns** The underlying data store requires no modification and replication should be provided in the form of a shim, uniquely by intercepting and injecting messages in the system. The solution should be general enough to be used by any key-value stores providing the same storage model (or a superset) and the same API (or a subset).

GeoD intercepts requests from clients, and determines what transitions happening at the data store level, propagating them to the other replicas. GeoD’s semantics guarantees that no conflicts will arise from concurrent operations by employing
automatic conflict resolution – the outcome of the operations is designed to be associative, commutative and idempotent. The separation of concerns in the GeoD architecture is represented in Figure 3.1.

In addition to convergent conflict handling, GeoD tracks the causality between operations. If an operation \( op_1 \) happens before another operation \( op_2 \) and they both target the same object or bucket, the outcome of operation \( op_1 \) is discarded.

The following section will detail the design of GeoD and discusses possible variations.

3.1 Causality Tracking

GeoD employs the use of vector clocks to capture the causality relationship between the different versions of an object. We define the causality relationship in the system as the happened-before order: if one replica sees a given (local or remote) update \( u_1 \), by definition all other updates issued locally happen after \( u_1 \). If two updates are not causally related, they are concurrent (as previously described). More formally, the set of updates issued to the system globally is a partially ordered set.

3.2 Conflict Handling and Merging Semantics

Concurrent operations may conflict with one another. As an example, if a client changes the value of an object while another deletes it, the semantic outcome is not well-defined. In the presence of concurrent updates to the same object, we want to be able to enforce strong eventual consistency semantics. In order to achieve this, we must make all the operations associative, commutative and idempotent, while maintaining their meaning.

The choices available between create and drop, as well as between add and delete are simple: one can either opt for a delete-wins approach (simpler to implement; which favors the deletion of information) or an add-wins approach (preferred; avoids

Figure 3.1. Separation of concerns in the GeoD architecture, where a shim layer replicates the underlying data store by intercepting requests from clients.
deleting information if it is concurrently accessed). We chose the latter, as it proves more familiar to reason with, and avoids accidental removal of information.

There is yet another conflicting case – when two users concurrently set the value of a key, what should be its result? Clearly, the ideal value should be a merge of both values, which is dependent on the context on which it is inserted (i.e., what the abstract binary value represents in the context of the application storing it). There are a number of options we can choose from:

**Last Writer Wins** Although we cannot rely on physical clocks to ensure consistency, they can be used as a tiebreak for conflicts. Consider the following composition: [physical clock][replica ID]. This will allow us to provide linearizability between all updates, from all replicas, at each of the replicas. This means that we will always be able to decide that one update is more recent than the other, effectively discarding the oldest. This is the approach taken by most eventually-consistent storage systems, such as Amazon S3.

**CRDT-Merge Function** Any associative, commutative and idempotent function is enough to merge conflicting updates. Common examples are maximum, minimum, summation and multiplication functions. This provides potentially better results than the last-writer-wins, but is very context-dependent. Although we can allow the user to decide how he wants to merge the keys of a bucket, the data-store is also limited to using very simple policies, which may not be the optimal for each application.

<table>
<thead>
<tr>
<th>Operation Pairs</th>
<th>createBucket(bid)</th>
<th>dropBucket(bid)</th>
</tr>
</thead>
<tbody>
<tr>
<td>createBucket(bid)</td>
<td>Bucket <em>bid</em> is created</td>
<td>Bucket <em>bit</em> is created; Drop operation ignored</td>
</tr>
<tr>
<td>dropBucket(bid)</td>
<td>Bucket <em>bit</em> is created; Drop operation ignored</td>
<td>Bucket <em>bit</em> is dropped</td>
</tr>
<tr>
<td>setKey(bid, oid, value)</td>
<td>Bucket <em>bid</em> is created</td>
<td>Bucket <em>bit</em> is created; Drop operation ignored</td>
</tr>
<tr>
<td>deleteKey(bid, oid)</td>
<td>Bucket <em>bid</em> is created</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1. Merge semantics for bucket operations

<table>
<thead>
<tr>
<th>Operation Pairs</th>
<th>setKey(bid, oid, value)</th>
<th>deleteKey(bid, oid)</th>
</tr>
</thead>
<tbody>
<tr>
<td>setKey(bid, oid,</td>
<td>?</td>
<td>Object <em>oid</em>’s value is set</td>
</tr>
<tr>
<td>value)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>deleteKey(bid, oid)</td>
<td>Object <em>oid</em>’s value is set</td>
<td>Object <em>oid</em> is deleted</td>
</tr>
</tbody>
</table>

Table 3.2. Merge semantics for object operations
Delegate to Application  As this provides the best results, at the cost of increased complexity for the programmer. However, applications may not want or be able to make use of this feature – if such capabilities are provided, they should be optional.

Some clients may not want to deal with conflict resolution, specially legacy applications. However, other clients may want to deal with concurrent conflicting operations.

The ideal solution is to provide both: have a merge policy based on last-writer-wins attached to the normal \textit{get} operation, but augment the API with a \textit{getConcurrent} operation, which returns multiple values (the set of most recent concurrent versions of the object). To achieve this, a vector clock is attached to each operation as the primary tiebreak. In addition, every operation is tagged with a physical timestamp, generated by the clock at the local replica. Therefore, physical clocks of all of the replicas should be kept in sync.

In addition to storing multiple versions of a given object (one corresponding to each value written), an index of versions and their order must be kept and maintained. This responsibility is offloaded to the data store.

3.2.1 Delta Mutators

There are two approaches to propagate the changes done to a replica: a state-based, and an operation-based. The state-based approach involves sending the entire state of the replica – this can quickly become a huge overhead, as the state grows too big. On the other hand, one may send the log of local operations, which is not optimal – overwritten states can be discarded safely.

To avoid these limitations, we use the $\delta$-CRDT approach: a state-based approach that encodes only the most recent changes (i.e. the ones that the destination replica is not aware).

In this section we describe the $\delta$-\textit{mutators} [18, Definition 1], one for each update operation in the system. The computation of mutators is tightly related to the data model (bucket-key-value store) and operations provided to manipulate it (bucket creation and deletion, object creation and deletion).

To follow the $\delta$-CRDT specification, each of the operations must be associated with a mutator function, which encodes the internal state changes caused by each instance of the operation. For example, if a user creates bucket $B$, the mutator must encode that bucket $B$ was created, and that it does not create conflicts with any other concurrent operation in any site.

Table 3.3 and Table 3.4 show how each instance of an operation affects the internal state of the data store, along with the preconditions for it to complete successfully.
3.3 RECOVERING DELETED DATA

A problem arising with the previously discussed merge semantics is the need to undelete buckets and their contents in case a concurrent createBucket or object update operation is issued. Therefore, the actual deletion of data must be delayed until we are sure no concurrent updates happened and all replicas acknowledge the delete. Whenever a bucket is deleted, we only mark the bucket and its contents as deleted. All local subsequent requests to the marked-as-deleted bucket result in an error, until recreated.

The actual deletion at a given replica occurs when it receives acknowledgement from all other replicas that they received the deletion, and no concurrent creation or update of the bucket happened. The fact that deleted buckets may reappear after some time won’t be accounted in most user’s mental models, and can be effectively considered strange behaviour. An alternative is to synchronize bucket operations, eliminating their concurrency.

3.4 Dissemination and Garbage Collection

All replicas periodically broadcast their current-epoch updates to all other sites, along with a version vector encoding what sets of updates it has seen. As soon as all replicas asynchronously acknowledge they have seen a given set of updates, those updates can safely be garbage-collected.

As long as there is some connectivity and sufficient bandwidth (i.e. the average bandwidth between replicas exceeds the average amount of information to be sent), replicas are guaranteed to make progress, and information can be garbage-collected before a too-large amount is accumulated.
3.5 Extensions

In this section we discuss possible enhancements that can be added to GeoD, based on existing literature.

GeoD encompasses only the core operations of key-value stores. However, it can be easily extended to richer data models, such as column-family stores (as Eiger \[55\] does), or even incorporating the mechanics of Indigo \[24\] to support arbitrary applications.

Incorporating Indigo would also allow GeoD to support global invariants. For example, in a given context the key-value pairs in a bucket may represent account numbers and their balance. If we want to enforce that all accounts must have positive balance, withdrawals cannot be done without synchronization.

In addition to seeing objects as black-boxes, one could extend GeoD to be a Key-CRDT store \[32\], as in SwiftCloud \[82\]. This would allow for rich conflict-resolution policies at the data-store level, freeing developers from the burden.
Chapter 4

GeoD Implementation

4.1 Overview

In this section we detail the components of GeoD with enough detail to produce a real-world implementation of the system. The GeoD implementation has three major components: an interference component dependent on the data-store and its API, a delta-management module and a propagator. The internal architecture of GeoD can be seen in figure 4.1.

Client requests are intercepted and analyzed at the API Translation component – mainly to check if the user is accessing items that have been marked as deleted. Any delete operations issued by the user are removed from the request, and are instead written into the Delete Markers database. After these two tasks complete, the request is then forwarded to the data store.

Once the data store returns a response, it is intercepted and modified: the result of any delete operations that were not sent to the data store must be added to the response body before forwarding it to the client. At this stage, one must infer the internal changes that occurred in the data-store, using our \( \delta \)-CRDT abstraction – each of the operations is translated into its corresponding set of state changes and sent to the Delta Management module.

The Delta Management is responsible for taking these mutators and merging them into a global delta object, which encodes all the internal state changes that happened in the current epoch. Periodically, the Delta Management will push the delta object and forward it for propagation to the other sites, and start a new epoch (i.e. reset the delta object to a no-changes representation, incrementing the local version vector).

Upon reception of remote updates, they are compared with the latest local versions. If the remote version is older, or is a concurrent delete or drop, it is discarded; otherwise, they are merged in the local data store. Any delete or drop operations that have not been acknowledged by all replicas are instead converted to a delete marker, rather than applied at the data store itself. If a non-delete operation is concurrent with a delete marker, it removes the delete marker.
4.2 GeoD Protocol

In this section we formalize the protocol used by clients to communicate with GeoD. GeoD exposes an API equivalent to that of a Bucket-Key-Value store. Each GeoD request can be seen as a log of operations. The grammar defining GeoD requests to data stores can be seen in [definition 4.2.1] Different key-value stores have different but equivalent APIs – they can be translated into GeoD’s API. Some may allow requests to a single bucket, while others may encode the bucket ID in the object requests, etc.

**Definition 4.2.1.** BNF specification of the GeoD protocol exposed to clients

\[
\langle \text{request} \rangle \quad ::= \quad \langle \text{commands} \rangle
\]

\[
\langle \text{commands} \rangle \quad ::= \quad \langle \text{commands} \rangle \ \langle \text{command} \rangle \quad \mid \quad \langle \text{command} \rangle
\]

\[
\langle \text{command} \rangle \quad ::= \quad \text{create bid} \quad \mid \quad \text{load bid} \quad \mid \quad \text{drop bid} \quad \mid \quad \text{add bid oid value} \quad \mid \quad \text{get bid oid} \quad \mid \quad \text{delete bid oid}
\]

In the examples here presented we will use JSON in place of a syntax tree for clarity. An example of a request to GeoD can be seen in [example 4.2.1]

**Example 4.2.1.** A GeoD request in JSON format. The request adds the key-value pairs \((key1, newval1)\) and \((key2, newval2)\) to an already-existing bucket \(bucket1\).
4.3 API Translation

The API Translation component is responsible for intercepting the requests to the data-store. For every command that changes the data-store’s internal state, it emits a mutator [section 3.2.1] encoding the state transition, which is then merged into the deltas to be broadcasted to the other sites. The mutator’s structure is defined in [definition 4.3.1]. It must also be able to do the inverse – to generate requests to the data-store based on mutators. This is vital to the merging of remote updates.

**Definition 4.3.1.** BNF specification of the list of δ-mutators

\[
\langle mutators \rangle ::= \langle buckets \rangle
\]

\[
\langle buckets \rangle ::= \langle bucket \rangle \langle bucket \rangle \\
| \langle bucket \rangle
\]

\[
\langle bucket \rangle ::= bid \langle exists \rangle \langle objects \rangle
\]

\[
\langle objects \rangle ::= \langle objects \rangle \langle object \rangle \\
| \langle object \rangle
\]

\[
\langle object \rangle ::= oid \langle exists \rangle value
\]

\[
\langle exists \rangle ::= true | false
\]

In addition to the translation routines, it also requires to keep track of deleted items – if a user requests access to one of these, a *not found* error response must be generated and sent back to the user. Conversely, if a user deletes one of these items, a delete marker must be generated for future reference. In our implementation we only need to keep track of deleted buckets. The data structure’s organization can be seen in [definition 4.3.2].

```json
[  
  {  
    "op": "add",
    "bid": "bucket1",
    "oid": "key1",
    "value": "newval1"
  },  
  {  
    "op": "add",
    "bid": "bucket1",
    "oid": "key2",
    "value": "newval2"
  }
]
```
Algorithm 4.1 Handling Requests from Clients

```plaintext
function HANDLECLIENTREQUEST(request)
    deleteMarkers ← GETDELETEMARKERS()
    dropOperations ← ∅
    for all bid ∈ request do ▷ Check accesses to deleted buckets
        if bid ∈ deleteMarkers then
            return ErrorMessage()
    for all operation ∈ request do ▷ Remove drop operations
        if operation.op = drop then
            dropOperations ← operation
    response ← SendToDatastore(request)
    response ← INCLUDEDELETEMARKERS(dropOperations, response)
    mutators ← GENERATEMUTATORS(request, response)
    if response.status = OK then
        deleteMarkers ← deleteMarkers ∪ dropOperations
        MERGELOCALDELTAS(mutators)
    return response
```

Definition 4.3.2. BNF specification of the delete markers structure

```plaintext
⟨delete-markers⟩ ::= ⟨delete-markers⟩ ⟨delete-marker⟩
| ⟨delete-marker⟩
⟨delete-marker⟩ ::= bid ⟨vv⟩
⟨vv⟩ ::= ⟨vv⟩ ⟨vventry⟩
| ⟨vventry⟩
⟨vventry⟩ ::= ⟨site-id⟩ ⟨vvindex⟩
```

### 4.4 Delta Management

At all stages we need to keep a complete summary of all the incremental changes done in the data store. We call this summary a *delta*. The Delta Management component is responsible for merging the mutators into a global *delta* object encoding the union of the incremental state changes. The deltas are a fast-growing set of data with no size bounds that needs to be persisted.

**Definition 4.4.1.** BNF specification of the deltas

```plaintext
⟨deltas⟩ ::= ⟨vvindex⟩ ⟨buckets⟩
⟨buckets⟩ ::= ⟨buckets⟩ ⟨bucket⟩
| ⟨bucket⟩
```
4.5. PROPAGATOR

\[
\langle \text{bucket} \rangle ::= \text{bid} \langle \text{vv} \rangle \langle \text{exists} \rangle \langle \text{objects} \rangle
\]

\[
\langle \text{objects} \rangle ::= \langle \text{objects} \rangle \langle \text{object} \rangle \\
| \langle \text{object} \rangle
\]

\[
\langle \text{object} \rangle ::= \text{oid} \langle \text{versions} \rangle
\]

\[
\langle \text{versions} \rangle ::= \langle \text{versions} \rangle \langle \text{version} \rangle \\
| \langle \text{version} \rangle
\]

\[
\langle \text{version} \rangle ::= \langle \text{vv} \rangle \langle \text{exists} \rangle \text{value}
\]

\[
\langle \text{exists} \rangle ::= \text{boolean}
\]

\[
\langle \text{vv} \rangle ::= \langle \text{vv} \rangle \langle \text{vventry} \rangle \\
| \langle \text{vventry} \rangle
\]

\[
\langle \text{vventry} \rangle ::= \text{site-id} \text{ scalar-clock}
\]

4.5 Propagator

The propagator is the component in charge of sending and receiving messages from all other participating GeoD processes in the replication group. The actual implementation details of this component are not relevant for the application presented here and are therefore omitted.

The propagator should implement a broadcast protocol [30, 34, 60] to periodically propagate its local deltas to the other sites. As GeoD’s only requirement for the inter-site network is that message integrity is preserved, any broadcast algorithm suffices – no need for reliability nor atomicity.

In addition to the broadcast protocol, it needs to keep a consistent view of the members of the system.

This chapter detailed the necessary core components to develop an optimistic replication shim. In the following chapter we discuss a possible deployment scenario for GeoD with heterogeneous storage nodes, where some nodes store data and others only store metadata.
Chapter 5

Emulating Versioning Support

Big Data applications favor the utilization of key-value stores over traditional relational database systems for their improved ability to scale. These usually map each object to a single value, not supporting multiple versions for a given key. However, applications that require complex merge semantics might want to be exposed to object versioning. One of such applications is GeoD. This chapter presents how to emulate versioning support by adding a shim on top of key-value stores that only support a single value per object.

5.1 Problem Definition

We formulate the problem of enabling multiple versions per object on top of an arbitrary strongly-consistent bucket-key-value store providing the core API (tables 2.1 and 2.2) and extending it with a versioning API (table 2.3).

As we are trying to map three parameters (key, value and version) into an API that accepts two (key and value), there are two possible approaches:

5.1.1 Encoding Versions in Value

The simplest solution is to encode all the versions in the value. This changes the mapping of each key to an object, to having each key mapped to a list of versions of that object. The list of versions is a black-box for the key-value store, and is externally managed. However, for objects with many versions this may become impractical due to the large amount of data retrieved in every operation. In practice, all the read operations must retrieve all the versions of the object, and all write operations must read the current list of versions and append a new version to it. This may add significant network utilization, as well as increased latency.

5.1.2 Encoding Versions in the Key

Another solution is to encode the version in the key, by partitioning the actual keyspace into disjoint logical keyspaces. This addresses the previous network utiliza-
tion problems, as one is able to retrieve each version individually, and may append new versions without reading the previous ones. Some data stores, however, may impose a limit on the length of a key.

For example, if the maximum key length is 128 bytes and the user specifies a key with length 120 bytes, only 8 bytes are left to encode the version. If the version cannot be encoded in the remaining 8 bytes, storing or accessing such version results in an error.

In practice, this can be lifted by simply shortening the length of keys that can be specified by the user, thus reserving a fixed (small) amount of bytes to specify the version. Although this imposes a limit on the number of versions, using techniques such as base64 [45] can encode a large amount of versions in little space.

5.2 The VersionD Shim

In this section we introduce VersionD, a versioning shim for key-value stores providing the core API. VersionD encodes the version identifiers in the key part of the object. For design simplicity, we consider assumption 1 in the design of VersionD. The separation of concerns in the VersionD architecture can be seen in figure 5.1.

VersionD handles versioning in a way similar to that of Amazon S3 [1]. The add version replaces the current value by appending a new version to the object. The add version does the same, but the version assigned to the value is arbitrary – it may be more recent, older, or concurrent. The delete operation appends a delete marker to the object – the object is marked and shown as removed, but no data is lost. The delete version removes a specific version from the data store.

Due to versioning and causality, three get operations are provided. The normal get returns the result of merging all the latest values of the object. The get version simply returns a specific version of an object. get latest returns all the latest values of the object – the set of values corresponding to all the latest concurrent versions.

Assumption 1. The underlying key-value store imposes no limit on the length of keys.

Figure 5.1. Separation of concerns in the VersionD architecture, where a shim layer augments the API of the underlying data store by incorporating versions in the mapping of objects.
It is clear that we need to keep some metadata about the versions of an object, in addition to their actual values. In our implementation we only need to keep track of the set of latest versions of an object (i.e. those that are not older than any other version), but other operations may be supported if needed. For example, one may want to keep the full list of versions of an object to support operations such as `listAllVersions(oid)`, providing clients with the listing of versions being stored.

Table 5.1 describes how client operations map into the underlying core operations. In order to partition the bucket space to store different kinds of information without them conflicting, we add a prefix (Algorithm 5.1). This prefix behaves as a selector for the partition we want to access.

Algorithm 5.1 Key Mapping Functions. The symbol || denotes the concatenation of strings. The symbol • is a delimiter used in the key that does not occur in oid, or is escaped. In a practical implementation the actual prefixes used can be optimized to fit in fewer bits.

```plaintext
function M_latest(oid)  # Maps a key to its set of concurrent versions
    return 'LATEST' || • || oid

function M_versions(oid)  # Maps a key to its versions
    return 'VERSIONS' || • || oid

function M_version(oid, version)  # Maps a (key, version) pair to its value
    return M_versions(oid) || • || version
```

The mechanism presented here is sufficient to provide the desired versioning features, and combined with GeoD makes it possible to transform any key-value store into a fully-featured weakly-consistent key-value store, with customizable merge semantics and versioning support.

5.3 Improvements

We present some improvements for VersionD that would be useful in practice, but are not relevant to the core functionality of the system.

5.3.1 Garbage Collection

The implementation presented here does not perform garbage collection of old versions. Most applications don’t make use of obsolete versions and, in these cases, they can be deleted. Garbage collection can be done easily, by slightly modifying the mapping between client operations and data store operations. No further storage is required.

5.3.2 Metadata Caching

Most of the data store routines start by retrieving the required metadata associated with the key in the request. In our case, the metadata is a potentially small list of
### Table 5.1

<table>
<thead>
<tr>
<th>Client Operations</th>
<th>Data Store Operations</th>
</tr>
</thead>
</table>
| add(oid, value)   | version = NewVersion()  
|                   | add(M\_latest(oid), {version})  
|                   | add(M\_versions(oid, version), value)  |
| add\_version(oid, value, version) | latest = latest(get(M\_latest(oid)), version)  
|                   | add(M\_latest(oid), latest)  
|                   | add(M\_versions(oid, version), value)  |
| delete(oid)       | version = NewVersion()  
|                   | add(M\_latest(oid), {version})  
|                   | add(M\_versions(oid, version), deleteMarker)  |
| delete\_version(oid, version) | add(M\_latest(oid), {version})  
|                   | add(M\_versions(oid, version), deleteMarker)  |
| get(oid)          | get(M\_latest(oid)) \(\rightarrow (v_1, v_2, \ldots, v_N)\)  
|                   | get(M\_version(oid, vi)), \(\forall i \in 1, 2, \ldots, N\)  
|                   | \(\text{merge}(v_1, v_2, \ldots, v_N)\)  |
| get\_version(oid, version) | get(M\_version(oid, version))  |
| get\_latest(oid)  | get(M\_latest(oid)) \(\rightarrow (v_1, v_2, \ldots, v_N)\)  
|                   | get(M\_version(oid, vi)), \(\forall i \in 1, 2, \ldots, N\)  |

version IDs (considering a scenario with low average number of versions per object). For example, one million objects with an average of two versions each, with each version being encoded in 10 bytes on average, combined would take less than 20MB of space. Thus, caching would allow us to satisfy the client requests while removing one roundtrip to the data store.

### 5.4 Previous Work

Felber et al. [37] consider the problem of providing versioning to distributed strongly-consistent key-value stores. They concluded that a plain key-value store cannot emulate a versioned key-value store. However, VersionD provides versioning support on top of a non-distributed key-value store, by adding the replication layer above the versioning layer. Therefore, their Separation Result [37] does not hold. We are able to bring versioning support to a replicated data store through separation of concerns and reordering the software layers, effectively bypassing their limitation.
Chapter 6

Shared-Data Architecture

6.1 Architecture Overview

In this chapter we explore a specific use-case in a non-conventional architecture, by deploying GeoD on top of SindexD, Scality’s [11] bucket-key-value store featuring prefix queries. The goal of deploying GeoD is to optimistically replicate the external state of SindexD. SindexD does not persist any of the data itself – it offloads all of it into a set of two DHTs, using them as logically-independent key-value stores. The key details of this architecture are:

Shared Data DHT The DHT where the object data is stored is shared between the SindexD instances in all sites. All sites can write to any location in this DHT. This DHT is not replicated – it is stretched among all sites. All the data placed here is immutable – if one chunk needs to be changed, the value of the new chunk is placed at a different key location in the DHT.

Private Metadata DHT The DHT where all the metadata is stored is private and only accessible by the local SindexD instance. This DHT is replicated at all the sites, i.e. each site maintains an independent copy that is only locally accessible.

Chunked Object Store Large objects are split into several fixed-size chunks, each of them mapped into a different key in the data DHT.

Chunk Indexing The data chunks are indexed via a B-Tree-like structure. The tree nodes are stored at random locations in the metadata DHT.

A pure GeoD solution is not ideal for deployment in the RING, Scality’s software-defined storage system. Due to the non-deterministic placement of data in the shared ring, together with the lack of synchronization between the SindexD replicas, each site will place in the shared RING its own copy of each object. In addition, collisions may occur, as both sites may generate the same hash for different objects.

The ideal case would be for the different SindexD instances eventually map to the same locations in the shared data ring, avoiding writing and storing one copy
6.2 Single Copy Per Chunk

Currently each object is split into several chunks, which are then indexed using a B-Tree-like structure. The nodes of the tree are stored in a private metadata ring, and all the records (the actual data nodes) are stored in the shared data ring. All the nodes of the B-Tree are given random identifiers for load balancing purposes. In addition, each of the records is also mapped into a random location in the ring for the same reason.

6.2.1 Deterministic Mapping

In order to make the mapping deterministic and distributed, without sacrificing the load balancing properties, we propose the utilization of consistent hashing [47]. It has all the desired properties, and allows for convergent mapping of all the replicas into the same locations.
Definition 6.2.1. Deterministic collision-free mapping.

The deterministic mapping of chunks into their location in the shared data ring is given by:

\[ \text{key}_{\text{RING}} \leftarrow H(\text{site}, \text{version}, \text{key}, \text{chunk}) \]

If two chunks have a different number, belong to a different object, to a different version of an object, or are written by a different site, we guarantee them to have different keys (assuming no hash collisions).

By replacing the current mapping function with consistent hashing, each chunk of each object will be mapped into a specific place. In addition, by including the site ID, we guarantee that two different sites writing concurrently to the same chunk of the same object will not overwrite each others’ changes.

6.2.2 Remote Update Signaling

In addition to the mapping, SindexD must now be able to distinguish between local updates and remote updates. We propose the addition of an API call, signaling that a given update is remote.

Definition 6.2.2. Additional SindexD API call to support remote update signaling:

\[ \text{addRemote}(\text{siteID}, \text{versionID}, \text{objectId}, \text{summary}) \]

Where \( \text{summary} \) is a the list of chunks affected, or simply the number of chunks if no copy-on-write is available. No need to transfer the actual data, as every SindexD instance can simply pull it directly from the RING.

6.2.3 Traffic Duplication

The proposed solution for \textit{data multiplication} also addresses the problem of traffic duplication, as only a summary with the IDs of the chunks affected is transmitted between the GeoD instances.

In summary, this architecture minimizes the propagation latency of updates, as well as the amount of traffic required for propagation due to the separation of data and metadata. The shared data repository may be queried on-demand to fetch the required data, instead of data being constantly broadcasted by all replicas.

The following chapter explains how to augment GeoD’s functionality by adding versioning to objects, allowing for complex merge semantics and logging the values of an object.
Chapter 7

Conclusion

This work shows it is possible to extend the feature set of existing key-value stores by providing optimistic replication as a shim layer.

In this thesis we presented GeoD, a shim providing optimistic replication to independent key-value store instances. GeoD works by intercepting requests from clients to the datastore, detecting the resulting internal state transitions and propagating them to the other sites. It’s built-in conflict resolution ensures that concurrent operations don’t result in data loss.

The built-in design for conflict resolution follows two principles: it should be as familiar to the user as possible (i.e. it should behave similarly as if it was a non-replicated system), and should prevent unwanted loss of information – any deletion that is concurrent with a write to the same object will be ignored. Causality is tracked by resorting to version vectors. In case two values are written concurrently, the one with the largest physical timestamp wins (last-write-wins policy).

We also presented VersionD, an upgrade to GeoD which extends the client API with a versioning API, allowing for assigning multiple values to a single object. This allows for applications to use their own conflict resolution semantics, rather than being limited by the built-in mechanism. The API changes are backwards compatible – any application working with GeoD works the same with VersionD. The downside to VersionD is that an API call from a client can translate to several calls to the key-value store, increasing the latency for requests and the load on the system.

Finally, we also describe an efficient architecture for optimistically replicated data stores, using two distinct storage components – a common data repository, shared by all the sites, and a private metadata repository, optimistically replicated at all sites. This allows to reduce massively the size of synchronization messages, thus reducing bandwidth costs and propagation latency and, consequently, the number of conflicting operations.
Bibliography


[63] Ameya Nayak, Anil Poriya, and Dikshay Poojary. Type of NOSQL Databases and its Comparison with Relational Databases.


