Gonçalo Mendes Cabrita

Non-uniform replication for replicated objects

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

Orientador: Nuno Manuel Ribeiro Preguiça,
Professor Associado,
DI, FCT, Universidade NOVA de Lisboa

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

March, 2017

Non-uniform replication for replicated objects

Copyright © Gonçalo Mendes Cabrita, Faculdade de Ciências e Tecnologia, Universidade NOVA de Lisboa.
A Faculdade de Ciências e Tecnologia e a Universidade NOVA de Lisboa têm o direito, perpétuo e sem limites geográficos, de arquivar e publicar esta dissertação através de exemplares impressos reproduzidos em papel ou de forma digital, ou por qualquer outro meio conhecido ou que venha a ser inventado, e de a divulgar através de repositórios científicos e de admitir a sua cópia e distribuição com objetivos educacionais ou de investigação, não comerciais, desde que seja dado crédito ao autor e editor.

Este documento foi gerado utilizando o processador (pdf)LaTeX, com base no template "unlthesis" [1] desenvolvido no Dep. Informática da FCT-NOVA [2].
[1] https://github.com/joaomlourenco/unlthesis [2] http://www.di.fct.unl.pt

To my family, and friends.

Acknowledgments

I would like to thank my advisor, Prof. Dr. Nuno Preguiça, for the opportunity to work with him on this project. His guidance, encouragement, and support made this work possible.

I would also like to thank my colleagues in the department for their encouragement, and support. At last, I would like to thank my family, and friends for their continued support in my academic endeavors.

This work has been partially funded by CMU-Portugal research project GoLocal Ref. CMUP-ERI/TIC/0046/2014, EU LightKone (grant agreement n.732505), and by FCT/MCT project NOVA-LINCS Ref. UID/CEC/04516/2013.

Abstract

A large number of web applications/services are supported by applications running in cloud computing infrastructures. Many of these application store their data in geo-replicated key-value stores, that maintain replicas of the data in several data centers located across the globe. Data management in these settings is challenging, with solutions needing to balance availability and consistency. Solutions that provide high-availability, by allowing operations to execute locally in a single data center, have to cope with a weaker consistency model. In such cases, replicas may be updated concurrently and a mechanism to reconcile divergent replicas is needed. Using the semantics of data types (and operations) helps in providing a solution that meets the requirements of applications, as shown by conflict-free replicated data types.

As information grows it becomes difficult or even impossible to store all information at every replica. A common approach to deal with this problem is to rely on partial replication, where each replica maintains only part of the total system information. As a consequence, each partial replica can only reply to a subset of the possible queries. In this thesis, we introduce the concept of non-uniform replication where each replica stores only part of the information, but where all replicas store enough information to answer every query. We apply this concept to eventual consistency and conflict-free replicated data types and propose a set of useful data type designs where replicas synchronize by exchanging operations.

Furthermore, we implement support for non-uniform replication in AntidoteDB, a geo-distributed key-value store, and evaluate the space efficiency, bandwidth overhead, and scalability of the solution.

Keywords: Non-uniform Replication; Partial Replication; Replicated Data Types; Eventual Consistency; Key-Value Stores;

Resumo

Um grande número de aplicações/serviços web são suportados por aplicações que correm em infra-estruturas na nuvem. Muitas destas aplicações guardam os seus dados em bases de dados chave-valor geo-replicadas, que mantêm réplicas dos dados em vários centros de dados geograficamente distribuídos. A gestão de dados neste contexto é difícil, sendo necessário que as soluções encontrem um equilíbrio entre disponibilidade e consistência. Soluções que forneçam alta disponibilidade, executando operações localmente num único centro de dados, têm de lidar com um modelo de consistência mais fraco. Nestes casos, existe a possibilidade de réplicas serem actualizadas concorrentemente e é necessário um mecanismo para reconciliar réplicas divergentes. A utilização das semânticas de tipos de dados (e operações) ajuda no fornecimento de uma solução que cumpra os requisitos das aplicações, como demonstrado por tipos de dados livres de conflitos (CRDTs).

Com o crescimento da informação armazenada torna-se difícil ou até impossível guardar toda a informação em todas as réplicas. Para lidar com este problema é comum a utilização de técnicas de replicação parcial, onde cada réplica mantém apenas parte da informação total do sistema. Por consequência, cada réplica parcial consegue apenas responder a um subconjunto de operações de leitura. Nesta tese introduzimos o conceito de replicação não uniforme onde cada réplica quarda apenas parte da informação, mas onde todas as réplicas guardam informação suficiente para responder a todas as operações de leitura. Aplicamos este conceito à consistência eventual e a tipos de dados livres de conflitos e propomos um conjunto de tipos de dados onde réplicas sincronizam por propagação de operações.

Adicionalmente, implementámos também suporte para a replicação não uniforme no AntidoteDB, uma base de dados chave-valor geo-distribuída, e avaliamos o espaço ocupado, a quantidade de dados transmitidos, e a escalabilidade da solução.

Palavras-chave: Replicação não uniforme; Replicação parcial; Tipos de dados replicados; Consistência eventual; Sistemas de armazenamento chave-valor;

Contents

List of Algorithms xvii

List of Figures xix

List of Tables xxi

Listings xxiii

1 Introduction 1
- 1.1 Context . . . 1
- 1.2 Motivating the problem . . . 2
- 1.3 The solution . . . 2
- 1.4 Contributions . . . 3
- 1.5 Document Structure . . . 4

2 Related Work 5
- 2.1 CRDTs . . . 5
  - 2.1.1 Delta-based CRDTs . . . 7
  - 2.1.2 Computational CRDTs . . . 8
- 2.2 Key-value stores . . . 11
  - 2.2.1 Data Models . . . 12
  - 2.2.2 Consistency Guarantees . . . 14
  - 2.2.3 Partitioning . . . 16
  - 2.2.4 System examples . . . 16
- 2.3 Computing Frameworks . . . 19
  - 2.3.1 Apache Hadoop . . . 20
  - 2.3.2 Spark . . . 21
  - 2.3.3 Spark Streaming . . . 22
  - 2.3.4 Storm . . . 23
  - 2.3.5 Percolator . . . 24
  - 2.3.6 Titan . . . 25
  - 2.3.7 Lasp . . . 25
- 2.4 Summary . . . 26

3 Non-uniform replication model 27
3.1 System model . . . 27
3.2 System convergence . . . 28
3.3 Non-uniform eventual consistency . . . 29
3.3.1 Eventual Consistency . . . 29
3.3.2 Non-uniform eventual consistency . . . 29
3.4 Protocol for non-uniform eventual consistency . . . 30
3.4.1 Fault-tolerance . . . 32
3.5 Summary . . . 32

4 Operation-based NuCRDTs 35
4.1 Average . . . 35
4.2 Histogram . . . 36
4.3 Top-K . . . 38
4.4 Top-K with removals . . . 39
4.5 Filtered Set . . . 41
4.6 Top Sum . . . 44
4.7 Summary . . . 45

5 Comparing NuCRDTs with CRDTs 47
5.1 Histogram . . . 47
5.2 Top-K . . . 48
5.3 Top-K with removals . . . 49
5.4 Filtered Set . . . 51
5.5 Summary . . . 51

6 Integration of NuCRDTs in AntidoteDB 53
6.1 AntidoteDB architecture . . . 53
6.2 Implementing the data types . . . 54
6.3 Modifications in AntidoteDB to support NuCRDTs . . . 55
6.3.1 Requirement 1: Operation typechecking . . . 56
6.3.2 Requirement 2: Ignoring no-op effect-updates . . . 56
6.3.3 Requirement 3: Operation compaction . . . 56
6.3.4 Requirement 4: Durability of masked operations . . . 58
6.3.5 Requirement 5: Generating new operations from downstream operations . . . 58
6.4 Summary . . . 59

7 Evaluation 61
7.1 Dissemination overhead and replica sizes . . . 61
7.1.1 Top-K . . . 62
7.1.2 Top-K with removals . . . 62

7.2 Scalability 63
7.2.1 Top-K 64
7.2.2 Top-K with removals 65
7.3 Summary 66

8 Conclusion **67**
8.1 Publications 67
8.2 Future Work 68

Bibliography **69**

A Appendix 1: Example NuCRDT implementation **75**

List of Algorithms

1 State-based G-Counter 6
2 Operation-based G-Counter 7
3 Delta-based G-Counter 8
4 Computational CRDT that computes the average of values added 9
5 Computational CRDT that computes the Top-K player scores 10
6 Computational CRDT that computes the Top-K player scores with support for removals 11
7 Replication algorithm for non-uniform eventual consistency 30
8 Design: Average 36
9 Design: Histogram 37
10 Design: Top-K 38
11 Design: Top-K with removals 40
12 Design: F-Set 43
13 Design: Top Sum 45
14 Algorithm for compacting collections of transactions 57

List of Figures

2.1 Example of the relational data model 13
2.2 Example of the key-value data model 13
2.3 Example of the hybrid data model 14

5.1 Histogram: total message size and mean replica size 48
5.2 Top-K: total message size and mean replica size 48
5.3 Top-K with removals: total message size and mean replica size with a workload of 95% adds and 5% removes 49
5.4 Top-K with removals: total message size and mean replica size with a workload of 99% adds and 1% removes 50
5.5 Top-K with removals: total message size and mean replica size with a workload of 99.95% adds and 0.05% removes 50
5.6 F-Set: total message size and mean replica size 51

6.1 AntidoteDB Architecture 54

7.1 Top-K: total message size and mean replica size 62
7.2 Top-K with removals: total message size and mean replica size with a workload of 95% adds and 5% removes 63
7.3 Top-K experiments 65
7.4 Top-K with removals experiments with a workload of 95% adds and 5% removes 66

List of Tables

7.1 Mean round-trip time between Amazon Web Services EC2 instances 64

Listings

2.1 WordCount in Apache Hadoop (using Scala) 20
2.2 WordCount in Spark (using Scala) 22
2.3 WordCount in Storm (using Scala) 23
2.4 WordCount in Lasp (using Erlang) 26
A.1 Average NuCRDT implementation in Erlang 75

Chapter 1

Introduction

1.1 Context

Over the past decade, the widespread availability of high-speed Internet access has led to a large increase of user activity in Internet services. Several of these services have become incredibly commonplace and have seen widespread adoption by millions of people across the globe. Examples of such services include social networks, document hosting services, and on-line stores.

To cope with this increasing demand, developers are forced to find ways to improve the scalability of their services in order to keep up with the enormous rate of requests while still maintaining a low response time. A lot of these services' data is stored in globally distributed and geo-replicated key-value stores [1, 2, 3, 4]. These data stores maintain replicas of their data in several data centers located across the globe in an effort to not only improve their availability, but also to reduce latency to users in different continents.

Data management in these settings is extremely challenging, with solutions needing to find a good balance between availability and data consistency. Data stores that provide high-availability by allowing operations to execute locally in a single data center sacrifice a linearizable consistency model, typically achieved by using consensus algorithms to maintain a single global view and provide a lockstep transition of the system. Instead, these systems receive updates locally and propagate the operations to other replicas in an asynchronous manner. Although this allows these systems to reduce their latency, they must now cope with a weaker consistency model where replicas can be updated in a concurrent fashion and a mechanism to reconcile diverging replicas is required.

Recently, a substantial amount of recent research has resulted in the proposal of different approaches to the problem of data convergence. One of the proposed solutions [5] has

explored conflict-free replicated data types (CRDTs) that allow replicas to be modified concurrently. These data types have predefined policies to deal with the convergence of data, and these policies help application programmers by providing strong operational semantics and relieving them from the burden of deciding how to merge data from diverging replicas themselves.

1.2 Motivating the problem

With the increase of information maintained by data stores it is often impossible or undesirable to keep all data in all replicas. Besides sharding data among multiple machines in each data center, it is often interesting to keep only part of the data in each data center. In systems that adopt a partial replication model [6, 7, 8], as each replica only maintains part of the data, it can only process a subset of the database queries. Sometimes, it is even necessary to obtain data from multiple replicas for replying to a query.

Navalho *et al.* [9] have proposed several design extensions to CRDTs where the state of the object is the result of a computation – e.g. the average, the top-K elements; over the executed updates. An interesting aspect of this work is that one of the proposed designs departs from traditional CRDTs and replication models in that the state of the replicas does not need to be equivalent to correctly process a read query. For example, the replicas of a CRDT that implements the top-K set do not need to be equal, only the top K elements need to be the same for the resulting read query to be equal at all replicas.

This opens the opportunity to a number of optimizations in the replication process, in which there is no need to propagate every update to every replica, further reducing the dissemination cost and replica sizes of CRDTs. However, it also poses a number of challenges, as with this approach for two replicas to synchronize it might be necessary to establish more than two one-way synchronization steps, which is not the case with standard CRDTs used in one-way convergent data stores [10].

1.3 The solution

We begin by exploring an alternative replication model, which captures the essence of the non-traditional CRDT design proposed by Navalho *et al.* where each replica maintains only part of the data but can process all queries. The key insight is that for some data objects, not all data is necessary for providing the result of read operations.

We apply this alternative partial replication model to CRDTs, formalizing the model for an operation-based synchronization approach in which CRDTs synchronize by exchanging operations. In this context, when an operation is executed at some replica it may not be necessary to propagate it to other replicas if it produces no effect (or no immediate effect). We apply the proposed model to eventual consistency and establish sufficient conditions for having a system that provides non-uniform eventual consistency. We present an algorithm that satisfies the identified conditions.

For the new model to be useful, it is necessary to provide data types that can address application requirements. To this end, we propose a set of non-uniform CRDT (NuCRDT) designs including designs that maintain an aggregated average, a histogram, a filtered set, and several designs for sets that maintain a top-K. We evaluate these designs by simulation, showing that NuCRDTs entail much lower space overhead for storage and bandwidth usage for synchronization when compared with state-of-the-art alternatives, including delta-based CRDTs and computational CRDTs.

To evaluate the impact of non-uniform replication in the performance of an existing system, we have integrated support for non-uniform replication in AntidoteDB, a geo-replicated key-value store, where values are CRDTs.

To achieve this, we first modified the synchronization process to control when updates are propagated to other replicas. On one hand, it is desirable that an update is not propagated when it produces no effects – e.g. adding an element that does not fit in the top-K. On the other hand, it may be necessary to propagate updates as the result of receiving an update – e.g. when a remove operation makes it necessary to propagate an element that now belongs to the top-K and has not previously been propagated.

The second modification raises the question of the durability of operations which are not immediately propagated. While not propagating an operation is important for efficiency (not only less information is exchanged, but also the size of each replica is smaller), it poses problems to fault-tolerance as it is important to keep a minimum number of replicas to guarantee that an operation is not lost when a node fails.

We then evaluate the performance of non-uniform replication in AntidoteDB by measuring the scalability of some of our designs and comparing them to the operation-based CRDTs already supported by AntidoteDB.

1.4 Contributions

The main contributions of this thesis are:

- The non-uniform replication model, the application of this model to eventually consistent systems, the sufficient conditions for providing non-uniform eventual consistency, and an algorithm that satisfies these conditions;

- Multiple designs of operation-based NuCRDTs, and their simulated comparison to state-of-the-art uniform CRDTs in terms of dissemination cost and replica size;

- The integration of support for non-uniform replication in the AntidoteDB key-value store. This includes the design and implementation of such support. We additionally evaluated the performance of AntidoteDB NuCRDTs when compared with traditional CRDTs.

1.5 Document Structure

The rest of this document is organized as follows:

- In chapter 2, we present the related work. Exploring CRDTs, key-value stores, and distributed computing frameworks;
- In chapter 3, we describe the new replication model, formalizing its semantics for an eventually consistent system;
- In chapter 4, we present our designs of operation-based NuCRDTs using the proposed model;
- In chapter 5, we compare our designs to state-of-the-art CRDTs using simulation;
- In chapter 6, we discuss how our designs fit into AntidoteDB, and what changes are required to support them;
- In chapter 7, we evaluate the implementations of our designs in AntidoteDB against the operation-based CRDTs supported by AntidoteDB;
- In chapter 8, we present our conclusions.

Chapter 2

Related Work

This chapter overviews the state of the art in the areas related with the work done in the context of this thesis. This chapter is organized as follows:

- In section 2.1, we present an overall study on Conflict-free Replicated Data Types;
- In section 2.2, several key-value store systems are presented;
- In section 2.3, we explore several distributed computing frameworks and their intrinsics.

2.1 CRDTs

Conflict-free Replicated Data Types [5] are data types designed to be replicated. Each replica can be modified without requiring coordination with other replicas. CRDTs encode merge policies that are used to guarantee that all replicas converge to the same value after all updates are propagated to all replicas. CRDTs allow operations to be executed immediately on any replica, avoiding classic problems that arise due to network latency or faults.

By satisfying three key mathematical properties - idempotency, commutativity, and associativity; CRDTs are guaranteed (by design) to always converge to a single common state.

With these guarantees, CRDTs provide eventual consistency with well defined semantics, making it easier for programmers to reason about them. Two main flavors of CRDTs have been defined, (i) state-based and (ii) operation-based.

Convergent Replicated Data Types State-based CRDTs (or CvRDTs) synchronize by exchanging the state of the replica. To guarantee that replicas converge, the internal

state must form a monotonic semilattice, every operation must change the state moving it upward in the lattice, and the merge operation computes the least upper bound of the state in each replica. This form of CRDT proves to be inefficient when handling large objects since exchanging the entire state over the network when small changes are introduced adds considerable overhead. This variant allows for fair-lossy channels to be used, requiring only that the synchronization graph is connected.

Commutative Replicated Data Types Operation-based CRDTs (or CmRDTs) synchronize by propagating all executed updates to all replicas. To guarantee that replicas converge, concurrent operations must satisfy the commutativity property. This variant assumes that a reliable causally-ordered broadcast communication protocol is used.

Instead of a merge operation like in CvRDTs, CmRDTs instead extend their update operations by splitting them into two functions:

(1) prepare-update (also known as atSource);

(2) effect-update (also known as downstream).

The prepare-update function is responsible for generating an internal representation of the update, while the effect-update function is responsible for locally executing the update and asynchronously propagating it to other replicas.

A simple example of a CRDT is the G-Counter. The G-Counter is a grow-only counter which only provides the increment operation. The state-based version of a G-Counter, presented in algorithm 1, is inspired by vector clocks. As such, each G-Counter holds a vector of integers where each replica is assigned an entry in the vector. This is represented by a mapping between replica identifiers and their local counter. Accessing an inexistent mapping returns the default value, 0. To increment the global counter a replica simply increments its local counter. The global value of the counter is the sum of all the replica counters. To merge two replicas, it is sufficient to take the maximum for each replica entry.

Algorithm 1 State-based G-Counter

```
1: payload [𝕀 ↦ ℕ] P                          ▷ One entry per replica
2:     initial []
3: update INCREMENT():
4:     let g = MYID()                          ▷ g: source replica
5:     P[g] ← P[g] + 1
6: query VALUE():
7:     ∑_{i∈𝕀} P[i]
8: function MERGE(x, y):
9:     [i ↦ MAX(x.P[i], y.P[i]) | ∀i ∈ 𝕀]
```

The operation-based version of a G-Counter, presented in algorithm 2, is much simpler, as it is sufficient to keep the value of the counter if we assume exactly once delivery of updates. Since in this model updates are processed locally and later executed and propagated to other replicas, the update function is split into two phases: (i) atSource, and (ii) downstream. However, in this particular case atSource simply emits its input to downstream.

Algorithm 2 Operation-based G-Counter

```
1: payload ℕ i
2:     initial 0
3: query VALUE():
4:     i
5: update INCREMENT():
6:     ATSOURCE():                    ▷ Empty due to no required processing
7:     DOWNSTREAM():                  ▷ No precond: delivery order is empty
8:         i ← i + 1
```

2.1.1 Delta-based CRDTs

Delta-based CRDTs [11] are a CRDT variant introduced to combat the main weakness of state-based CRDTs, their dissemination costs. Essentially, they are a midway between state and operation-based solutions. Delta-based CRDTs ship a delta of the state instead of its entirety, this reduces the cost of sending updates over the network. However, since the mutator returns a delta-state and not an operation (like in operation-based CRDTs) it does not require exactly-once delivery semantics.

In delta-based CRDTs, updates are replaced with delta-mutators. Unlike update operations that modify and then return the full state, delta-mutators return a delta-state. A delta-state is a value in the same join-semilattice which represents the update induced by the mutator that generated it. Delta-mutations can be joined into delta-groups. A delta-group is simply the delta-state resulting of the combination of the delta-group's mutators.

In algorithm 3 we present the delta-based G-Counter. The state is simply a mapping between replica identifiers and their local counter. Accessing an inexistent mapping returns the default value, 0. The increment delta-mutator returns a mapping from the local replica to its incremented counter. The query operation returns the sum of all replica counters. The merge operation joins two mutators my taking the maximum for each replica entry.

In this case, the G-Counter only ships the counter fields that were mutated (the delta) instead of all its fields.

Algorithm 3 Delta-based G-Counter

```
1: payload [𝕀 ↦ ℕ] P                                  ▷ One entry per replica
2:     initial []
3: mutator INCREMENT^δ():
4:     g ← MYID()                                     ▷ g: source replica
5:     [g ↦ P[g] + 1]
6: query VALUE():
7:     ∑_{i∈𝕀} m[i]
8: function MERGE(m, m'):
9:     [i ↦ MAX(m[i], m'[i]) | ∀i ∈ 𝕀]
```

2.1.2 Computational CRDTs

Computational CRDTs [9] are an extension to state-based CRDTs where the state of the object is the result of a computation - e.g. the average, the top-K elements; over the executed updates.

Three different generic computational CRDT designs have been proposed, that can be used for functions that satisfy the following mathematical properties: (i) incremental, (ii) incremental and idempotent, and (iii) partially incremental.

Incremental The first design considers only incremental computations, where computing the function over two disjoint sets of events and combining the results is equal to computing the function over the union of the two sets. Formally, a computation is incremental if there is a function f, such that for any two disjoint sets of events E_1 and E_2, we have:

$$F^f(E_1 \cup E_2, hb_{E_1 \cup E_2}) = f(F^f(E_1, hb_{E_1}), F^f(E_2, hb_{E_2}))$$

Where hb_E represents a partial causality order on E.

In this design, each replica must:

(1) compute its contributions separately from other replicas;

(2) maintain a map of the contributions of each other replica;

(3) update its contributions when receiving an update operation (by combining previously computed contributions with the contribution of the new operation);

(4) keep the most recently computed result for the partial result of each merged replica[1].

The value of a replica can be computed by applying an aggregation function to the contributions of all replicas.

[1] If the resulting values are monotonic then this information can be automatically inferred, otherwise it must be maintained explicitly. One solution is to use a monotonic version number for each computed result.

An example of this design is a CRDT that computes the average of values added, presented in algorithm 4. In this example, the state of the CRDT is a tuple containing the total sum of values added and the number of values added. Each replica explicitly maintains a map of the contributions of each other replica. As such, we store each replica's state tuple in a vector. Updates occurring on a replica only update this replica's entry in the vector.

The CRDT provides an update operation that adds a value by updating the total sum of values, and increments the number of added values by one. The query function adds the values and number of added values of all entries, with the average being computed as the sum of values over the sum of adds. To merge two replicas the CRDT uses the state tuple with the highest number of values added, for each of the replicas.

Algorithm 4 Computational CRDT that computes the average of values added

```
1:  payload [⟨v, t⟩] P                                   ▷ One entry per replica
2:      initial [⟨0,0⟩, ⟨0,0⟩, . . . , ⟨0,0⟩]
3:  update ADD(v):
4:      g ← MYID()                                       ▷ g: source replica
5:      P[g] ← ⟨P[g].v + v, P[g].t + 1⟩
6:  query VALUE():
7:      ∑_i P[i].v / ∑_i P[i].t
8:  function MERGE(x, y):
9:      [MAX(x_1, y_1) | x_1 ∈ x ∧ y_1 ∈ y]
10: function MAX(⟨v_1, t_1⟩, ⟨v_2, t_2⟩):
11:     if t_1 > t_2 then ⟨v_1, t_1⟩
12:     else ⟨v_2, t_2⟩
```

Incremental and Idempotent This design considers computations that are incremental as well as idempotent. For a computation to be both incremental and idempotent the function must respect the incremental property previously defined while allowing the sets of events to potentially overlap.

This is similar to the incremental design, except that since the computation is idempotent it is possible to maintain in each replica only the computed result. When an update is received, or on a merge, the new value can be computed by executing the idempotent function.

An example an implementation of such a computational CRDT is a data type that computes the top-K player high scores for a leaderboard, presented in algorithm 5. In this leaderboard, only the highest score of a player is displayed.

The state of this CRDT is a set containing tuples with a player identifier and a score. The update function, *add*, runs the merge operation on the replica against a set containing only the new tuple. The query function simply returns the current state. The merge operation finds the highest score tuple for each player and filters the collection of those tuples using a max function so that the resulting collection only contains the top-K

scores. We define our total tuple ordering as follows, $(V_1 > V_2) \vee (V_1 = V_2 \wedge N_1 > N_2) \implies \langle N_1, V_1 \rangle > \langle N_2, V_2 \rangle$.

Algorithm 5 Computational CRDT that computes the Top-K player scores

```
1: payload {⟨n, v⟩} S                          ▷ set of ⟨n, v⟩ pairs; n ∈ ℕ; v ∈ ℕ
2:     initial {}
3: update ADD(n, v):
4:     S ← MERGE(S, {⟨n, v⟩})
5: query VALUE():
6:     S
7: function MERGE(x, y):
8:     MAXK({⟨n, v⟩ ∈ (x ∪ y) : ∄⟨n, v1⟩ ∈ (x ∪ y) : v1 > v})
```

Partially incremental This design considers computations where only a subset of the update operations respect the previously defined incremental property.

An example of a partially incremental computational CRDT is a top-K object where elements can be removed. In this case, elements that do not belong in the top-K may later become part of it, if a top-K element is removed. To address this case we must use workarounds to reason about the elements that are still in the top-K so that we may execute our computations correctly. There exist two proposed workarounds (i) maintaining a Set CRDT that contains the elements that have not been deleted, and (ii) having each replica maintain all operations locally executed and only propagate to the other replicas the operations that might affect the computed result.

In the first case, all replicas must maintain the complete set and all the updates need to be propagated to all replicas. In the second case, each replica must maintain a set of operations and the results of the computations (subsets of operations) performed at other sites. Essentially, the second approach has a lower dissemination cost but its replicas will be larger.

This is conceptually similar to the previous example, but in this top-K player high score computational CRDT we are be able to remove elements. This requires maintaining extra information as when removing an element in the top this requires some other element to be moved to the top. Algorithm 6 presents a design that address this problem.

The state is a tuple containing the set of all operations executed locally, and a vector containing a tuple with the subset of operations that results in the top-K computation and a monotonic integer for all replicas. The monotonic integer is used to compare replicated results to determine which one is the most recent.

An update operation simply updates the local set of operations in the state tuple to include the add or remove operation executed. The diff operation aggregates the operations that cause a change in the results, re-computes its local top-K, and returns a tuple with that information. The query operation aggregates the operations that can affect the state of the computation - e.g. if $\text{ADD}(id, score) \prec \text{DEL}(id)$ (where $\prec$ denotes a causal order between two operations) then $\text{ADD}(id, score)$ will not affect the computation; and

filters that collection of operations using a max function such that the resulting collection only contains the top-K score additions. We use the same total tuple ordering as in the previous example.

The merge operation joins the given set of operations with the existing local one, and updates the subset of operations that results in the computation for each replica – using the monotonic integer previously mentioned to determine the most recent result.

Algorithm 6 Computational CRDT that computes the Top-K player scores with support for removals

```
1:  payload ({op(args)} operations,                          ▷ S: set of add(n, v); n ∈ ℕ; v ∈ ℕ
2:              [⟨{add(n, v)} S, T⟩] results                 ▷ T: monotonic integer
3:      initial ({}, [⟨{}, 0⟩, ⟨{}, 0⟩, ..., ⟨{}, 0⟩])
4:  update ADD(n, v):
5:      operations.ADD(add(n, v))
6:  update DEL(n):
7:      operations.ADD(del(n))
8:  diff ():
9:      g ← MYID()                                           ▷ g: source replica
10:     ops ← {o ∈ operations : CHANGESRESULT(o)}
11:     res ← results[g ↦ ⟨CAUSALMAXK(operations ∪∀i results[i].S), results[g].T + 1⟩]
12:     ⟨ops, res⟩
13: query VALUE():
14:     CAUSALMAXK(operations ∪∀i results[i].S)
15: function MERGE(x, y):
16:     ops ← x.operations ∪ y.operations
17:     res ← x.results[i ↦ MOSTRECENT(x.results[i], y.results[i])] ∀i
18:     (ops, res)
19: function CHANGESRESULT(o):
20:     CAUSALMAXK(o ∪∀i results[i].S) ≠ CAUSALMAXK(∪∀i results[i].S)
21: function CAUSALMAXK(ops):
22:     MAXK({o ∈ ops : o = add(n, v) ∧ (∄o' ∈ ops : o ≺ o' ∧ o' = del(n))})
23: function MOSTRECENT(x, y):
24:     if x.T > y.T then x
25:     else y
```

2.2 Key-value stores

Key-value stores typically provide application developers with a distributed, highly available, and high performance database when compared to the classical relational databases. To provide better availability these systems sacrifice linearizability and leverage weaker forms of consistency such as causal and eventual consistency.

By relaxing their consistency constrains these systems bring their own set of trade-offs dependent on the specific consistency model they follow. As such, it is the application developer's responsibility to pick a database that can correctly model their application data while maintaining the specific invariants the application requires.

In the following sections we briefly present the different data models, consistency guarantees, and partitioning schemes that key-value stores provide. Following that, we discuss several key-value stores and describe their properties and guarantees.

2.2.1 Data Models

Every database has a data model that it uses to internally represent its data in a structured way. There are several models that a database may choose, each with its own set of trade-offs that can affect querying speed, scalability, and restrict query semantics.

2.2.1.1 Relational Model

As the name indicates, the relational data model uses relations between entities to represent its data. An example of this model is shown in figure 2.1. Each collection of entities is stored as a table, each row in this table represents an entity and each column represents an attribute of the entity.

For each table, one or more attributes can constitute a unique key which can be used to efficiently perform queries by indexing the table (and that unequivocally identify the item in the table). This model also provides support for secondary indexes – indexes that are built over column values; which are important to improve the performance of queries over for example the name of a track.

This is a particularly simple and powerful data model. Since entities may be related to other entities and because this relationship can be expressed internally, the model allows for very complex queries to be performed over groups of entities. It also has the advantage of being well known and understood by programmers.

The data model also provides support for transactions which allow users to make updates to different tables atomically.

2.2.1.2 Key-value Model

This is the typical data model used in key-value stores, which simply maps keys to values. An example of this model is presented in figure 2.2. In this model information storage and retrieval is very efficient since it can be directly represented as a dictionary, which allows the use of a hash function for fast indexing of elements.

A distributed hash table [12] is usually used to store data partitioned in a set of values. This model does not support powerful and complex queries like the relational model and typically does have support for any kind of transactions.

2.2.1.3 Key-value and Column hybrid Model (Column Family)

This model uses tables but not with the same semantics as the relational model. Instead, tables are viewed as distributed multi dimensional maps indexed by a key. Every operation under a single row key is guaranteed to be atomic per replica regardless of how

Table of Artists:

id	name
1	Massive Attack
2	Tycho

Table of Albums:

id	name	artist_id
12	Blue Lines	1
15	Awake	2

Table of Tracks:

id	name	album_id
30	Unfinished Sympathy	12
40	Awake	15

Figure 2.1: Example of the relational data model

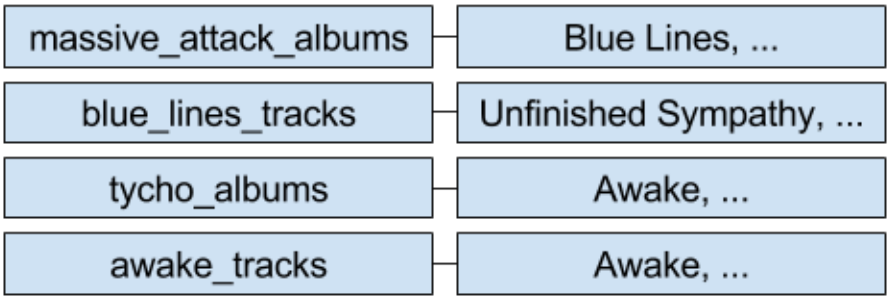

Figure 2.2: Example of the key-value data model

many columns are being read or written into. The model also allows for columns to be grouped together into what are called Column Families. One of the most popular data stores today, Cassandra [3], uses this data model. An example is shown in figure 2.3.

Similarly to the key-value model, the hybrid model has weaker support for complex and powerful queries when compared to the relational model. However, the recent introduction of CQL (Cassandra Query Language) in Cassandra shows that a SQL-like query system can be used in models like this to great extent.

This model also provides support for indexes built over column values, which helps improves the performance of queries where several columns may share the same values, e.g., the name of an artist in a table containing all songs.

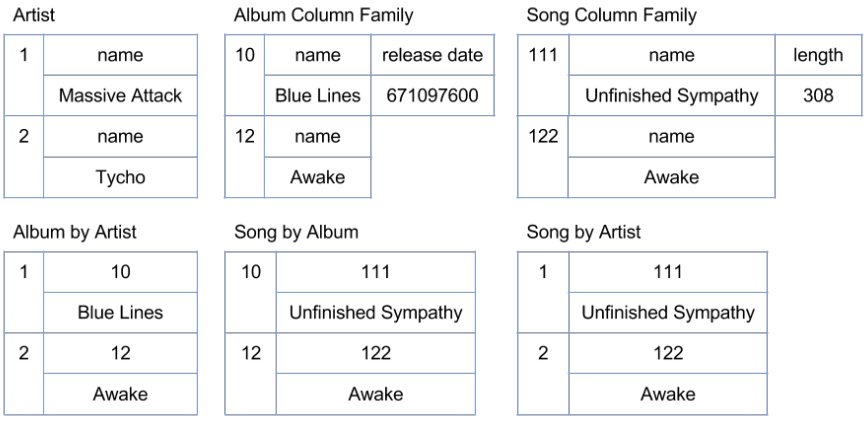

Figure 2.3: Example of the hybrid data model

2.2.2 Consistency Guarantees

Data stores offer varying degrees of consistency, each with its own set of trade-offs between performance and consistency. Typically if we have stronger consistency guarantees then we are sacrificing performance to achieve that level of consistency. However, some applications benefit from this sacrifice due to requiring a stronger form of consistency.

In the following sub-subsections we discuss the consistency models that are later referenced in this section.

2.2.2.1 Eventual Consistency

Eventual Consistency guarantees that (i) eventually all updates are delivered to all replicas and (ii) replicas that received the same updates **eventually reach** the same state.

In this model, update operations may be executed on any replica without requiring immediate synchronization. Therefore, the state of the replicas may diverge temporarily but these replicas will eventually converge to an equivalent state.

2.2.2.2 Read Your Writes

The Read Your Writes consistency model guarantees that if a write is executed over some record, then any attempt to read the value of such record will return said write (or later values).

2.2.2.3 Strong Eventual Consistency

Strong Eventual Consistency guarantees us that (i) eventually all updates are delivered to all replicas and (ii) replicas that received the same updates **have** the same state.

The key difference between this model and the Eventually Consistent model is that it provides stronger semantics for resolving merge conflicts.

2.2.2.4 Causal Consistency

Causal Consistency guarantees that while values may diverge among replicas, the values observed by a client respect a causal history of state-changing operations.

Essentially if a client updates the state of a replica with the following causal dependencies: $s1 \prec s2 \prec s3$, then all other replicas will receive updates in an order that respects this causality constraint. This guarantees that if we can observe $s3$ then the replica has also received both $s1$ and $s2$.

2.2.2.5 Causal+ Consistency

Causal+ Consistency [13] has the same guarantees as Causal Consistency but it also enforces a liveness property on the system in which eventually every replica converges to the same state. This liveness property is the same of Eventual Consistency.

2.2.2.6 Transactional Causal Consistency

Transactional Causal Consistency [14] extends Causal+ Consistency by adding interactive transactions. Transactions in this model read from a causally consistent snapshot and may also update multiple objects at the same time - while respecting atomicity constraints. If two transactions concurrently update the same object, the updates are merged using some automatic reconciliation mechanism (e.g. CRDTs).

2.2.2.7 Snapshot Isolation

In transaction semantics Snapshot Isolation guarantees that all reads performed inside a transaction execute in a database snapshot taken when the transaction starts. For the transaction to succeed the updated values must have not changed since the transaction began, or the transaction will be rolled back.

2.2.2.8 Strong Consistency

In strong consistency models, the system provides the illusion that a single data replica exists. Two main models have been proposed. In serializability, a transaction executes atomically at a given moment, and all other transactions either completely execute before or after. In linearizability, the result of operations must be compatible with operations executed at the same time.

Since there is only one way for the system to move forward, it is very simple for the application developer to reason about the system's state. However, to operate in this way the system needs to resort to heavy forms of synchronization typically through the use of consensus algorithms such as Paxos [15] or Raft [16]. This heavy use of synchronization reduces the system's performance and its ability to tolerate faults while remaining available.

2.2.3 Partitioning

Typically, databases partition their data among several nodes to achieve better scalability and load-balancing. However, it is also crucial to replicate data across multiple nodes for fault tolerance.

A good way of providing this is to make use of Consistent Hashing [17] techniques. In Consistent Hashing a hash function is used to map a data object to a node in the system which will store the object. Usually it will form a hash ring by overlapping the highest and lowest hash value. Each node in the system is assigned a random value in this range to be used as an identifier. The hash function should be well mixed so that data is equally distributed among all nodes. Searching for an entry in the system is very efficient, since simply applying the hash function will give us the object's location. However, Consistent Hashing incurs some maintenance costs because the system has to manage nodes joining and leaving the hash ring, which requires updating the collection of active nodes.

If object replication is used, an object will typically be replicated in the nth consecutive nodes where n is the degree of replication.

2.2.4 System examples

We now present several systems that fall under the key-value store umbrella.

2.2.4.1 Akka Distributed Data

Akka Distributed Data [18] is a data store used to share data between actor nodes in an Akka Cluster [19]. Distributed Data uses state-based CRDTs as values, providing strong eventual consistency in a distributed key-value store.

In Distributed Data reads and writes support several degrees of consistency (local replica, n replicas, a majority of replicas, and all replicas), allowing the programmer to choose how consistent the operations should be. Deleted keys cannot be reused. Any actor may subscribe to updates over any CRDT in the store.

Certain types of CRDTs accumulate some kind of data that can stop being relevant. For example, the G-Counter CRDT maintains a counter per each replica. In long running systems this can be an issue since there are always nodes leaving and joining the system. Due to this, Distributed Data periodically garbage collects CRDTs to prune data associated with nodes that have been previously removed from the cluster.

To disseminate data, Distributed Data runs a specific actor called *Replicator* on each node in the Cluster which is responsible for communicating with all other *Replicator* actors in the Cluster through the use of a variation of a push-pull gossip protocol.

2.2.4.2 Cassandra

Cassandra [3] is a distributed key-value store. It uses the hybrid data model as previously presented. This hybrid data model allows the database to provide a richer data schema

than typical key-value stores since Cassandra's table columns can hold columns as well.

It employs eventual consistency through the use of a Last Writer Wins strategy to resolve conflicts that arise from conflicting writes. In this strategy the value that is kept is the one with the latest cell timestamp.

Cassandra allows for nodes to be distributed among multiple data centers. For data distribution across nodes Cassandra uses consistent hashing, and for data placement it provides two distinct strategies (i) SimpleStrategy, and (ii) NetworkTopologyStrategy.

SimpleStrategy This strategy places the first replica on a node determined by the partitioner. Additional replicas are then placed on the successor nodes without considering the topology (rack or data center).

NetworkTopologyStrategy This strategy allows the configuration of how many replicas we want per data center, and attempts to place replicas within the same data center on distinct racks since nodes in the same rack may fail at the same time.

2.2.4.3 AntidoteDB

AntidoteDB [20] is SyncFree's reference platform and a distributed, geo-replicated and highly-available key-value store. It implements the Cure [14] protocol, which was developed with the goal of providing the highest level of consistency while being highly available.

The Cure protocol allows AntidoteDB to provide highly available transactions (HAT) with the Transactional Causal Consistency (TCC) model. This model allows for transactions to read from a snapshot while also guaranteeing the atomicity of updates.

To encode causal dependencies the system relies on the timestamps of events, and allows each node to use their own timestamps to avoid centralized components. The protocol implemented by Cure assumes that each DC is equipped with a physical clock and that clocks are loosely synchronized by a protocol such as Network Time Protocol.

The snapshot accessed by a transaction is identified by a vector of realtime timestamps, with one entry per Data center.

The database provides common operation-based CRDTs such as counters, sets, maps, and sequences.

2.2.4.4 Dynamo

Dynamo [4] is an eventually consistent key-value store developed by Amazon. Dynamo has a simplistic key-value data model, where each value is a blob meaning that there is no data schema that the database knows of.

In order to detect conflicting writes to the same key, Dynamo uses Vector Clocks [21]. On reads, if the system has detected conflicting objects it can send all the conflicts to

the client and since the client is typically aware of the data semantics it can make an informed decision on how to merge the different object versions.

To distribute data across nodes, Dynamo applies consistent hashing techniques. Dynamo can be configured to replicate objects as needed, and the object will be replicated to N neighboring nodes.

Dynamo also allows for tunable consistency by configuring the number of nodes that should participate in a read (R) and in a write (W) operation. For example, setting R and W so that $R + W > N$ leaves us with a quorum-like system and Read Your Writes (RYW) consistency.

2.2.4.5 DynamoDB

DynamoDB [1] is Amazon's commercial NoSQL database as a service. DynamoDB is built on the same principles of Dynamo, and it has a similar data model.

Being a fully managed service, DynamoDB allows the user to offload administrative concerns of operation and scaling to Amazon. The system works as a key-value store and allows the user to scale up or down each tables' throughput capacity without suffering downtime.

While being an eventually consistent database, DynamoDB provides different types of consistency per read operation, allowing the users to specify if they want an eventually consistent or strongly consistent read.

DynamoDB also provides atomic updates but only in a few selected cases. Namely, the system provides the user with the capability of incrementing or decrementing a numeric attribute in a row atomically, and atomically adding or removing to sets, lists, or maps.

To help the user coordinate concurrent updates to the same data, DynamoDB provides a conditional write operation which allows the user to specify invariants. If the invariant is false, the write fails.

2.2.4.6 Redis

Redis [22] is an in-memory key-value store with disk persistence. It provides high availability with Redis Sentinel and automatic partitioning with Redis Cluster.

Redis supports several values such as strings, lists, sets, sorted sets, among others. Operations over these values run atomically.

To provide persistence, Redis can dump the dataset to disk periodically or append each command to a log.

A form of transactions is also provided, although rollbacks are not supported. In fact, Redis commands may fail during a transaction if called with the wrong syntax or against keys holding the wrong data types. In this case Redis will keep executing the rest of the transaction instead of aborting.

2.2.4.7 Riak

Riak [2] is a distributed, geo-replicated, and highly available key-value store that provides eventual consistency. A form of strong consistency is also available but it is considered experimental at this point and not suited for production environments.

For achieving convergence, Riak supports state-based CRDTs and provides several data types such as flags, registers, counters, sets, and maps.

Borrowing heavily from Dynamo's design it implements consistent hashing and virtual nodes for data distribution across replicas. Like Dynamo it also uses vector clocks to trace dependencies between object versions, however it extends this functionality by allowing the developer to use dotted version vectors [23] instead.

2.2.4.8 Summary of Data Stores

This section approached several different key-value stores, exploring their similarities and differences. Namely, aspects such as consistency guarantees, partitioning schemes, replication, and conflict resolution were discussed.

All key-value stores presented are highly available and provide some form of eventual consistency but only Akka Distributed Data, AntidoteDB, and Riak have support for CRDTs. Of all the data stores, Redis is the only one that does not provide a partitioning strategy out of the box. However, one can use Redis together with Redis Cluster which does provide a partitioning solution similar to consistent hashing where each key is part of a hash slot.

When it comes to data representation, all of the data stores except Cassandra follow a key-value data model. Cassandra opts instead to use a hybrid model.

2.3 Computing Frameworks

Computing Frameworks provide programmers with a way of processing data in a distributed, fault-tolerant, and parallel manner. These frameworks automatically manage dispatching, scheduling, and IO functionalities, exposing only a simple (and usually functional) API which allows the programmer to only focus on describing the computation.

Methods of processing these computations can be divided in (i) batch and (ii) stream.

Batch In a batch system, the computation is executed at once, processing all input that is known when the processing starts and producing the result. Frameworks for batch processing typically import data from data stores (such as SQL or NoSQL databases) or file systems (such as HDFS [24]) and perform a sequence of computations over it. The biggest downside is that only historical data can be processed.

Stream In a stream-based system, data is processed as it arrives to the system. In this case, frameworks can still import from data stores and file systems, but they may also

make good use of message broker systems such as Kafka [25] to receive and process values in realtime.

Unlike highly available key-value stores, these systems do not typically allow the addition of extra computation nodes after having started their computations.

We now present a few of the most well known computing frameworks, and for frameworks that are open to public use we also present a WordCount example for comparison.

2.3.1 Apache Hadoop

Apache Hadoop [26] is a fault-tolerant distributed computing framework designed with batch processing in mind. It is composed of several modules, most notably the Hadoop Distributed File System [24] and Hadoop MapReduce which are inspired by Google's File System [27] and MapReduce [28] respectively.

Hadoop allows for the processing of large data sets in large clusters using the MapReduce programming model. It greatly leverages data locality whenever possible to reduce network traffic.

Because it makes use of the MapReduce programming model it has the benefit of breaking down computations into smaller chunks, which allows for programs to be automatically parallelized and executed on large clusters.

HDFS HDFS provides a distributed file system that partitions datasets across several servers called DataNodes and stores metadata on a separate server called the NameNode. The system allows for a replication factor to be set on a file-by-file basis. All servers are fully connected, forming a clique network. HDFS also provides an API that exposes the location of specific file blocks. Hadoop can use this information to schedule tasks to nodes where the relevant computing data is located, which improves read performance.

Hadoop MapReduce Hadoop MapReduce is based on the programming model of the same name. In this programming model the application programmer uses the functional constructs `map` and `reduce` to break computations into small pieces. These computations are then automatically parallelized and executed on large clusters by the framework. A third (and optional) construct, `combine`, allows for the partial merging of data in mappers which speeds up programs where the mappers produce too many duplicate or similar mergeable records.

Listing 2.1, shows a WordCount example using Apache Hadoop in the Scala programming language. The mapper splits sentences into words and emits tuples containing the word and the number one. The reducer aggregates all the tuples matching each unique word and emits a tuple containing the word and the sum of each word's tuple values.

Listing 2.1: WordCount in Apache Hadoop (using Scala)

```
class TokenizerMapper extends Mapper[Object,Text,Text,IntWritable] {
  val one = new IntWritable(1)
  val word = new Text

  override def map(key:Object, value:Text,
                   context:Mapper[Object,Text,Text,IntWritable]#Context) = {
    for (t <-  value.toString().split("\\s")) {
      word.set(t)
      context.write(word, one)
    }
  }
}

class IntSumReducer extends Reducer[Text,IntWritable,Text,IntWritable] {
  override
  def reduce(key:Text, values:java.lang.Iterable[IntWritable],
             context:Reducer[Text,IntWritable,Text,IntWritable]#Context) = {
    val sum = values.foldLeft(0) { (t,i) => t + i.get }
    context.write(key, new IntWritable(sum))
  }
}

object WordCount {
  def main(args:Array[String]):Int = {
    val conf = new Configuration()
    val otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs
    if (otherArgs.length != 2) {
      println("Usage: wordcount <in> <out>")
      return 2
    }
    val job = new Job(conf, "word count")
    job.setJarByClass(classOf[TokenizerMapper])
    job.setMapperClass(classOf[TokenizerMapper])
    job.setCombinerClass(classOf[IntSumReducer])
    job.setReducerClass(classOf[IntSumReducer])
    job.setOutputKeyClass(classOf[Text])
    job.setOutputValueClass(classOf[IntWritable])
    FileInputFormat.addInputPath(job, new Path(args(0)))
    FileOutputFormat.setOutputPath(job, new Path((args(1))))
    if (job.waitForCompletion(true)) 0 else 1
  }
```

2.3.2 Spark

Spark [29] is a system which allows for the batch processing of data in-memory on large clusters in a fault-tolerant manner. The system allows reusing the intermediate results built across multiple computations without having to write them to external storage as opposed to other similar processing systems. This allows for the re-transformation and

re-computation of data without the bottlenecks incurred by writing to storage which allow for measurable speedups.

Spark's programming model uses Resilient Distributed Datasets (RDDs), which are read-only, partitioned collections of records. RDDs may only be created through deterministic operations (transformations) over data in stable storage or other RDDs. RDDs store just enough information on how they were derived from other datasets so they can be reconstructed after a failure. Users can also control persistence and partitioning of RDDs, and indicate which RDDs will be reused and choose storage strategies for them.

Spark also builds up a log of the transformations used to build the datasets rather than the data itself. This allows for the quick recovery of the datasets by rerunning the computations in the case of a node crash.

Listing 2.2, shows a WordCount example using Spark in the Scala programming language. We start by opening a text file from HDFS. Mapping each of the lines in the file, splitting the line and flattening the result. Mapping each of the words and outputting a tuple containing the word and the number one. And finally, reducing over the collection by using the words as keys and summing the value for each key. The resulting collection contains a tuple for each existing word in the text file and the number of times it appeared. The collection is then saved in HDFS as a text file.

Listing 2.2: WordCount in Spark (using Scala)

```
1 val textFile = sc.textFile("hdfs://...")
2 val counts = textFile.flatMap(line => line.split("␣"))
3                      .map(word => (word, 1))
4                      .reduceByKey(_ + _)
5 counts.saveAsTextFile("hdfs://...")
```

2.3.3 Spark Streaming

Spark Streaming is an implementation of Discrete Streams [30], and is built on top of Spark as an extension. This extension provides the system with streaming computation capabilities, but still manages to deal with faults and slow nodes (stragglers). An interesting property of Spark Streaming is that it can combine streaming datasets with regular Spark RDDs, allowing the user to join recent data with historical data.

Instead of using a continuous operator model like in most other realtime computing frameworks, Discrete Streams opt to structure their computations as a set of short, stateless, deterministic tasks. The state is then propagated across tasks in Spark RDDs which allow for easy and deterministic recomputations in the case of faults.

The WordCount example for Spark Streaming is the same as Spark's, which is presented in listing 2.2.

2.3.4 Storm

Storm [31] is a realtime fault-tolerant and distributed stream data processing system. Typically a Storm system will pull data from a queue such as provided by systems like Kafka [25] and run queries over it.

Storm's data processing architecture consists of streams of tuples that flow through topologies. Topologies are composed of Spouts and Bolts. A Spout is a source of tuples for a given topology. A Bolt is essentially a consumer, doing some processing over the tuples received and passing them to the next set of bolts.

Storm topologies can have cycles, so they can be pictured as a directed graph. Each vertex in this graph represents a Spout or Bolt and an edge represents the data flow between Spouts and Bolts or Bolts and Bolts.

Listing 2.3, shows a WordCount example using Storm in the Scala programming language. It defines a Bolt (SplitSentence) that splits sentences into lists of words, another Bolt (WordCount) that counts the number of times a word is received, and a third bolt (Print) that prints received values to standard output. A default spout (RandomSentenceSpout) is used that spouts sentences that are randomly generated.

Our Storm topology is defined as follows:

1. RandomSentenceSpout is our initial spout;
2. SplitSentence is a bolt that aggregates values from RandomSentenceSpout;
3. WordCount is a bolt that aggregates values from SplitSentence;
4. Print is a bolt that groups values from WordCount.

Listing 2.3: WordCount in Storm (using Scala)

```
1  class SplitSentence extends StormBolt(outputFields = List("word")) {
2    def execute(t: Tuple) = t matchSeq {
3      case Seq(sentence: String) => sentence split "␣" foreach
4        { word => using anchor t emit (word) }
5      t ack
6    }
7  }
8
9  class WordCount extends StormBolt(List("word", "count")) {
10   var counts: Map[String, Int] = _
11   setup {
12     counts = new HashMap[String, Int]().withDefaultValue(0)
13   }
14   def execute(t: Tuple) = t matchSeq {
15     case Seq(word: String) =>
16       counts(word) += 1
17       using anchor t emit (word, counts(word))
18       t ack
```

```
19    }
20  }
21
22  class Print extends StormBolt(List()){
23    override def execute(input: Tuple): Unit = println(input)
24  }
25
26
27  object WordCountTopology {
28    def main(args: Array[String]) = {
29      val builder = new TopologyBuilder
30
31      builder.setSpout("randsentence", new RandomSentenceSpout, 5)
32      builder.setBolt("split", new SplitSentence, 8)
33          .shuffleGrouping("randsentence")
34      builder.setBolt("count", new WordCount, 12)
35          .fieldsGrouping("split", new Fields("word"))
36      builder.setBolt("print", new Print).shuffleGrouping("count")
37
38      val conf = new Config
39      conf.setDebug(true)
40      conf.setMaxTaskParallelism(3)
41
42      val cluster = new LocalCluster
43      cluster.submitTopology("word-count", conf, builder.createTopology)
44      Thread sleep 10000
45      cluster.shutdown
46    }
47  }
```

2.3.5 Percolator

Percolator [32] is a system built for large-scale incremental processing of updates over large datasets. It was developed to replace Google's previous batch-based indexing system for their web search index. The system allows Google to process the same number of documents per day (compared to their old approach) while still reducing the average age of documents in search results by 50%.

To structure computations Percolator uses Observers. Observers act like event handlers and are invoked when a user-specified column changes.

Percolator provides multi-row ACID compliant transactions by implementing snapshot isolation semantics.

However, despite providing high performance incremental computations, if a computation's result cannot be broken down into small updates then it is better handled by a MapReduce-like [28] system.

2.3.6 Titan

Titan [33] is a system that leverages computational CRDTs to enable the incremental stream processing of data. It uses computational CRDTs to perform elementary computations while maintaining computation specific invariants. As with other CRDTs, computational CRDTs can be freely replicated without conflicts and modified concurrently without coordination.

More deeply, Titan can be treated as a decentralized storage system with an environment that supports the execution of computational CRDTs. In this system, complex computations are defined as a graph created by chaining computational CRDTs.

The system runs on a set of nodes in a single cluster, organizing the nodes in a single-hop DHT which allows for any two nodes to communicate directly. Each individual node manages, stores, and provides the execution of a collection of computational CRDTs or computational CRDT partitions. Each computational CRDT is versioned and a sequence of versions is maintained. The computational CRDTs are partitioned as specified by the programmer.

2.3.7 Lasp

Lasp [34] presents a new programming model by bringing together ideas from deterministic dataflow programming and CRDTs. It provides CRDTs as built-in data types in the language, and allows application developers to deterministically compose different kinds of CRDTs. Furthermore, Lasp programs can also apply functional programming primitives such as *map*, *filter*, and *fold* over the composed CRDTs to transform them.

Previous research [35] formalized lattice variables as a primitive for parallel computations, but this programming model differs from the distributed setting where different types of failures may arise.

Despite appearing to be able to operate over non-monotonic values, Lasp operates over a CRDT's metadata, which is monotonic. Also, because a Lasp program is equivalent to a functional program, this means that application programmers can reason about the program in a deterministic way (since a functional program provides referential transparency).

Because Lasp makes such heavy use of CRDTs, it can be leveraged for building scalable, high-performance applications even when facing intermittent connectivity - including but not limited to IoT and mobile applications.

Listing 2.4, shows a WordCount example in Lasp using the Erlang programming language. We begin by reading a file into memory, splitting the file into several lines, and splitting each line into words. We then declare a grow only map CRDT (G-Map) using Lasp and load each of the words in our collection into the G-Map using a tuple with the word and the atom *increment*. The atom *increment* denotes that if the key already exists in the G-Map, its value will be incremented by one. Finally, we use Lasp's query function

to retrieve the value of our G-Map which has words as keys and the number of times they appeared as values.

Listing 2.4: WordCount in Lasp (using Erlang)

```
{ok, Binary} = file:read_file('...'),
{ok, {Map, _, _, _}} = lasp:declare(gmap),

lists:foreach(fun (Word) ->
  lasp:update(Map, {Word, increment}, a)
end, binary:split(Binary, [<<"\n">>, <<"␣">>], [global])),

{ok, Result} = lasp:query(Map),
io:fwrite("~p~n", [Result]).
```

2.4 Summary

In this chapter we have covered the work that lays the foundation of the development of this thesis.

The chapter primarily examined CRDTs, delta-CRDTs, and computational CRDTs. These data types provide strong semantics for building eventually consistent replicated systems, easing their development.

We also approached several different key-value stores, exploring their similarities and differences. Namely, aspects such as consistency guarantees, partitioning, replication and conflict resolution were discussed. The most relevant being Akka Distributed Data, AntidoteDB, and Riak – all of which can be extended by the work presented in future chapters.

Finally, we dived on distributed computing frameworks touching on their specific features and programming models. Of the discussed frameworks, Lasp and Titan stand out by allowing the composition of different replicated data types to produce complex computations.

In the next chapter we present the non-uniform replication model which captures the insight of partially incremental state-based computational CRDTs, where replicas only propagate elements which may change the results of other replicas, and formalizes it.

Chapter 3

Non-uniform replication model

In this chapter we introduce the non-uniform replication model and formalize its semantics for an eventually consistent system that synchronizes by exchanging operations. The model we introduce captures the semantics of partially incremental state-based computational CRDT synchronization where replicas only exchange elements when they may change the results of remote replicas. This behavior allows replicas to maintain only a portion of the total state while still correctly replying to read operations.

Furthermore, the model is generic enough to be applied to all kinds of replicated objects.

3.1 System model

We consider an asynchronous distributed system composed by n nodes, in which nodes may exhibit fail-stop faults but not byzantine faults. We assume nodes are connected via reliable links and the communication system supports at least one communication primitive, *mcast(m)*, that can be used by a process to send a message to all other processes in the system. A message which is sent by a correct process is eventually received by either all correct processes or none.

Without loss of generality, we assume that the system replicates a single object. An object is defined as a tuple $(\mathcal{S}, s^0, \mathbb{Q}, \mathcal{U}_p, \mathcal{U}_e)$, where $\mathcal{S}$ is the set of valid states of the object, $s^0 \in \mathcal{S}$ is the initial state of the object, $\mathbb{Q}$ is the set of read-only operations, $\mathcal{U}_p$ is the set of prepare-update operations, and $\mathcal{U}_e$ is the set of effect-update operations.

A read-only operation executes only at the replica where the operation is invoked, its source, and has no side-effects, i.e., the state of an object does not change after the operation executes. To update the state of the object, a prepare-update operation, $u_p \in \mathcal{U}_p$, is issued. A u_p operation executes only at the source, has no side-effects, and generates

an effect-update operation, $u_e \in \mathcal{U}_e$. At the source replica, u_e executes immediately after being generated from u_p.

We denote the state that results from executing operation o in state s as $s \bullet o$. For a prepare-update or read-only operation, $o \in \mathbb{Q} \cup \mathcal{U}_p$, $s \bullet o = s$. The result of applying operation $o \in \mathbb{Q} \cup \mathcal{U}_p \cup \mathcal{U}_e$ in state $s \in \mathcal{S}$ is denoted as $o(s)$.

We number the execution of effect-update operations at some replica sequentially from 1. The k^{th} operation execution at replica i will be noted o_i^k, where $o \in \mathcal{U}_e$. The state of a replica i after executing n operations is $s_i^n = s^0 \bullet o_i^1 \bullet \ldots \bullet o_i^n$. We denote the multi-set of operations executed to reach state s_i^n as $O(s_i^n) = \{o_i^1, \ldots, o_i^n\}$.

Given that only effect-update operations produce side-effects, we restrict our analysis to these operations to simplify reasoning about the evolution of replicas in the system. To be precise, the execution of a u_p operation generates an instance of an effect-update operation. For simplicity of presentation we refer to instances of operations simply as operations. With O_i the set of operations generated at node i, the set of operations generated in an execution, or simply the set of operations in an execution, is $O = O_1 \cup \ldots \cup O_n$.

Henceforth we refer to the set of effect-update operations, $\mathcal{U}_e$, as $\mathcal{U}$ for simplicity of presentation and reasoning.

3.2 System convergence

We denote the state of the replicated system as a tuple $(s_1, s_2, \ldots, s_n)$, with s_i the state of the replica i. We consider that the state of the replicas is synchronized via a replication protocol that exchanges messages among the nodes of the system. When messages are delivered they are executed in order to update the state of the replica.

We say a system is in a quiescent state if, for a given set of operations, the replication protocol has propagated all messages necessary to synchronize all replicas, i.e., additional messages sent by the replication protocol will not modify the state of the replicas. In general, replication protocols attempt to achieve a convergence property, where the state of any two replicas is equivalent in a quiescent state.

Definition (Equivalent state)**.** Two states, s_i and s_j, are *equivalent*, $s_i \equiv s_j$, iff the result of executing some operation $o_n \in \mathbb{Q} \cup \mathcal{U}$ after executing a sequence of operations $o_1, \ldots, o_{n-1}$ with $o_1, \ldots, o_n \in \mathbb{Q} \cup \mathcal{U}$ in both states is equal, i.e., $o_n(s_i \bullet o_1 \bullet \ldots \bullet o_{n-1}) = o_n(s_j \bullet o_1 \bullet \ldots \bullet o_{n-1})$

This property is enforced by most replication protocols, which either provide a strong or weaker form of consistency [15, 36, 37]. We note that this property does not require the internal state of the replicas to be the same, but only that replicas always return the same result for any executed sequence of operations.

For our replication model, we propose relaxing this property and instead require only that the execution of read-only operations return the same value. We name this property *observable equivalence* and define it formally as follows.

Definition (Observable equivalent state)**.** Two states, s_i and s_j, are *observable equivalent*, $s_i \overset{o}{\equiv} s_j$, iff the result of executing some operation $o_n \in \mathbb{Q}$ after executing a sequence of operations $o_1, \ldots, o_{n-1}$ with $o_1, \ldots, o_n \in \mathbb{Q}$ in both states is equal, i.e., $o_n(s_i \bullet o_1 \bullet \ldots \bullet o_{n-1}) = o_n(s_j \bullet o_1 \bullet \ldots \bullet o_{n-1})$

We now define a non-uniform replication system as one that guarantees only that replicas converge to an observable equivalent state.

Definition (Non-uniform replicated system)**.** We say that a replicated system is non-uniform if once the replication protocol reaches a quiescent state it can guarantee that the state of any two replicas in the system is observable equivalent.

3.3 Non-uniform eventual consistency

3.3.1 Eventual Consistency

For any given execution, with O the operations of the execution, we say a replicated system provides *eventual consistency* iff in a quiescent state: (i) every replica executed all operations belonging to O; and (ii) the state of any pair of replicas is equivalent.

To achieve the first property a sufficient condition is to propagate all generated operations using reliable broadcast, and to execute any received operations. A simple way of achieving the second property is to only allow the existence of commutative operations. In this manner, if all operations commute with each other then the execution of any serialization of O in the initial state of the object will always lead to an equivalent state.

Henceforth, unless stated otherwise, we assume that all operations are commutative. In this case, as all serializations of O are equivalent, we denote the execution of a serialization of O in state s simply as $s \bullet O$.

3.3.2 Non-uniform eventual consistency

For any given execution, with O the operations of the execution, we say a replicated system provides *non-uniform eventual consistency* iff in a quiescent state: (i) the state of any replica is observable equivalent to the state obtained by executing some serialization of O; and (ii) the state of any pair of replicas is observable equivalent.

For a given set of operations in an execution O, we say that $O_{core} \subseteq O$ is a set of core operations of O iff $s^0 \bullet O \overset{o}{\equiv} s^0 \bullet O_{core}$. We define a set of operations that are irrelevant to the final state of the replicas as follows: $O_{masked} \subseteq O$ is a set of masked operations of O iff $s^0 \bullet O \overset{o}{\equiv} s^0 \bullet (O \setminus O_{masked})$.

Theorem 1 (Sufficient conditions for NuEC)**.** A replication system provides *non-uniform eventual consistency (NuEC)* if, for the set of operations O, in their execution the following conditions hold: (i) every replica executes a set of core operations of O; and (ii) all operations commute.

Proof. From the definition of core operations of O, and by the fact that all operations commute, it follows immediately that if a replica executes the set of core operations, then the final state of the replica is observable equivalent to the state obtained by executing a serialization of O. Additionally, any two replicas always reach an observable equivalent state. □

3.4 Protocol for non-uniform eventual consistency

We now build on the sufficient conditions for providing *non-uniform eventual consistency* to design a replication protocol that attempts to minimize the number of operations propagated to other replicas. The key idea is to avoid propagating operations that are part of the masked set of operations. The challenge is to achieve this using only local information, which includes only a subset of the executed operations.

Algorithm 7 Replication algorithm for non-uniform eventual consistency

```
1:  S : state: initial s^0                                   ▷ Object state
2:  log_recv : set of operations: initial {}
3:  log_local : set of operations: initial {}                ▷ Local operations not propagated
4:  log_coreLocal : set of operations: initial {}            ▷ Local core operations not propagated
5:
6:  execOp(op): void                                         ▷ New operation generated locally
7:      if hasImpact(op, S) then
8:          log_coreLocal = log_coreLocal ∪ {op}
9:      log_local = log_local ∪ {op}
10:     S = S • op
11:
12: opsToPropagate(): set of operations                      ▷ Operations that must be propagated
13:     ops = maskedForever(log_local ∪ log_coreLocal, S, log_recv)
14:     log_local = log_local \ ops
15:     log_coreLocal = log_coreLocal \ ops
16:     opsImpact = log_coreLocal ∪ hasObservableImpact(log_local, S, log_recv)
17:     opsPotImpact = mayHaveObservableImpact(log_local, S, log_recv)
18:     return opsImpact ∪ opsPotImpact
19:
20: sync(): void                                             ▷ Propagates local operations to remote replicas
21:     ops = opsToPropagate()
22:     compactedOps = compact(ops)                          ▷ Compacts the set of operations
23:     mcast(compactedOps)
24:     log_coreLocal = {}
25:     log_local = log_local \ ops
26:     log_recv = log_recv ∪ ops
27:
28: on receive(ops): void                                    ▷ Process remote operations
29:     S = S • ops
30:     log_recv = log_recv ∪ ops
```

Algorithm 7 presents the pseudo-code of an algorithm for achieving *non-uniform eventual consistency*. The algorithm does not address the durability concerns regarding masked operations, as this is discussed in a later section.

Apart from the state of the object, the algorithm also maintains three sets of operations: (i) $log_{coreLocal}$, the set of core effect-update operations generated at the local replica which have not yet been propagated to other replicas; (ii) log_{local}, the set of effect-update operations generated at the local replica which have not yet been propagated to other replicas; and (iii) log_{recv}, the set of effect-update operations that have been propagated to all replicas, including operations which were locally generated.

When an effect-update operation is generated, the *execOp* function must be called. This function adds the new operation to the log of local operations and updates the state of the local object. If the new operation has an impact on the observable state of the object, then it is also added to the log of local core operations.

Function *sync* is used to propagate local operations to remote replicas. It begins by calculating which new operations must be propagated, compacts the resulting set of operations, multicasts the compacted set of operations, and finally updates the local set of operations accordingly. A call to function *compact* (line 22) is used to reduce the total number of operations to be propagated while guaranteeing that the execution is equivalent. When a replica receives a set of operations (line 28), it updates the local state and the set of operations accordingly.

Function *opsToPropagate* addresses the key challenge of deciding which operations need to be propagated to other replicas. To this end, we divide the operations in four groups.

First, the *forever masked* operations, which are operations that will remain in the set of masked operations independently of operations that may execute in the future. Given a top-K example, an operation that adds a new player score will forever mask all operations that added a lower score for the same player. These operations are removed from the sets of local operations.

Second, the *core* operations, as computed locally. These operations must always be propagated, as they will (typically) impact the observable state at every replica.

Third, the operations that might impact the observable state when considering concurrent operations that execute at other replicas but are not propagated, as they are masked. Given that there is no way to know which operations executed in remote replicas have not yet been propagated, it is necessary to propagate these operations as well. Given a modified top-K example where instead of using the highest score we use the total sum of all scores a player has, an add operation that would not move a player to the top would fall into this category.

Fourth, the remaining operations that might impact the observable state in the future, depending on the evolution of the observable state. These operations remain in log_{local}. In a top-K example, an operation that adds a score that will not be in the top, as computed

locally, is in this category as it might become part the top scores after a larger score is removed.

For proving that the algorithm can be used to provide non-uniform eventual consistency, we need to prove the following property.

Theorem 2. Algorithm 7 guarantees that in a quiescent state, considering all operations O in an execution, all replicas have received all operations in a core set O_{core}.

Proof. To prove this property, we need to prove that there exists no operation that has not been propagated by some replica and that is required for any O_{core} set. Operations in the first category have been identified as masked operations regardless of any other operations that might have or will be executed. Thus, by definition of masked operations, a O_{core} set will not need to include these operations. The fourth category includes operations that do not influence the observable state when considering all executed operations – if they might have impact, they would be in the third category. Thus, these operations do not need to be in a O_{core} set. All other operations are propagated to all replicas. Thus, in a quiescent state, every replica has received all operations that impact the observable state. □

3.4.1 Fault-tolerance

Non-uniform replication aims at reducing the cost of communication and the size of replicas, by avoiding propagating operations that do not influence the observable state of the object. This raises the question of the durability of operations that are not immediately propagated to all replicas.

One way to solve this problem is to consider a setting with multiple data centers where objects are replicated both within a data center (among local replicas) and outside a data center (to other remote data centers). In this case, the durability of masked operations is guaranteed by replicating objects among replicas within a data center. However, if the entire data center fail-stops the masked operations will be lost.

Another way is to simply consider all replicas as independent and immediately propagate every operation to $f + 1$ replicas to tolerate f faults. This ensures that an operation survives even in the case of f faults. We note however that it would be necessary to adapt the proposed algorithm, so that in the case where a replica receives an operation for durability reasons, it would propagate the operation to other replicas if the source replica fails. This can be achieved by considering it as any local operation (and introducing a mechanism to filter duplicate reception of operations).

3.5 Summary

In this chapter we have presented the non-uniform replication model, a model for replicated objects in a distributed system where replicas only synchronize updates which

impact their observable state. We have also formalized the semantics of the model for an eventually consistent system that synchronizes by exchanging operations. The model allows us to reduce the dissemination cost of sending updates between replicas, as well as the size of replica objects.

In the next chapter we introduce a set of operation-based CRDT designs that adopt this model.

Chapter 4

Operation-based NuCRDTs

In this chapter we show how to apply the concept of non-uniform replication to design useful operation-based CRDTs. These designs are inspired by the state-based computational CRDTs proposed by Navalho *et al.* [9], which also allow replicas to diverge in their quiescent state.

We note that not all designs we present allow replicas to diverge. More precisely, replicas of the histogram, average, and top-K (without removals) designs do not diverge when their state is quiescent as their operations are always considered core. This demonstrates the generic nature of the model.

4.1 Average

The first design we introduce is an object that computes the average of the values added to the object. The data type maintains a sum of values added and the number of values added, which allows it to compute the average, and can be used to compute the average of ratings in a web page. The semantics of the operations defined in the Average CRDT is the following: *add(n)* adds *n* to the sum of values and increments the number of values by 1; *add(sum,num)* adds *sum* to the sum of values and *num* to the number of values; *get()* returns the current average of the object.

Algorithm 8 presents a design that implements this semantics. The prepare-update operation *add(n)* is provided for user convenience, as it generates an effect-update *add(n,1)*. The prepare-update operation *add(sum,num)* generates an effect-update *add(sum,num)*.

Each object replica maintains only a tuple, *accum*, with the sum of values added and the number of values added. The execution of an *add(sum,num)* consists in adding *sum* to the first tuple field and *num* to the second tuple field.

Function HASIMPACT always returns *true* because the average is influenced by every

Algorithm 8 Design: Average

```
1:  accum : ⟨sum, num⟩ : initial ⟨0, 0⟩
2:
3:  GET(): double
4:      return sum/num
5:
6:  prepare ADD(n)                                   ▷ Shim prepare-update
7:      generate add(n, 1)
8:
9:  prepare ADD(sum, num)
10:     generate add(sum, num)
11:
12: effect ADD(sum, num)
13:     accum.sum = accum.sum + sum
14:     accum.num = accum.num + num
15:
16: HASIMPACT(op, S) : boolean
17:     return true                  ▷ In this data type operations always have impact
18:
19: MASKEDFOREVER(log_local, S, log_recv) : set of operations
20:     return {}                        ▷ In this data type operations are always core
21:
22: MAYHAVEOBSERVABLEIMPACT(log_local, S, log_recv) : set of operations
23:     return {}                        ▷ In this data type operations are always core
24:
25: HASOBSERVABLEIMPACT(log_local, S, log_recv) : set of operations
26:     return {}                        ▷ In this data type operations are always core
27:
28: COMPACT(ops): set of operations
29:     accums = {⟨sum, num⟩ : add(sum, num) ∈ ops}
30:     acc = tuplePointwiseSum(accums)
31:     return {add(acc.sum, acc.num)}
```

value. Functions MASKEDFOREVER, MAYHAVEOBSERVABLEIMPACT, and HASOBSERVABLEIMPACT always return the empty set since operations in this data type are always core. Function COMPACT takes a set of instances of *add(sum,num)* operations and joins them together into a single *add(sum,num)* – this behavior is similar to joining delta-groups in delta-based CRDTs [11].

4.2 Histogram

We now introduce the Histogram CRDT that maintains a histogram of values added to the object. To this end, the data type maintains a mapping of bins to integers and can be used to maintain a voting system on a website. The semantics of the operations defined in the histogram is the following: *add(n)* increments the bin n by 1; *merge(histogram$_{delta}$)* adds the information of a histogram into the local histogram; *get()* returns the current

histogram.

Algorithm 9 Design: Histogram

```
1:  histogram : map bin ↦ n : initial []
2:
3:  GET(): map
4:      return histogram
5:
6:  prepare ADD(bin)                                   ▷ Shim prepare-update
7:      generate merge([bin ↦ 1])
8:
9:  prepare MERGE(histogram)
10:     generate merge(histogram)
11:
12: effect MERGE(histogram_delta)
13:     histogram = pointwiseSum(histogram, histogram_delta)
14:
15: HASIMPACT(op, S) : boolean
16:     return true                  ▷ In this data type operations always have impact
17:
18: MASKEDFOREVER(log_local, S, log_recv) : set of operations
19:     return {}                        ▷ In this data type operations are always core
20:
21: MAYHAVEOBSERVABLEIMPACT(log_local, S, log_recv) : set of operations
22:     return {}                        ▷ In this data type operations are always core
23:
24: HASOBSERVABLEIMPACT(log_local, S, log_recv) : set of operations
25:     return {}                        ▷ In this data type operations are always core
26:
27: COMPACT(ops): set of operations
28:     deltas = {hist : merge(hist_delta) ∈ ops}
29:     hist = pointwiseSum(deltas)
30:     return {merge(hist)}
```

This data type is implemented in the design presented in Algorithm 9. The prepare-update *add(n)* generates an effect-update *merge*($[n \mapsto 1]$). The prepare-update operation *merge(histogram)* generates an effect-update *merge(histogram)*.

Each object replica maintains only a map, *histogram*, which maps *bins* to integers. The execution of a *merge*($histogram_{delta}$) consists of doing a pointwise sum of the local histogram with $histogram_{delta}$.

As in the previous design, function HASIMPACT always returns *true* because adding a value always changes the state of the histogram. Functions MASKEDFOREVER, MAYHAVEOBSERVABLEIMPACT, and HASOBSERVABLEIMPACT always return the empty set since operations in this data type are always core. Function COMPACT takes a set of instances of *merge* operations and joins the histograms together returning a set containing only one *merge* operation.

4.3 Top-K

In this section we introduce the top-K CRDT. This data type allows access to the top-K elements added to the object and can be used, for example, for maintaining a leaderboard in an online game. The top-K defines only one update operation, *add(id,score)*, which adds element *id* with score *score*. The *get()* operation simply returns the K elements with largest scores.

Algorithm 10 Design: Top-K

```
1:  elems : set of ⟨id, score⟩ : initial {}
2:
3:  GET(): set
4:      return elems
5:
6:  prepare ADD(id, score)
7:      generate add(id,score)
8:
9:  effect ADD(id, score)
10:     elems = topK(elems ∪ {⟨id, score⟩})
11:
12: HASIMPACT(op, S): boolean
13:     R = S • op
14:     return S ≠ R
15:
16: MASKEDFOREVER(log_local, S, log_recv): set of operations
17:     adds = {add(id_1, score_1) ∈ log_local :
18:         (∃add(id_2, score_2) ∈ log_recv : id_1 = id_2 ∧ score_2 > score_1)
19:     return adds
20:
21: MAYHAVEOBSERVABLEIMPACT(log_local, S, log_recv): set of operations
22:     return {}   ▷ In this data type operations are always either core or forever masked
23:
24: HASOBSERVABLEIMPACT(log_local, S, log_recv): set of operations
25:     return {}   ▷ In this data type operations are always either core or forever masked
26:
27: COMPACT(ops): set of operations
28:     return ops                          ▷ This data type does not use compaction
```

Algorithm 10 presents the design of the top-K CRDT. The prepare-update *add(id,score)* generates an effect-update *add(id,score)*.

Each object replica maintains only a set of K tuples, *elems*, with each tuple being composed of an *id* and a *score*. The execution of *add(id,score)* inserts the element into the set, *elems*, and computes the top-K of *elems* using the function *topK*. The order used for the *topK* computation is as follows: $\langle id_1, score_1\rangle > \langle id_2, score_2\rangle$ iff $score_1 > score_2 \vee (score_1 = score_2 \wedge id_1 > id_2)$. We note that the *topK* function returns only one tuple for each element *id*.

Function HASIMPACT checks if the top-K elements change after executing the given operation. Function MASKEDFOREVER computes the adds that become masked by other add operations for the same *id* that are larger according to the defined ordering. Due to the way the top is computed, the lower values for some given *id* will never be part of the top. Functions MAYHAVEOBSERVABLEIMPACT and HASOBSERVABLEIMPACT always return the empty set since operations in this data type are always core or forever masked. Function COMPACT simply returns the given *ops* since the design does not require compaction.

4.4 Top-K with removals

This section introduces the design for the top-K with removals CRDT. This data type extends the previous one with a remove operation. The data type could be used to maintain a leaderboard in an online game, where the remove operation is used to remove scores of a player that has been cheating.

For defining the semantics of our data type, we start by defining the happens before relation among operations. To this end, we start by considering the happens-before relation established among the events executed [21]. The events that are considered relevant are: the generation of an operation at the source replica, its local execution, propagation, and execution at other replicas. We say that operation op_i happens before operation op_j iff the generation of op_i happened before the generation of op_j in the partial order of events.

The semantics of the operations defined in the top-K with removals is the following: *add(id,score)* adds a new pair to the object; *rmv(id)* removes any pair with *id* that was added by an operation that happened-before the *rmv* (note that this will include also operations that have not been propagated to the source replica of the remove). This leads to an *add-wins* policy [38], where a remove has no impact on concurrent adds. The *get()* operation returns the K pairs with the largest *score*.

Algorithm 11 presents a design that implements this semantics. The prepare-update operation *add* generates an effect-update *add* that has an additional parameter consisting in a timestamp $\langle$*replica identifier*,*val*$\rangle$, with *val* a monotonically increasing integer. The prepare-update operation *rmv* generates an effect-update *rmv* that includes an additional parameter consisting in a vector clock that summarizes the add operations that happened before the remove operation. The object maintains a vector clock, *vc*, that is updated when a new add is generated or executed locally. Additionally, this vector clock is updated whenever a replica receives a message from a remote replica (to summarize also the adds known by the sender that have not been propagated to this replica).

Besides this vector clock, each replica maintains: (i) a set *elems* with the elements added by *add* operations known locally (and that have not been removed yet); and (ii) a map *removes* that for each element *id* has a vector clock with a summary of the add operations that happened before all removes of *id* (for simplifying the presentation of

Algorithm 11 Design: Top-K with removals

```
1:  elems : set of ⟨id, score, ts⟩ : initial {}
2:  removes : map id ↦ vectorClock: initial []
3:  vc : vectorClock: initial []
4:
5:  GET() : set
6:      els = topK(elems)
7:      return {⟨id, score⟩ : ⟨id, score, ts⟩ ∈ els}
8:
9:  prepare ADD(id, score)
10:     generate add(id, score, ⟨getReplicaId(), ++vc[getReplicaId()]⟩)
11:
12: effect ADD(id, score, ts)
13:     if removes[id][ts.siteId] < ts.val then
14:         elems = elems ∪ {⟨id, score, ts⟩}
15:         vc[ts.siteId] = max(vc[ts.siteId], ts.val)
16:
17: prepare RMV(id)
18:     generate rmv(id,vc)
19:
20: effect RMV(id, vc_rmv)
21:     removes[id] = pointwiseMax(removes[id], vc_rmv)
22:     toRemove = {⟨id_0, score, ts⟩ ∈ elem : id = id_0 ∧ ts.val < vc_rmv[ts.siteId]}
23:     elems = elems \ toRemove
24:
25: HASIMPACT(op, S): boolean
26:     R = S • op
27:     return topK(S) ≠ topK(R)
28:
29: MASKEDFOREVER(log_local, S, log_recv): set of operations
30:     adds = {add(id_1, score_1, ts_1) ∈ log_local :
31:         (∃add(id_2, score_2, ts_2) ∈ log_local : id_1 = id_2 ∧ score_1 < score_2 ∧ ts_1.val < ts_2.val)∨
32:         (∃rmv(id_3, vc_rmv) ∈ (log_recv ∪ log_local) : id_1 = id_3 ∧ ts_1.val < vc_rmv[ts_1.siteId]}
33:     rmvs = {rmv(id_1, vc_1) ∈ log_local :
34:         ∃rmv(id_2, vc_2) ∈ (log_local ∪ log_recv) : id_1 = id_2 ∧ vc_1 < vc_2}
35:     return adds ∪ rmvs
36:
37: MAYHAVEOBSERVABLEIMPACT(log_local, S, log_recv): set of operations
38:     return {}                          ▷ This case never happens for this data type
39:
40: HASOBSERVABLEIMPACT(log_local, S, log_recv): set of operations
41:     adds = {add(id_1, score_1, ts_1) ∈ log_local : ⟨id_1, score_1, ts_1⟩ ∈ topK(S.elems)}
42:     rmvs = {rmv(id_1, vc_1) ∈ log_local :
43:         ∃⟨id_2, score_2, ts_2⟩ ∈ topK(S.elems) : id_1 = id_2 ∧ ts_2.val < vc_1[ts_2.siteId]}
44:     return adds ∪ rmvs
45:
46: COMPACT(ops): set of operations
47:     return ops                         ▷ This data type does not use compaction
```

the algorithm, we assume that a key absent from the map has associated a default vector clock consisting of zeros for every replica).

The execution of an *add* consists in adding the element to the set of *elems* if the add has not happened-before a previously received remove for the same element – this can happen as operations are not necessarily propagated in causal order. The execution of a *rmv* consists in updating *removes* and deleting from *elems* the information for adds of the element that happened-before the remove. To verify if an add has happened-before a remove, we check if the timestamp associated with the add is reflected in the remove vector clock (lines 13 and 22). This ensures the intended semantics of the remove operation.

The order used for the *topK* computation (in GET and HASIMPACT) is as follows: $\langle id_1, score_1, ts_1 \rangle > \langle id_2, score_2, ts_2 \rangle$ iff $score_1 > score_2 \vee (score_1 = score_2 \wedge id_1 > id_2) \vee (score_1 = score_2 \wedge id_1 = id_2 \wedge ts_1 > ts_2)$. We note that the *topK* function returns only one tuple for each element *id*.

We now analyze the code of the functions used in the replication protocol. Function HASIMPACT simply checks if the top-K elements change after executing the new operation. Function MASKEDFOREVER computes: the local adds that become masked by other adds (those for the same element with a lower score) and removes (those for the same element that happened-before the remove); the removes that become masked by other removes (those for the same element that have a smaller vector clock). In the latter case, it is immediate that a remove with a smaller vector clock becomes irrelevant after executing one with a larger vector clock. In the former case, a local add for an element is masked by a more recent local add for the same element with a larger score as it is not possible to remove only the effects of the later add without removing the effect of the older one. A local add also becomes permanently masked by a remove that happened-after the add.

Function MAYHAVEOBSERVABLEIMPACT returns the empty set, as for having impact on any observable state an operation must also have impact on the local observable state after the object is in a quiescent state.

Function HASOBSERVABLEIMPACT computes: the local adds that have not been propagated to other replicas and are part of the top-K at the local replica; and the local removes that will remove an element in the top-K. Besides operations executed after the last synchronization, this function returns the operations that became relevant for the top-K due to the execution of some other operation (an add can be made relevant by the fact that an element in the top has been removed, and a remove can be made relevant by the fact that an older add had become relevant). Function COMPACT simply returns the given *ops* since the design does not require compaction.

4.5 Filtered Set

We now introduce the design for the Filtered Set CRDT (F-Set). The data type allows access to elements that satisfy a specific filter and can be used, for example, for maintaining

a collection of employees that can be filtered by age, gender, or some other constraint.

The semantics of the operations defined is the following: *add(e)* adds a new element to the object; *changeFilter(f)* changes the filter of the object; *get()* returns the elements in the object that satisfy the current filter function.

Algorithm 12 presents a design that implements this CRDT. The prepare-update for operation *add* generates an effect-update *add*. The prepare-update operation *changeFilter* generates an effect-update *changeFilter* with an additional parameter, a *timestamp*. The additional parameter is composed of a replica identifier, *siteId*, and a value, *val*, and is used to order concurrent *changeFilter* updates at different replicas.

Each F-Set object replica maintains: (i) a set *elems* with the elements added by *add* operations; (ii) an anonymous function *filter* with the active *filter* (if a *changeFilter* operation has not occured yet, the default *filter* which satisfies all possible elements is used); and (iii) a timestamp *ts* with the timestamp of the last *changeFilter* operation.

The execution of an *add* simply adds the element to the set of *elems*. The execution of a *changeFilter* operation updates the current *filter* and *ts* if the new timestamp is greater than the current one.

Function HASIMPACT checks if the execution of the new operation adds a new element that satisfies the active filter or if it changes the filter. Function MASKEDFOREVER computes the local adds that are masked by other adds and the changeFilters that become masked by other changeFilters. In the latter case, it is immediate that a changeFilter with a smaller timestamp becomes irrelevant after executing the one with a larger timestamp. In the former case, a local add for an element is masked by some other add for the same element that is already core.

Function MAYHAVEOBSERVABLEIMPACT returns the empty set, as for having impact on any observable state an operation must also have impact on the local observable state.

Function HASOBSERVABLEIMPACT computes the set of adds that have not yet been propagated that add elements which satisfy the active filter where the element being added has not previously been propagated or received. Besides operations executed after the last synchronization, this function returns the operations that became relevant for computing the current filtered set due to the execution of some other operation (an add can be made relevant by the fact that the filter has changed). Function COMPACT simply returns the given *ops* since the design does not require compaction.

Algorithm 12 Design: F-Set

```
1:  elems : set of elements : initial {}
2:  filter : λ : initial e ↦ true
3:  ts : tuple of ⟨siteId, val⟩ : initial ⟨0, 0⟩
4:
5:  GET() : set
6:      return {e ∈ elems : filter(e) = true}
7:
8:  prepare ADD(e)
9:      generate add(e)
10:
11: effect ADD(e)
12:     elems = elems ∪ {e}
13:
14: prepare CHANGEFILTER(f)
15:     generate changeFilter(f, ⟨getReplicaId(), getTimestamp()⟩)
16:
17: effect CHANGEFILTER(f, timestamp)
18:     if timestamp.val > ts.val ∨ (timestamp.val = ts.val ∧ timestamp.siteId > ts.siteId) then
19:         filter = f
20:         ts = timestamp
21:
22: HASIMPACT(op, S): boolean
23:     R = S • op
24:     filteredR = {e ∈ R.elems : R.filter(e) = true}
25:     filteredS = {e ∈ S.elems : S.filter(e) = true}
26:     return filteredR ≠ filteredS ∨ R.ts ≠ S.ts
27:
28: MASKEDFOREVER(log_local, S, log_recv): set of operations
29:     adds = {add(e_1) ∈ log_local : ∃add(e_2) ∈ log_recv : e_1 = e_2}
30:     filters = {changeFilter(f_1, ts_1) ∈ log_local :
31:         ∃changeFilter(f_2, ts_2) ∈ log_local ∪ log_recv : ts_2.val > ts_1.val
32:             ∨(ts_2.val > ts_1.val ∧ ts_2.siteId > ts_1.siteId}
33:     return adds ∪ filters
34:
35: MAYHAVEOBSERVABLEIMPACT(log_local, S, log_recv): set of operations
36:     return {}                                  ▷ This case never happens for this data type
37:
38: HASOBSERVABLEIMPACT(log_local, S, log_recv): set of operations
39:     adds = {add(e) ∈ log_local : filter(e) = true ∧ ∄add(e) ∈ log_recv}
40:     return adds
41:
42: COMPACT(ops): set of operations
43:     return ops                                 ▷ This data type does not use compaction
```

4.6 Top Sum

We now present the design for the Top Sum CRDT. This design appears similar to previous top-K designs but the semantics is quite different. In this data type instead of associating scores with some identifier, the identifier is associated with the sum of all its scores. The data type can be used for maintaining a leaderboard in an online game where every time a player completes some challenge it is awarded some number of points, with the current score of the player being the sum of all points awarded.

This design is interesting because it is hard to know which operation may have impact in the observable state. For example, consider a scenario with two replicas, where the score of last element in the top is 100. If the known score of an element is 90, an add of 5 received in one replica may have impact in the observable state if the other replica has also received an add of 5 or more. One approach would be to propagate these operations, but this would lead to propagating all operations. To try to minimize the number of operations propagated we use the following heuristic inspired by escrow transactions [39]: for each id, each replica maintains the sum of operations that have been propagated to all replicas. A replica propagates local operations to other replicas if the sum of local adds exceeds the difference between the minimum element in the top and the sum of adds propagated to all replicas divided by the number of replicas.

The design defines only one update operation, *add(id, n)*, which simply increments the score of *id* by *n*. The *get()* operation returns a mapping of the top-K identifiers and corresponding scores, as defined by the function *topK* used in the algorithm. The order used for the *topK* function is as follows: $\langle id_1, v_1 \rangle > \langle id_2, v_2 \rangle$ iff $v_1 > v_2 \vee (v_1 = v_2 \wedge id_1 > id_2)$.

Algorithm 13 presents a design that implements this semantics. The only prepare-update operation, *add*, generates an effect-update *add* with the same parameters. Each replica of this data type maintains only one field, *state*, which represents the current state of the object as a mapping between identifiers and their current score sum. The execution of an *add(id, n)* simply increments the score sum of *id* by *n*.

Function HASIMPACT checks if the top-K elements change after executing the new operation. Function MASKEDFOREVER returns the empty set, as operations in this design can never be forever masked. Function MAYHAVEOBSERVABLEIMPACT computes the set of *add* operations that can potentially have an impact on the observable state using the previously defined heuristic.

Function HASOBSERVABLEIMPACT computes the set of *add* operations that have not yet been propagated that have their corresponding *id* present in the top-K.

Function COMPACT takes a set of instances of *add* operations and compacts the *add* operations that affect the same identifier.

Algorithm 13 Design: Top Sum

```
 1: state : map id ↦ sum: initial []
 2:
 3: GET() : map
 4:     return topK(state)
 5:
 6: prepare ADD(id, n)
 7:     generate add(id, n)
 8:
 9: effect ADD(id, n)
10:     state[id] = state[id] + n
11:
12: HASIMPACT(op, S): boolean
13:     R = S • op
14:     return topK(S.state) ≠ topK(R.state)
15:
16: MASKEDFOREVER(log_local, S, log_recv): set of operations
17:     return {}                          ▷ This case never happens for this data type
18:
19: MAYHAVEOBSERVABLEIMPACT(log_local, S, log_recv): set of operations
20:     top = topK(S.state)
21:     adds = {add(id, _) ∈ log_local : s = sum_val({add(i, n) ∈ log_local : i = id})
22:         ∧ s > ((min(top) − (S.state[id] − s)) / getNumReplicas())}
23:     return adds
24:
25: HASOBSERVABLEIMPACT(log_local, S, log_recv): set of operations
26:     top = topK(S.state)
27:     adds = {add(id, _) ∈ log_local : id ∈ top}
28:     return adds
29:
30: COMPACT(ops): set of operations
31:     adds = {add(id, n) :
32:         id ∈ {i : add(i, _) ∈ ops}
33:         ∧ n = sum({k : add(id_1, k) ∈ ops : id_1 = id})}
34:     return adds
```

4.7 Summary

In this chapter we have presented a collection of useful designs for operation-based CRDTs that follow the non-uniform replication model formalized in the previous chapter. The presented designs demonstrate the generic nature of the model, allowing it to be applied to different data types. This can be seen in the designs of the Average, Histogram, and Top-K (without removals) where the data types do not exhibit state divergence when they are in a quiescent state.

In the next chapter we evaluate some of these designs against comparable implementations using state-of-the-art CRDTs that adopt a uniform replication model.

Chapter 5

Comparing NuCRDTs with CRDTs

In this chapter we evaluate our data type designs. To this end, we compare our designs (Op NuCRDT), against: delta-based CRDTs [11] that propagate all operations to all replicas in an efficient manner (Delta CRDT); and the state-based computational CRDT designs proposed by Navalho *et al.* [9] (State C-CRDT).

Our evaluation was performed by simulation using a discrete event simulator. To show the benefit in terms of bandwidth and storage, we measured the total size of messages sent between replicas for synchronization and the mean size of replicas.

We extended our designs, Op NuCRDT, and the computational CRDT designs to support up to 2 replica faults by propagating all operations to, at least, 2 other replicas besides the source replica. This extension was only applied to the top-K with removals and the F-Set.

We simulated a system with 5 replicas for each data type. In each simulation run 500,000 update operations were generated. The values used in each operation (regardless of the data type) were randomly selected using a uniform distribution. Furthermore, in all simulation runs a replica synchronizes after executing 100 updates.

5.1 Histogram

For the Histogram data type simulation we assume a total of 1,000 bins. The implementation of the histogram for the both Delta CRDT and the State C-CRDT used a collection of counter CRDTs where each counter represents one bin. Additionally each local counter for the State C-CRDT must maintain n extra counters, one for each replica the CRDT has previously received updates from. The Op NuCRDT was the same as presented in algorithm 9. Figure 5.1 shows the results for this data type.

In both metrics, our histogram data type achieved parity with the Delta CRDT. This

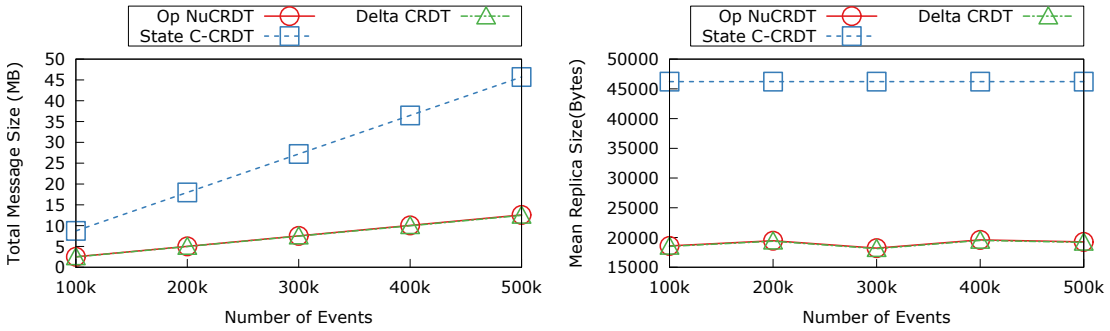

Figure 5.1: Histogram: total message size and mean replica size

is to be expected since they have the same space complexity and propagate only the bins that have changed (the delta) when synchronizing. The State C-CRDT performs up to 3.75 times worse in total message size and 2.5x worse in mean replica size than the other designs as when synchronizing updates it must send the entire state of its local histogram and must also maintain a histogram for every replica site it has previously received updates from.

5.2 Top-K

The experiment for the top-K used the following configuration: K was configured to 100, player identifiers were selected with a uniform distribution from a domain of 10,000, and scores were generated randomly with a uniform distribution from 0 to 250,000. The State C-CRDT followed the implementation proposed by Navalho *et al.* [9] where the data type keeps only the top-K elements, and always propagates its full state. The Delta CRDT was implemented as a G-Set, where each new element is always added to the set. The Op NuCRDT implemented algorithm 10. The results are shown in figure 5.2.

Our data type achieved a significantly lower bandwidth cost when compared to the State C-CRDT (up to 50 times lower), and a decent improvement over the Delta CRDT (4 times lower). The Delta CRDT proved to be particularly efficient when compared to the

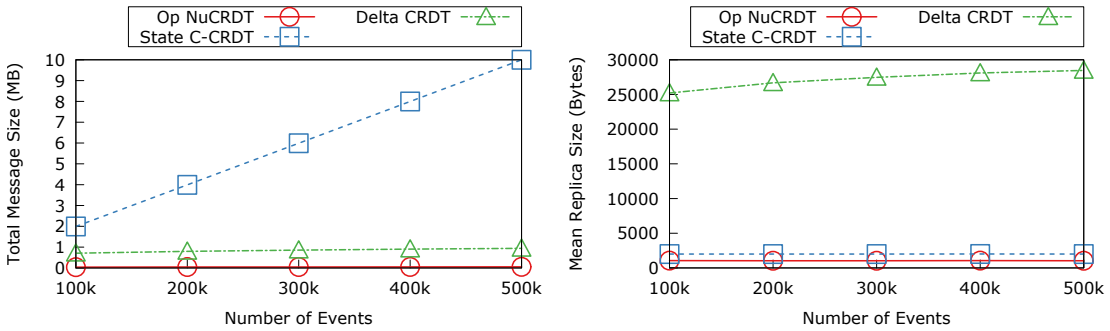

Figure 5.2: Top-K: total message size and mean replica size

State C-CRDT (up to 12 times lower) since each replica only propagates new elements, while the State C-CRDT must always propagate its full state which causes its total message size to increase accordingly.

The results for the replica size show the space efficiency of both our data type and the State C-CRDT. The large Delta CRDT replica sizes are expected since the G-Set maintains all the added elements.

5.3 Top-K with removals

In the experiments with the top-K with removals, K was configured to be 100, player identifiers were selected with a uniform distribution from a domain of 10,000, and scores were generated randomly with a uniform distribution from 0 to 250,000.

The Delta CRDT was implemented as a 2P-Set, a data type that composes two G-Sets together – one for additions and one for removals; when an addition happens the element is inserted into the first G-Set, when a remove occurs the affected elements in the first G-Set are moved to the second G-Set. The State C-CRDT followed the implementation proposed by Navalho *et al.* [9]. The Op NuCRDT implemented algorithm 11.

Given the expected usage of a top-K for supporting a leaderboard, we expect the remove to be an infrequent operation (to be used only when a user is removed from the game). Thus, our workloads were designed with this in consideration. Figure 5.3 shows the results for a workload of 95% of adds and 5% of removes. Figure 5.4 shows the results for a workload of 99% of adds and 1% of removes. And finally, figure 5.5 shows the results for a workload of 99.95% of adds and 0.05% of removes.

In all workloads our design achieved a significantly lower bandwidth cost when compared to either the Delta CRDT (up to 25 times lower) and State C-CRDT (up to 6 times lower). The reason for this is that our design only propagates operations that will be part of the top-K. In the Delta CRDT, each replica propagates all new operations and not only those that are a part of the top. In the State C-CRDT design, every time the top is modified, the new top has to be propagated. Additionally, the proposed design for the

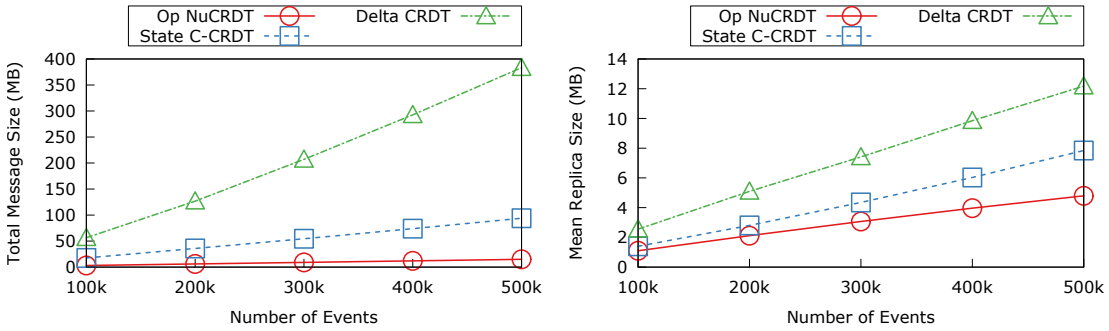

Figure 5.3: Top-K with removals: total message size and mean replica size with a workload of 95% adds and 5% removes

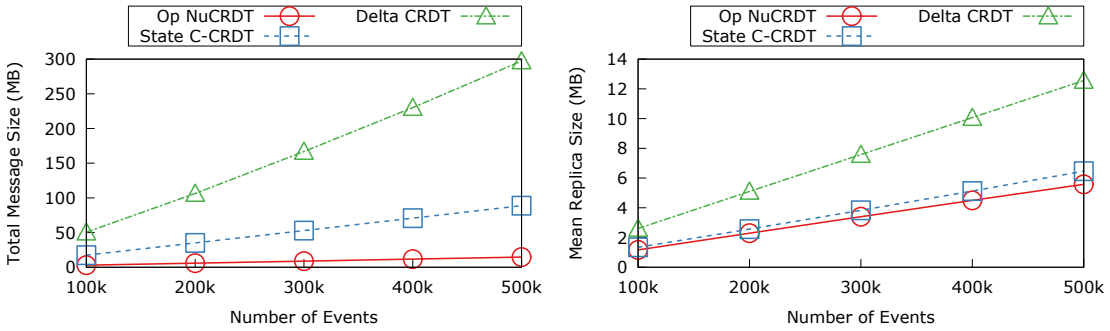

Figure 5.4: Top-K with removals: total message size and mean replica size with a workload of 99% adds and 1% removes

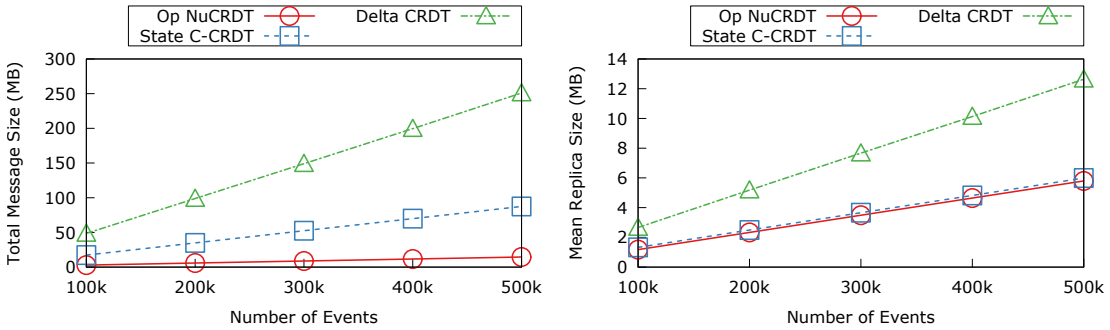

Figure 5.5: Top-K with removals: total message size and mean replica size with a workload of 99.95% adds and 0.05% removes

State C-CRDT always propagates removes.

The results for the replica size show that our design also manages to be more space efficient than other designs, up to 2.5 times smaller than the Delta CRDT and up to 1.6 times smaller than the State C-CRDT. This is a consequence of the fact that each replica, besides maintaining information about local operations, only keeps information from remote replicas received for guaranteeing fault-tolerance and those that have influenced the top-K at some moment in the execution. The State C-CRDT design additionally keeps information about all removes. The Delta CRDT keeps information about all elements that have not been removed or overwritten by a larger value. We note that as the percentage approaches zero, the replica sizes of our design and that of the State C-CRDT design start to converge to the same value. The reason for this is that the information maintained in both designs is similar and our more efficient handling of removes starts becoming irrelevant. The opposite is also true: as the number of removes increases, our design becomes even more space efficient when compared to the State C-CRDT.

5.4 Filtered Set

The F-Set data type simulation used the following configuration: integers added to the set were randomly selected with a uniform distribution from a domain of 100,000,000, and the filter function used validated whether an element is a multiple of 2 or not. We note that the filter function was not changed during the simulation. The Delta CRDT implementation used a G-Set, the State C-CRDT also used a G-Set but followed the idempotent design proposed by Navalho *et al.* [9]. We used the idempotent design here due to the fact that the partially incremental design incurs very significant dissemination overhead since when a replica synchronizes it must also propagate all results it knows about other replicas. This is impractical for big sets as the gains of not sending elements which do not validate the filter are much smaller than the overhead incurred by sending all results. The Op NuCRDT implemented algorithm 12. The results are shown in figure 5.6.

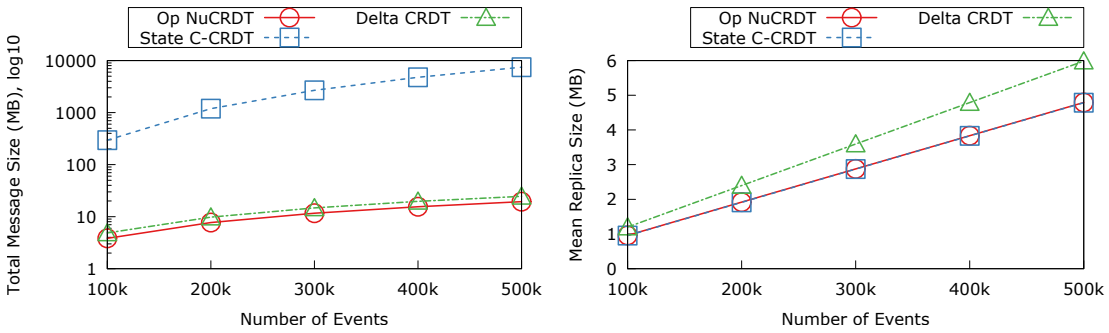

Figure 5.6: F-Set: total message size and mean replica size

Our data type achieved a significantly lower bandwidth cost when compared to the State C-CRDT (up to 384 times lower), and a modest improvement over the Delta CRDT (up to 1.3 times lower). The Delta CRDT sends a bit more information than our data type, due to the need to send all new elements to all replicas. The State C-CRDT due to following the idempotent design must send all elements that belong to the filtered set in every synchronization.

The results for the replica size show the space efficiency of the Op NuCRDT and the State C-CRDT designs, which achieve parity. The Delta CRDT was not as compact, since it must send all new elements to all replicas, resulting in larger replica sizes (up to 1.3 times larger).

5.5 Summary

In this chapter we have presented an evaluation of the proposed NuCRDT designs when compared with existing state-of-the-art alternatives, namely computational CRDTs and delta-based CRDTs. We have shown that our designs reduce the dissemination overhead

and replica size when compared to the state-of-the-art uniform designs. When specifically comparing with computational CRDTs our designs are able to improve on one, and sometimes both, metrics.

In the next chapter we present design and implementation of operation-based NuCRDTs in the AntidoteDB key-value store, together with the changes that were required in AntidoteDB to support our designs.

Chapter 6

Integration of NuCRDTs in AntidoteDB

This chapter describes AntidoteDB's architecture and details how we implemented support for non-uniformly operation-based CRDTs in AntidoteDB.

6.1 AntidoteDB architecture

AntidoteDB is a distributed database written in the Erlang programming language and leverages Riak Core [40], a framework for building distributed systems.

The system is globally distributed, running on several data centers at the same time. To provide scalability, AntidoteDB shards data among replicas, called partitions, within a data center using consistent hashing techniques. Read and write requests are served by the nodes that hold the data. AntidoteDB has support for highly available transactions [41] which can manipulate multiple objects in different replica nodes. A transaction's read and write operations execute in a single data center, by contacting only the replicas that hold the accessed data.

Figure 6.1 the AntidoteDB architecture where clients execute transactions on a data center. The system is responsible for correctly propagating the operations to the different replica nodes, as well as propagating operations asynchronously to remote data centers.

Each AntidoteDB node has four main components:

- **Transaction Manager:** This component implements the transaction protocol and is responsible for receiving client requests, executing them, coordinating transactions, and replying to client requests. Transaction operations are stored in the Log component – which also sends the operations to the InterDC Replication component. A read operation contacts the materializer component that materializes snapshots of objects stored in the nodes.

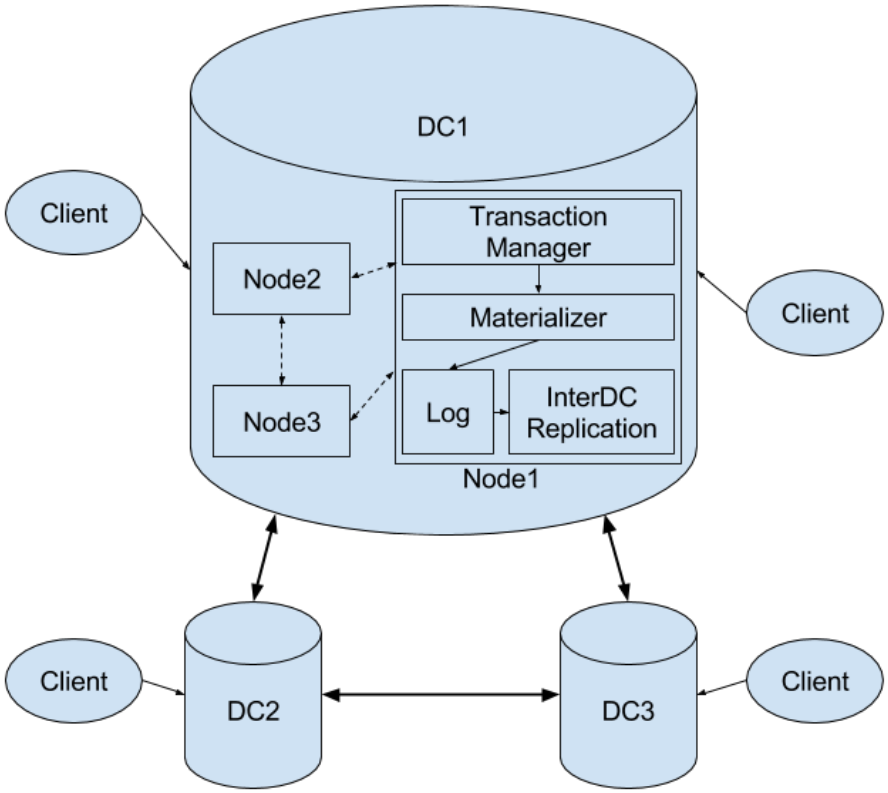

Figure 6.1: AntidoteDB Architecture

- **Materializer:** This component materializes and caches object versions (also called snapshots) requested by clients, by reading the log entries if necessary.
- **Log:** This component implements a log-based persistent layer. Updates are stored in this component when received and the log is persisted to disk to maintain data durability. This component also sends the updates to the InterDC Replication component to provide replication between data centers.
- **InterDC Replication:** This component is responsible for propagating updates among data centers. It has two parts. First, it uses ZeroMQ PubSub [42] to send updates from each log to each remote data center – in the form of transactions. Second, it implements a mechanism to advance the vector clock so that the received remote transactions can be made visible [14].

6.2 Implementing the data types

We began by implementing a few of the data types presented in earlier chapters, more specifically we implemented the following designs:

- Average, following algorithm 8;

- Top-K, following algorithm 10;
- Top-K with removals, following algorithm 11.

These implementations are self-contained in a module called `antidote_ccrdt` [43] which follows the module API already used by AntidoteDB's CRDTs [44], while adding the following data type APIs to help support our synchronization optimizations:

- CAN_COMPACT(op_1, op_2), verifies if the given effect-update operations can be compacted;
- COMPACT_OPS(op_1, op_2), returns the compacted operations;
- IS_REPLICATE_TAGGED(op), identifies if the given operation is tagged for replication (to maintain the durability of masked operations).

As AntidoteDB attempts to immediately propagate transactions after they commit, we had to modify this behavior to this end, we apply a buffering technique inside AntidoteDB to delay the propagation of the transactions – discussed in detail in a later section. Essentially this allows us to store effect-update operations in a temporary log which we later compact before synchronization. This technique makes use of CAN_COMPACT and COMPACT_OPS to remove forever masked operations. These functions are also used to avoid propagating operations that when locally executed were core but became masked. These operations are propagated only to a subset of replicas for durability.

To support designs where executing certain operations can cause previously existing operations to become core and requires these operations to be propagated, we changed the API for executing downstream effect-updates to allow returning a collection of effect-updates together with the new state of the object. Furthermore, the module also provides a function to identify which data types have this requirement, GENERATES_EXTRA_OPERATIONS.

6.3 Modifications in AntidoteDB to support NuCRDTs

To integrate our NuCRDT module into AntidoteDB we had to make sure we fulfilled the following set of requirements:

1. AntidoteDB must type check our data types and operations, as it currently does with CRDTs;
2. Prepare-update operations that will never impact the state of NuCRDTs should be discarded – these generate no-op effect-update operations in our data types, since AntidoteDB expects prepare-update operations to always return an effect-update;
3. For effect-update operations that have yet to be propagated to other data centers, we would like to compact these operations before propagation to reduce the number

of effect-updates to be transmitted and to optimize computations (as reducing the number of effect-updates also reduces the number of remote executions needed);

4. Effect-update operations that have no immediate impact in remote data center replicas should only be propagated to a subset of replicas for durability – these operations are tagged for replication in our data types;

5. When executing some effect-update operations locally, extra effect-update operations may be generated and will need to be propagated – these are operations that were previously executed locally but were considered as masked.

We now explain how we addressed these requirements.

6.3.1 Requirement 1: Operation typechecking

For supporting the typechecking of our data types, we extended AntidoteDB's type checker to also check for the data types and operations present in our `antidote_ccrdt` module. The modification was trivial, requiring only that we add our module's IS_TYPE(type) function to AntidoteDB's typechecker function.

6.3.2 Requirement 2: Ignoring no-op effect-updates

No-op effect-updates are generated when, for example, adding a player score to a top-K where the score being added does not fit into the top K. To entirely ignore no-op effect-updates inside AntidoteDB, we modified the Log component and the Materializer component to ignore effect-update operations where the operation payload is a no-op.

6.3.3 Requirement 3: Operation compaction

The goal of operation compaction was to use the log of effect-update operations that have yet to be propagated and compact the operations within it using the API functions provided by our data types. To give an example, consider a top-K where only the highest player score for each player is stored. If our log had two effect-updates: *add(1,50)* and *add(1, 70)*, the compaction of these two operations would result in *add(1, 70)*.

In AntidoteDB, as soon as a transaction commits it is immediately shipped to remote data centers and the operations of each log are propagated concurrently to maximize throughput and minimize coordination. This made our compaction technique harder. In our implementation we achieved a middle ground by buffering committed transactions for a brief, configurable, period before sending them to remote data centers. This allows us to compact the NuCRDT operations between different transactions within the delay.

To achieve this we modified the InterDC Replication component to buffer transactions that have been committed in the data center for the duration of a timer. Once the timer runs out it takes the transactions it has collected and compacts them into a single equivalent transaction that results in the same state for the affected objects.

The compaction mechanism is shown in algorithm 14 and works as follows. First, it takes the list of transactions and separates the effect-update operations that affect NuCRDTs and those that do not. The effect-update operations affecting NuCRDTs are stored in a map, where the key is a tuple containing the object key and the bucket where it is located, the value is a list of effect-update operations. Each entry in this map forms a log of unpropagated effect-update operations for each NuCRDT. This is where the compact functions in our API are applied. For each of the objects in the map, the log of operations is compacted incrementally by taking each element and attempting a compaction with each of the previous elements.

Algorithm 14 Algorithm for compacting collections of transactions

```
1:  compactCollection(transactions)
2:      map = takeNonUniformObjectOperations(transactions)
3:      otherOperations = takeUniformObjectOperations(transactions)
4:      for object, ops ∈ map do
5:          map[object] = compact(ops)
6:      allOperations = concatenate(getAllObjectOperations(map), otherOperations)
7:      lastTransaction = last(transactions)
8:      return replaceOperations(lastTransaction, allOperations)
9:
10: compact(ops):
11:     for op_1 ∈ ops do
12:         op_2 = op_1
13:         while hasPrevious(op_2) do
14:             op_2 = previous(op_2)
15:             if canCompact(op_2, op_1) then
16:                 ⟨new_2, new_1⟩ = compactOps(op_2, op_1)
17:                 if new_2 = no-op then remove(ops, op_2)
18:                 else if new_2 ≠ op_2 then replace(ops, op_2, new_2)
19:                 if new_1 = no-op then
20:                     remove(ops, op_1)
21:                     break
22:                 else if new_1 ≠ op_1 then replace(ops, op_1, new_1)
23:     return ops
```

Once the compaction finishes, the remaining operations are propagated to the remote data centers. A compacted operation will use the same commit time of the last transaction in the original list of transactions.

This modification has a few implications on the AntidoteDB causal transactional model as transactions that might have been compacted cannot be accessed individually in the remote data centers. The reason for this is that operations from earlier transactions are not being propagated and thus accessing the remaining effects of the transaction would violate the atomicity property. For example, if a client executes two transactions on DC1 within the buffer duration, then some other client executing transaction on DC2 cannot read a value that includes only the first transaction. Instead, it is necessary to guarantee

that all transactions in a buffer are contained in the snapshot.

To make sure AntidoteDB's stable snapshots evolve safely when compacting operations, we have modified the stabilization of vector clock snapshots such that they evolve in synchrony with the transaction buffer time. This ensures updates only become visible in a data center when all partitions in that data center observe that all transactions belonging to a buffer period have been delivered.

Due to the incurred overhead of the transaction buffering mechanism, it remains as a configurable option in AntidoteDB so the system user can make the trade-off between quicker overall replication or less network overhead for NuCRDTs.

6.3.4 Requirement 4: Durability of masked operations

The compaction mechanism creates one version of the log that contains both the core operations and the masked operations being propagated for durability, and a second version that only has the core operations. These two versions are then propagated to different subsets of data centers, where the first one is propagated to f data centers in order to maintain the durability of masked operations with up to f faults. The second version is propagated to the remaining data centers.

However, to broadcast different versions of a transaction to different data centers we had to modify the filter AntidoteDB uses for its ZeroMQ PubSub connection between data centers. As the initial filter contained only the partition number, the publisher could not distinguish between different data centers which made it impossible to send different versions to different data centers. To support this we simply added the data center identifier to the filter.

6.3.5 Requirement 5: Generating new operations from downstream operations

To support this requirement we modified the Materializer component to support generating a new operation when the execution of a downstream operation also returns new operations that must be propagated. However this new operation must not be generated immediately, instead since AntidoteDB uses object snapshots and a cache of operations these operations should only be generated when the operation that generated them is removed from the cache to avoid generating the same operation more than once.

An object's operations are removed from the cache once the number of cached operations for that object hits the configured threshold (50 by default) or when a read operation on the object is executed.

For example, given a top-5 where 45 additions have executed if we execute a remove operation that would cause a masked operation to become core this core operation would only be generated and propagated after 4 other operations executed or after a single read operation was executed.

6.4 Summary

In this chapter we discussed AntidoteDB's overall architecture and described our experience with implementing operation-based NuCRDTs in AntidoteDB. The implementation of the data types themselves was relatively simple, and the module that contains them is open-sourced on GitHub [43].

However, the modifications required to support the data types in AntidoteDB were not trivial. First, we had to add support to generate no-ops when an operation had no side effects. Second, as AntidoteDB's unit of propagation is a transaction and not an operation – and it is immediately propagated to remote data centers; we were forced to find a middle ground for log compaction by buffering transactions for a brief period and then compacting the collection of operations in those transactions. Third, AntidoteDB's ZeroMQ PubSub filter had no initial support for distinguishing between what data centers transactions were being sent to which was absolutely required to reduce the dissemination overhead of designs that must maintain the durability of masked operations.

In the next chapter we evaluate two of our operation-based NuCRDTs against operation-based CRDTs that model equivalent behavior in the AntidoteDB key-value store.

Chapter 7

Evaluation

In this chapter we evaluate the performance of Non-uniform CRDTs in AntidoteDB. To this end, we compare the Top-K and Top-K with removals Non-uniform CRDTs with the current available solution that uses an add-wins set CRDT. The add-wins set is implemented by generating a new unique token for every insert operation. A remove operation will remove all tokens associated with an element known in the replica where the operation was executed.

The experiments we present in this chapter try to assess whether the introduction of non-uniform replication in a geo-replicated database system allows to: (i) reduce the size of database replicas; (ii) reduce the bandwidth used for synchronizing replicas. Furthermore, we study the scalability of a system that uses non-uniform replication in comparison with a system using full replication.

7.1 Dissemination overhead and replica sizes

We started by measuring the size of the replicas and the bandwidth consumed for synchronizing replicas. To this end, we modified AntidoteDB to store in each data center the total size of messages transmitted for a given object. To measure the size of data type replicas we have introduced support for accessing the full object representation.

The experiment executes a sequence of randomly generated updates to different objects, where all different objects receive the same updates. The values used in each operation (regardless of the data type) were randomly selected using a uniform distribution. In the experiment we compare the Non-uniform CRDTs proposed in this work with the operation-based CRDTs currently available in AntidoteDB. Data points were recorded every 5,000 operations, by obtaining the total message size each data center had transmitted so far for each object and the size of the objects, which we later used to compute the

mean size of each object. All results represent the mean result of three independent runs.

The experiments were ran on Amazon Web Services EC2, using *m3.xlarge* machine instances for both the AntidoteDB nodes and the node issuing the benchmark operations. Each of the machines were launched in the *eu-west-1c* region. A total of 5 AntidoteDB nodes were used, each one forming its own data center (containing only one node). Each AntidoteDB node was configured to buffer transactions for a period of 200 milliseconds.

7.1.1 Top-K

We first evaluated the performance of the Top-K design. In this case we compared our design against an add-wins set that models the same semantics on the client side, by explicitly removing elements from the set which become masked. This experiment used the following configuration: K was configured to 100, player identifiers were selected with a uniform distribution from a domain of 10,000, and scores were generated randomly with a uniform distribution from 0 to 250,000. The results are shown in Figure 7.1.

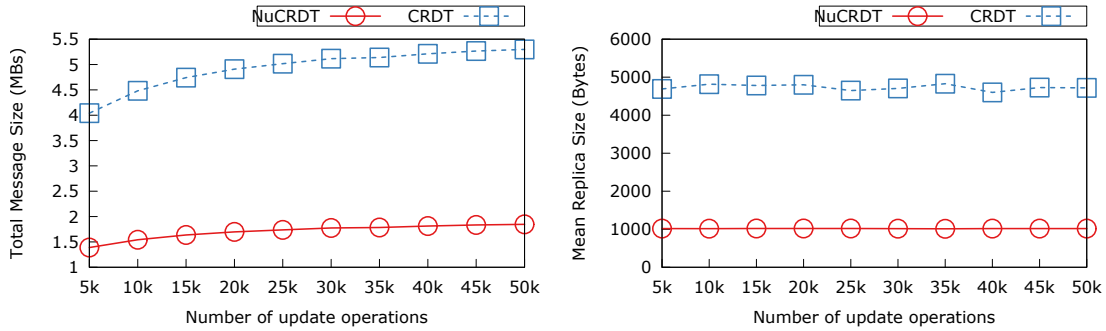

Figure 7.1: Top-K: total message size and mean replica size

For the total message size our data type achieved up to a 3 times lower dissemination cost. This is expected since to model the semantics of the top-K in the add-wins set, elements have to be explicitly removed once they are no longer part of the top.

The results for the replica size show the efficiency of the state representation of our data type (up to a 4.8 times reduction). Even though both objects mostly have the same number of elements (roughly 100 for the add-wins set and always 100 for the Top-K) our design implementation manages to have a better state representation since it does not require unique tokens like the add-wins set.

7.1.2 Top-K with removals

We now compare the design of the Top-K with removals against an add-wins set which models the same semantics on the client side, by explicitly managing the removal of each affected element as would occur in the Non-uniform CRDT.

In this experiment, K was configured to be 100, player identifiers were selected with a uniform distribution from a domain of 10,000, and scores were generated randomly

with a uniform distribution from 0 to 250,000. Furthermore, the system was configured to support from zero to two faults ($f = 0, f = 1, f = 2$) by propagating masked operations to f replicas.

As the simulations in section 5, given the expected usage of a top-K for supporting a leaderboard, we expect the remove to be an infrequent operation (to be used only when a user is removed from the game). Thus, the workload was chosen with this in consideration. Figure 7.2 shows the results for a workload of 95% of adds and 5% of removes.

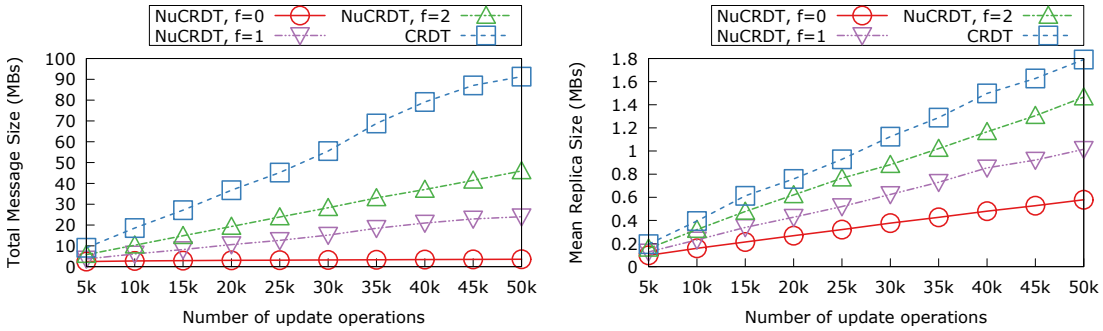

Figure 7.2: Top-K with removals: total message size and mean replica size with a workload of 95% adds and 5% removes

Our design achieved both a significant lower bandwidth cost (up to a 96% reduction for $f = 0$) and a lower replica size (up to a 67.7% reduction for $f = 0$) when compared to the add-wins set. This happens primarily because the add-wins set needs to propagate all elements to all replicas (even the ones that do not fit in the top) while our design only propagates the required elements to all replicas and the remaining elements are only propagated to a subset for durability.

When specifically comparing between the various instances of the Top-K with removals with varying degrees of replication, the total message size and mean replica size increases as expected. We note that when the data type tolerates up to 2 faults its mean replica size reaches 78% of the mean size of the add-wins set, while each element that is not in the top is replicated only in 60% of the replicas. This happens due to a few reasons: 1) The Top-K with removals must explicitly maintain more information regarding each element including the replica id and the replica timestamp, and 2) the Top-K with removals must maintain an explicit registry of removals.

7.2 Scalability

To evaluate the scalability of non-uniform replication we have used the Basho Bench [45] benchmarking tool developed by Basho Technologies. To use Basho Bench we developed a driver that specifies what operations can be executed for our use cases. For using AntidoteDB's original CRDT, the driver specification was extended to model the same semantics as NuCRDTs. To give an example, when modeling a top-K using an add-wins

	eu-west	eu-central	us-east	us-west	ap-northeast
eu-west	0.421	22.42	71.537	145.875	210.989
eu-central	22.42	0.417	89.241	156.304	254.216
us-east	71.537	89.241	0.459	61.699	143.058
us-west	145.875	156.304	61.699	0.453	117.695
ap-northeast	210.989	254.216	143.058	117.695	0.493

Table 7.1: Mean round-trip time between Amazon Web Services EC2 instances

set the elements which are no longer part of the top must be removed one by one – this is done explicitly by the client driver.

We now describe the benchmarking setup. These experiments were ran on Amazon Web Services EC2, using *m3.xlarge* machine instances for both the AntidoteDB nodes and the Basho Bench nodes. All benchmarks were ran against 5 AntidoteDB nodes, each one forming its own data center (containing only one node), resulting in a total of 5 data centers. The benchmark runs using 5 Basho Bench instances, each one in its own machine. Each Basho Bench instance spawned a configurable number of clients and connected to the data center node running in the same EC2 region.

Machines that ran AntidoteDB nodes were launched on the following region/availability zones: *eu-west-1c*, *eu-central-1a*, *us-east-1d*, *us-west-1c*, and *ap-northeast-1c*. Machines that ran Basho Bench nodes were launched in the same region and availability zone as the AntidoteDB node they were connecting to. The mean round-trip time over 100 Ping requests between each machine is shown in table 7.1.

Each AntidoteDB node was configured to buffer transactions for a period of 200 milliseconds. All benchmarks ran for 3 minutes. Each data point for each experiment represents the mean result of three independent runs. Prior to each run the AntidoteDB nodes were shutdown and their data was deleted, the software was then recompiled, the nodes were relaunched, and the data center nodes were reconnected. This ensured a fair benchmarking environment.

7.2.1 Top-K

We now present the evaluation results for the top-K and the add-wins implementation of a top-K. This experiment used the following configuration: K was configured to 100, player identifiers were selected with a uniform distribution from a domain of 10,000, and scores were generated randomly with a uniform distribution from 0 to 250,000. The results are presented in figure 7.3.

The results show that our non-uniform replication design scales much better that the add-wins set-based implementation of top-K. The reason for this lies in the fact that in the add-wins-based implementation it is necessary to remove an element whenever a new element is added to the top, resulting in a larger number of operations being executed. Additionally, in our design, as the top is populated with elements with large scores, the

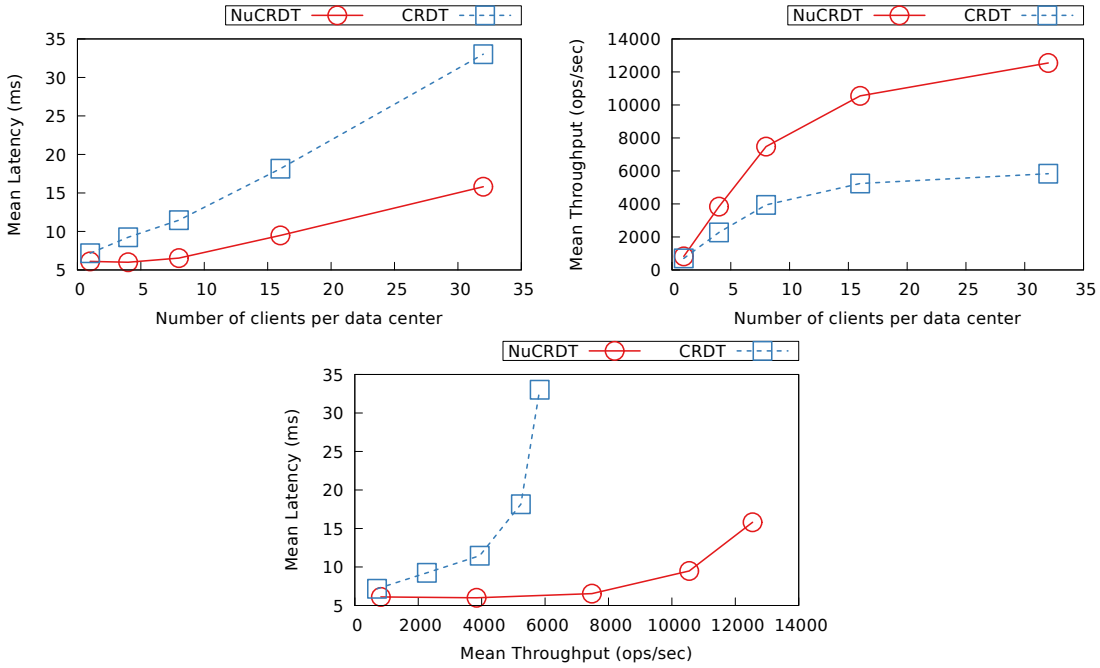

Figure 7.3: Top-K experiments

number of operation that are not immediately masked tends to zero.

7.2.2 Top-K with removals

We now present the evaluation results for the top-K with removals. In this case, we need to maintain the all inserted scores as a remove may delete only some of the scores. The configuration used in the experiments is the following: K was set to 100, player identifiers were selected with a uniform distribution from a domain of 10,000, and scores were generated randomly with a uniform distribution from 0 to 250,000. Furthermore, the system was configured to support from zero to two faults ($f = 0, f = 1, f = 2$) by propagating masked operations to f replicas. Similarly to the measurements of total message size and mean replica size, Figure 7.4 presents the results of a workload of 95% of adds and 5% of removes.

The results show that both design behave similarly under low load (up to 16 clients). For a larger number of clients per data center the mean latency of the add-wins set more than doubled while the mean latency of the NuCRDT design remained linear. For 128 clients per data center the add-wins set could not keep up with the increasing load and as result suffered a throughput drop. The NuCRDT design did not exhibit this behavior.

When comparing between the varying degrees of replication both latency and throughput were initially similar. However, after the number of clients per data center increased to more than 32 the latency also slightly increased for both $f = 1$ and $f = 2$. Correspondingly, the throughput also had a slower growth and could not reach the same value as $f = 0$.

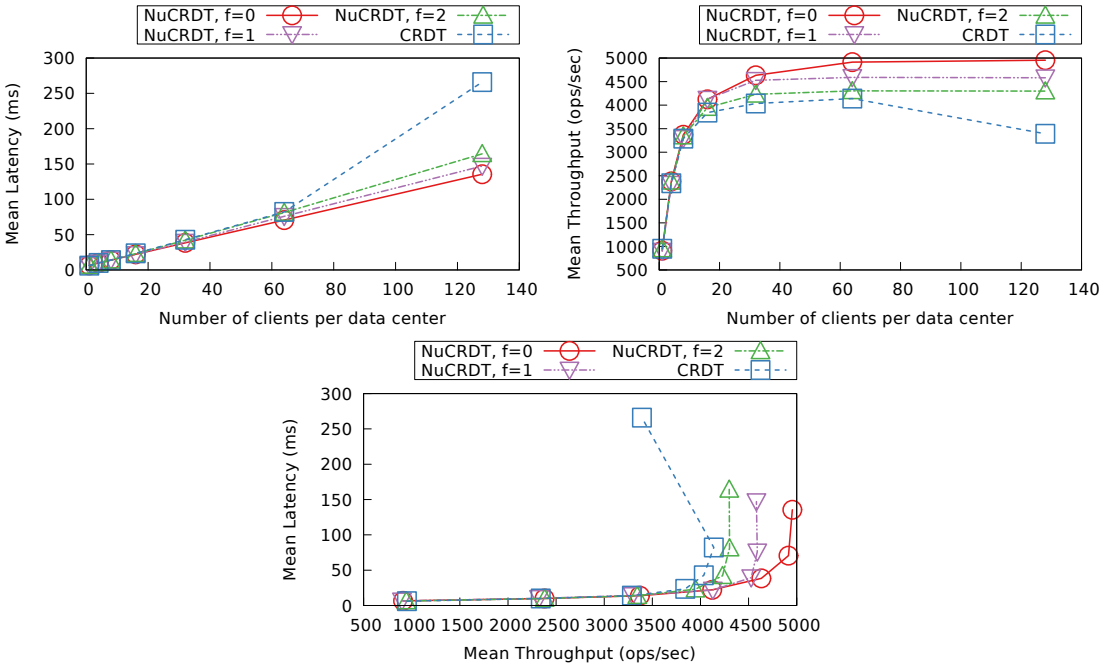

Figure 7.4: Top-K with removals experiments with a workload of 95% adds and 5% removes

We suspect that the marginal increase in scalability for this design is due to the top-K computation being executed inside the database, which could perhaps be optimized further.

7.3 Summary

The results presented in this chapter show that NuCRDT design perform better than their operation-based CRDTs counterparts. We first showed that the size of replicas and the bandwidth consumed by NuCRDT design is lower by up to 79% and 96% respectively. We also showed that the scalability of a system that used NuCRDT is better than when using operation-based CRDTs.

Chapter 8

Conclusion

In this thesis we have introduced a new replication model, the non-uniform replication model, and have formalized its semantics for an eventually consistent system that synchronizes by exchanging operations. Additionally we have shown how to apply the generic model to different kinds of operation-based CRDT designs.

Our simulated evaluations showed that our proposed operation-based NuCRDTs are equivalent to delta-based CRDTs and better than state-based computational CRDTs for the designs where the propagated operation is a representation of the state (such as in the Average and Histogram CRDTs). For other designs our NuCRDTs perform better than delta-based CRDTs and state-based computational CRDTs, significantly reducing the message dissemination overhead and lowering the replica sizes.

We have discussed how operation-based NuCRDTs can be implemented in a distributed key-value store that already supports CRDTs by describing our implementation in AntidoteDB. We have presented the changes we made to AntidoteDB that were required to support our designs, and the trade-offs they incur.

Finally, we have evaluated our NuCRDTs in AntidoteDB against operation-based CRDTs that model the same semantics on the client side. The results of our experiments showed that our NuCRDTs are more space efficient and incur a lesser dissemination overhead than the ones currently used in AntidoteDB. Furthermore, the NuCRDT design leads to a better scalability of the system, with a larger maximum throughput and a lower and more stable latency for operations.

8.1 Publications

Part of the results in this dissertation were submitted for publication:

Non-uniform replication for replicated objects G. Cabrita, N. Preguiça. Submitted to ACM Symposium on Principles of Distributed Computing, 2017.

8.2 Future Work

This thesis introduced the concept of non-uniform replication and formalized its semantics for an operation-based synchronization approach in the context of an eventually consistent system. Future work can study and formalize this replication model for other consistency models such as linearizability.

Additionally, more interesting data type designs which implement the non-uniform replication model can be explored, implemented, and evaluated using our modified version of AntidoteDB. This can potentially open up more use cases for CRDTs. Something we intend to explore in the future is the usefulness of NuCRDTs in the context of Big Data and Machine Learning environments.

Finally, the current implementation does not reflect a perfect fault-tolerant environment for masked operations which are only propagated for durability. As the receiving data centers for these operations are currently chosen arbitrarily it is impossible to know what subset of masked operations need to be re-replicated when a data center fails.

Bibliography

[1] Amazon.com, Inc. *Amazon DynamoDB*. Accessed: 2016-05-03. URL: http://aws.amazon.com/dynamodb.

[2] Basho Technologies, Inc. *Riak KV*. Accessed: 2016-05-10. URL: http://docs.basho.com/riak/kv.

[3] 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.

[4] 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.

[5] 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.

[6] N. Schiper, P. Sutra, and F. Pedone. "P-Store: Genuine Partial Replication in Wide Area Networks". In: *Proceedings of the 2010 29th IEEE Symposium on Reliable Distributed Systems*. SRDS '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 214–224. ISBN: 978-0-7695-4250-8. DOI: 10.1109/SRDS.2010.32. URL: http://dx.doi.org/10.1109/SRDS.2010.32.

[7] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. "Consistency-based Service Level Agreements for Cloud Storage". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: ACM, 2013, pp. 309–324. ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522731. URL: http://doi.acm.org/10.1145/2517349.2522731.

[8] T. Crain and M. Shapiro. "Designing a Causally Consistent Protocol for Geo-distributed Partial Replication". In: *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data*. PaPoC '15. Bordeaux, France: ACM, 2015, 6:1–6:4. ISBN: 978-1-4503-3537-9. DOI: 10.1145/2745947.2745953. URL: http://doi.acm.org/10.1145/2745947.2745953.

[9] D. Navalho, S. Duarte, and N. Preguiça. "A Study of CRDTs That Do Computations". In: *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data*. PaPoC '15. Bordeaux, France: ACM, 2015, 1:1–1:4. ISBN: 978-1-4503-3537-9. DOI: 10.1145/2745947.2745948. URL: http://doi.acm.org/10.1145/2745947.2745948.

[10] H. Attiya, F. Ellen, and A. Morrison. "Limitations of Highly-Available Eventually-Consistent Data Stores". In: *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*. PODC '15. Donostia-San Sebastián, Spain: ACM, 2015, pp. 385–394. ISBN: 978-1-4503-3617-8. DOI: 10.1145/2767386.2767419. URL: http://doi.acm.org/10.1145/2767386.2767419.

[11] P. Almeida, A. Shoker, and C. Baquero. "Efficient State-Based CRDTs by Delta-Mutation". In: *Networked Systems*. Ed. by A. Bouajjani and H. Fauconnier. Vol. 9466. Lecture Notes in Computer Science. Springer International Publishing, 2015, pp. 62–76. DOI: 10.1007/978-3-319-26850-7_5. URL: http://dx.doi.org/10.1007/978-3-319-26850-7_5.

[12] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications". In: *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM '01. San Diego, California, USA: ACM, 2001, pp. 149–160. ISBN: 1-58113-411-8. DOI: 10.1145/383059.383071. URL: http://doi.acm.org/10.1145/383059.383071.

[13] 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.

[14] 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: *IEEE 36th International Conference on Distributed Computing Systems*. ICDCS'16. Nara, Japan: IEEE Computer Society, June 2016.

[15] L. Lamport. "The Part-time Parliament". In: *ACM Trans. Comput. Syst.* 16.2 (May 1998), pp. 133–169. ISSN: 0734-2071. DOI: 10.1145/279227.279229. URL: http://doi.acm.org/10.1145/279227.279229.

[16] D. Ongaro and J. Ousterhout. "In Search of an Understandable Consensus Algorithm". In: *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*. USENIX ATC'14. Philadelphia, PA: USENIX Association, 2014, pp. 305–320. ISBN: 978-1-931971-10-2. URL: http://dl.acm.org/citation.cfm?id=2643634.2643666.

[17] 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.

[18] Lightbend, Inc. *Akka Distributed Data*. Accessed: 2016-06-08. URL: http://doc.akka.io/docs/akka/2.4.7/scala/distributed-data.html.

[19] Lightbend, Inc. *Akka Cluster*. Accessed: 2016-06-08. URL: http://doc.akka.io/docs/akka/2.4.7/common/cluster.html.

[20] SyncFree. *AntidoteDB*. Accessed: 2016-12-08. URL: http://http://antidote-db.com/.

[21] 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.

[22] *Redis*. Accessed: 2016-05-03. URL: http://redis.io.

[23] P. S. Almeida, C. Baquero, R. Gonçalves, N. Preguiça, and V. Fonte. "Scalable and Accurate Causality Tracking for Eventually Consistent Stores". In: *Proceedings of the Distributed Applications and Interoperable Systems, held as part of the Ninth International Federated Conference on Distributed Computing Techniques*. Berlin, Germany, 2014, pp. 67–81.

[24] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. "The Hadoop Distributed File System". In: *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. MSST '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10. ISBN: 978-1-4244-7152-2. DOI: 10.1109/MSST.2010.5496972. URL: http://dx.doi.org/10.1109/MSST.2010.5496972.

[25] J. Kreps, N. Narkhede, J. Rao, et al. "Kafka: A distributed messaging system for log processing". In: NetDB. 2011.

[26] Apache Software Foundation. *Apache Hadoop*. Accessed: 2016-05-18. URL: http://hadoop.apache.org/.

[27] S. Ghemawat, H. Gobioff, and S.-T. Leung. "The Google File System". In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. SOSP '03. Bolton Landing, NY, USA: ACM, 2003, pp. 29–43. ISBN: 1-58113-757-5. DOI: 10.1145/945445.945450. URL: http://doi.acm.org/10.1145/945445.945450.

[28] J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. URL: http://doi.acm.org/10.1145/1327452.1327492.

[29] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. "Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing". In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. NSDI'12. San Jose, CA: USENIX Association, 2012, pp. 2–2. URL: http://dl.acm.org/citation.cfm?id=2228298.2228301.

[30] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. "Discretized Streams: Fault-tolerant Streaming Computation at Scale". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: ACM, 2013, pp. 423–438. ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522737. URL: http://doi.acm.org/10.1145/2517349.2522737.

[31] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy. "Storm@Twitter". In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD '14. Snowbird, Utah, USA: ACM, 2014, pp. 147–156. ISBN: 978-1-4503-2376-5. DOI: 10.1145/2588555.2595641. URL: http://doi.acm.org/10.1145/2588555.2595641.

[32] D. Peng and F. Dabek. "Large-scale Incremental Processing Using Distributed Transactions and Notifications". In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. OSDI'10. Vancouver, BC, Canada: USENIX Association, 2010, pp. 251–264. URL: http://dl.acm.org/citation.cfm?id=1924943.1924961.

[33] D. Navalho, S. Duarte, N. Preguiça, and M. Shapiro. "Incremental Stream Processing Using Computational Conflict-free Replicated Data Types". In: *Proceedings of the 3rd International Workshop on Cloud Data and Platforms*. CloudDP '13. Prague, Czech Republic: ACM, 2013, pp. 31–36. ISBN: 978-1-4503-2075-7. DOI: 10.1145/2460756.2460762. URL: http://doi.acm.org/10.1145/2460756.2460762.

[34] C. Meiklejohn and P. Van Roy. "Lasp: A Language for Distributed, Coordination-free Programming". In: *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*. PPDP '15. Siena, Italy: ACM, 2015,

pp. 184–195. ISBN: 978-1-4503-3516-4. DOI: 10.1145/2790449.2790525. URL: http://doi.acm.org/10.1145/2790449.2790525.

[35] L. Kuper and R. R. Newton. "LVars: Lattice-based Data Structures for Deterministic Parallelism". In: *Proceedings of the 2Nd ACM SIGPLAN Workshop on Functional High-performance Computing*. FHPC '13. Boston, Massachusetts, USA: ACM, 2013, pp. 71–84. ISBN: 978-1-4503-2381-9. DOI: 10.1145/2502323.2502326. URL: http://doi.acm.org/10.1145/2502323.2502326.

[36] 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.

[37] W. Vogels. "Eventually Consistent". In: *Commun. ACM* 52.1 (Jan. 2009), pp. 40–44. ISSN: 0001-0782. DOI: 10.1145/1435417.1435432. URL: http://doi.acm.org/10.1145/1435417.1435432.

[38] 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.

[39] P. E. O'Neil. "The Escrow Transactional Method". In: *ACM Trans. Database Syst.* 11.4 (Dec. 1986), pp. 405–430. ISSN: 0362-5915. DOI: 10.1145/7239.7265. URL: http://doi.acm.org/10.1145/7239.7265.

[40] Basho Technologies, Inc. *Riak Core*. Accessed: 2017-01-25. URL: https://github.com/basho/riak_core.

[41] 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.

[42] iMatix, Inc. *ZeroMQ*. Accessed: 2017-01-25. URL: http://zeromq.org/.

[43] Gonçalo Cabrita. *AntidoteDB C-CRDT Module*. Accessed: 2017-01-25. URL: https://github.com/gmcabrita/antidote_ccrdt.

[44] SyncFree Consortium. *AntidoteDB CRDT Module*. Accessed: 2017-01-25. URL: https://github.com/syncFree/antidote_crdt.

[45] Basho Technologies, Inc. *Basho Bench*. Accessed: 2017-01-28. URL: https://github.com/basho/basho_bench.

Appendix 1: Example NuCRDT implementation

Listing A.1: Average NuCRDT implementation in Erlang

```
% A NuCRDT that computes the aggregated average.

-module(antidote_ccrdt_average).
-behaviour(antidote_ccrdt).
-include("antidote_ccrdt.hrl").

-export([
    new/0,
    new/2,
    value/1,
    downstream/2,
    update/2,
    equal/2,
    to_binary/1,
    from_binary/1,
    is_operation/1,
    is_replicate_tagged/1,
    can_compact/2,
    compact_ops/2,
    require_state_downstream/1
]).

-type sum() :: non_neg_integer().
-type num() :: non_neg_integer().

-type average() :: {sum(), num()}.
-type prepare_update() :: {add, sum()} | {add, average()}.
```

```
28 -type effect_update() :: {add, average()}.
29
30 %% Creates a new `average()`.
31 -spec new() -> average().
32 new() ->
33     {0, 0}.
34
35 %% Creates a new `average()` with the given `Sum` and `Num`.
36 -spec new(sum(), num()) -> average().
37 new(Sum, Num) when is_integer(Sum), is_integer(Num) ->
38     {Sum, Num};
39 new(_, _) ->
40     new().
41
42 %% Returns the value of `average()`.
43 -spec value(average()) -> float().
44 value({Sum, Num}) when is_integer(Sum), is_integer(Num) ->
45     Sum / Num.
46
47 %% Generates an `effect_update()` operation from a `prepare_update()`.
48 %%
49 %% The supported `prepare_update()` operations for this data type are:
50 %% - `{add, sum()}`
51 %% - `{add, average()}`
52 -spec downstream(prepare_update(), average()) -> {ok, effect_update()}.
53 downstream({add, {Value, N}}, _) ->
54     {ok, {add, {Value, N}}};
55 downstream({add, Value}, _) ->
56     {ok, {add, {Value, 1}}}.
57
58 %% Executes an `effect_update()` operation and returns the resulting state.
59 %%
60 %% The executable `effect_update()` operations for this data type are:
61 %% - `{add, sum()}`
62 %% - `{add, average()}`
63 -spec update(effect_update(), average()) -> {ok, average()}.
64 update({add, {_, 0}}, Average) ->
65     {ok, Average};
66 update({add, {Value, N}}, Average) when is_integer(Value),
67                                         is_integer(N), N > 0 ->
68     {ok, add(Value, N, Average)};
69 update({add, Value}, Average) when is_integer(Value) ->
70     {ok, add(Value, 1, Average)}.
71
72 %% Compares the two given `average()` states.
73 -spec equal(average(), average()) -> boolean().
74 equal({Value1, N1}, {Value2, N2}) ->
75     Value1 =:= Value2 andalso N1 =:= N2.
76
77 %% Converts the given `average()` state into an Erlang `binary()`.
```

```
78 -spec to_binary(average()) -> binary().
79 to_binary(Average) ->
80     term_to_binary(Average).
81
82 %% Converts a given Erlang `binary()` into an `average()`.
83 -spec from_binary(binary()) -> {ok, average()}.
84 from_binary(Bin) ->
85     {ok, binary_to_term(Bin)}.
86
87 %% Checks if the given `prepare_update()` is supported by the `average()`.
88 -spec is_operation(any()) -> boolean().
89 is_operation({add, {Value, N}}) when is_integer(Value), is_integer(N) -> true;
90 is_operation({add, Value}) when is_integer(Value) -> true;
91 is_operation(_) -> false.
92
93 %% Checks if the given `effect_update()` is tagged for replication.
94 -spec is_replicate_tagged(effect_update()) -> boolean().
95 is_replicate_tagged(_) -> false.
96
97 %% Checks if the given `effect_update()` operations can be compacted.
98 -spec can_compact(effect_update(), effect_update()) -> boolean().
99 can_compact({add, {_, _}}, {add, {_, _}}) -> true.
100
101 %% Compacts the given `effect_update()` operations.
102 -spec compact_ops(effect_update(), effect_update()) -> {{noop},
103                                                          effect_update()}.
104 compact_ops({add, {V1, N1}}, {add, {V2, N2}}) ->
105     {{noop}, {add, {V1 + V2, N1 + N2}}}.
106
107 %% Checks if the data type needs to know its current state to generate
108 %% `update_effect()` operations.
109 -spec require_state_downstream(any()) -> boolean().
110 require_state_downstream(_) -> false.
111
112 %%%% Private
113
114 %% Adds `sum()` and `num()` to the current `average()`.
115 -spec add(sum(), num(), average()) -> average().
116 add(Value, N, {CurrentValue, CurrentN}) ->
117     {CurrentValue + Value, CurrentN + N}.
```