UiT The Arctic University of Norway

Faculty of Science and Technology
Department of Computer Science

Fault-Tolerant Distributed Declarative Programs

Moritz Jörg

Master's thesis in Computer Science INF-3981 — June 2024

Supervisors

Main supervisor:	Weihai Yu	UiT The Arctic University of Norway, Faculty of Science and Technology, Department of Computer Science

“A distributed system is one in which the failure of a computer you didn’t even know existed can render your own computer unusable.”
–Leslie Lamport

“One of my most productive days was throwing away 1000 lines of code.”
–Ken Thompson

“The problem with object-oriented languages is they’ve got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle.”
–Joe Armstrong

Abstract

In our increasingly interconnected digital landscape, the constant generation and consumption of data on various computing devices present challenges for ensuring constant accessibility, particularly in intermittent network scenarios. The emerging focus on distributed systems is aimed at not only managing substantial data volumes but also guaranteeing storage on devices for low latency and high availability. A paradigm known as *local-first software* prioritize the storage of data on end-user devices as opposed to relying solely on centralized cloud services.

The intersection of Conflict-free Replicated Data Type (CRDT)s and Datalog, exemplified by Consistency as Logical Monotonicity (CALM), establishes that monotonic logic programs guarantee eventual consistency without the need for coordination. This synergy enables robust reasoning about data consistency and parallelism, paving the way for the Partitioned and Replicated Asynchronous Datalog (PRAD) runtime. Transforming sequential Datalog programs into distributed one, PRAD ensures that the distributed program meets the specified availability, parallel, and fault-tolerance requirements. To achieve this, PRAD augments Datalog programs with semiring data provenance and equips the provenance expressions with a CRDT.

One limitation of the current PRAD runtime is that it lacks a recovery mechanism. In the event that a site going offline or crashing, the data on that site is lost and the system's fault-tolerance is compromised. To address this issue, a repair mechanism can be implemented to restore or replicate the lost site. This comes with the added benefit of increasing the systems fault-tolerance and availability.

The main contribution of this thesis is the development of a novel approach to repairing a PRAD program at runtime. The repair mechanism is designed to restore an offline site by leveraging our Lightweight Commit (LWC)s with the help of the Causal Length Set (Cl-Set) CRDT. It is draws inspiration from the Git and Pijul distributed version control systems, and applying their principles to the PRAD runtime. The repair mechanism is designed to be lightweight and efficient, ensuring that the system can repair failures without compromising performance. Our approach differs from previous work in that we do not rely on vector clocks or sequence numbers. Instead, we utilize the Cl-Set CRDT to accommodate messages that may be delivered out of order or are duplicated.

The approach is evaluated through a series of experiments, in which the performance of the repair mechanism is compared to that of the existing PRAD runtime and multiple alternative approaches. The results demonstrate that the repair mechanism is both efficient and lightweight, and that it can restore a site in a reasonable amount of time, thereby confirming the viability of our approach for edge devices.

Acknowledgements

I'd like to thank my supervisor, Weihai Yu, for his guidance and support throughout this process. I'm grateful to you for your enthusiasm and encouragement. Without it, I might have chosen a less daunting topic for my thesis. Thank you for giving me the chance to work on this thesis.

I want to thank my classmates for their friendship and support throughout the process. We have had many discussions about the thesis, and your feedback has been invaluable. Special thanks to Marius, Sondre and Øyvind for always asking questions and challenging my assumptions. Thanks to my coworkers for providing much-needed distractions and support.

Finally, I want to thank my family for their unwavering support and encouragement. I couldn't have done it without you. Thank you for always being there for me.

Contents

List of Figures

List of Tables

List of Listings

List of Definitions

List of Abbreviations

API	Application Programming Interface
BEAM	Bogdan/Björn’s Erlang Abstract Machine
CALM	Consistency as Logical Monotonicity
CAP	Consistency, Availability, Partition Tolerance
CD	Continuous Deployment
CI	Continuous Integration
CRDT	Conflict-free Replicated Data Type
DevOps	Development and Operations
EDB	Extensional Database
ERTS	Erlang Run-Time System
ETS	Erlang Term Storage
GLB	Greatest Lower Bound
IDB	Intensional Database
JSON	JavaScript Object Notation
LUB	Least Upper Bound
LWC	Lightweight Commit
ODS	Open Distributed Systems

OTP	Open Telecom Platform
PRAD	Partitioned and Replicated Asynchronous Datalog
SEC	Strong Eventual Consistency
SQL	Structured Query Language
TDD	Test-Driven Development
UiT	University of Tromsø – The Arctic University of Norway
VCS	Version Control System

1 Introduction

We continuously generate and consume a vast amount of data on our computing devices, which can be both online or offline at times. However, certain applications require the data to be accessible at all times, even when the devices are intermittently disconnected from the network. This has led to an increased focus on designing distributed systems that not only handle large volumes of data, but also provide *low latency* and *high availability* in the face of network partitions and intermittent connectivity. With the ability to recover at runtime without interruption ensures minimal loss of performance, correctness, and availability.

Local-first software [1] is a method of designing software that prioritizes storing data on end-user devices over relying solely on centralized cloud-based services. The primary objective is to guarantee offline functionality, and seamless synchronization upon reconnection, even in the event of network failures. This leads us to a fundamental problem in distributed systems, known as the Consistency, Availability, Partition Tolerance (CAP) theorem [2].

The CAP theorem establishes that a distributed system is unable to simultaneously guarantee the three desired properties: consistency, availability, and partition tolerance. Consistency is defined as the condition where all nodes observe the same data at the same time. Availability signifies a guarantee that every request receives a response indicating whether it was successful or unsuccessful. In contrast, partition tolerance implies that the system continues to operate despite arbitrary message loss or failure of a portion of the system.

In adopting a local-first approach, we are prepared to forego strong consistency in favor of availability and partition tolerance. Conversely, a weak or eventual consistency model may be more appropriate in certain circumstances.

To achieve eventual consistency a well-established technique is to use Conflict-free Replicated Data Type (CRDT). CRDT's are data structures that allow multiple replicas to be updated and merged without coordination between the replicas. This ensures that the replicas converge towards a common state, when the same set of updates has been applied (*strong eventual consistency*). These properties guarantee the availability and eventual consistency of data across a variety of devices.

The management of data is typically carried out by databases' management systems, which are accessed through declarative data manipulation and query languages such as Structured Query Language (SQL) and Datalog. The connection between CRDT and Datalog is exemplified by the concept of Consistency as Logical Monotonicity (CALM) [3]. This states that monotonic logic programs are guaranteed to be eventually consistent, without the need for coordination between nodes. This enables the use Datalog as a query language for CRDT's, and the reasoning about the consistency of the data.

Elixir is a developing programming language on top of Erlang for systems due to its inherent support for concurrency and fault-tolerance. In the industry, companies such as Discord [4] and WhatsApp [5] have adopted the underlying Erlang virtual machine to build highly scalable systems with millions of users. The lightweight processes and actor model can obviate the necessity for complex locking mechanisms, with message passing serving as the principal means of communication between processes.

This makes Elixir an ideal candidate for the exploration of fault-tolerant distributed systems. It is hypothesized that the extension of Elixir with Partitioned and Replicated Asynchronous Datalog (PRAD) will provide a powerful tool for the construction of resilient and fault-tolerant Datalog programs.

1.1 Problem definition

There is currently work being done at the Open Distributed Systems (ODS) group at the University of Tromsø – The Arctic University of Norway (UiT) on PRAD [6], an extension to Datalog, which can turn a conventional Datalog program into a distributed one, thereby enabling the data to be partitioned and replicated across multiple sites. The data are always accessible at the devices, even when they are offline. The programs are eventually consistent. In other words, when the devices are connected, the results generated by the distributed program, will eventually be equivalent to those produced by a non-distributed Datalog program.

One of the key design goals of PRAD is to ensure fault tolerance through replication. In the event that a replica is unavailable, a PRAD program may continue to operate, albeit with a reduced replication degree and a diminished capacity to withstand site-level failures. The current implementation of PRAD lacks a recovery mechanism, which means that in the event of site crash, the data on that site is lost. A potential recovery mechanism should retain minimal metadata or have a method of garbage collection to prevent the data from growing indefinitely.

We focus on extending the PRAD runtime fault-tolerance with a recovery mechanism that allows sites to recover from failures. The primary goal of this thesis is to achieve the following stated goal:

> *The thesis aims to investigate the feasibility of repairing PRAD programs at runtime to maintain the fault-tolerance requirements.*

1.2 Scope, limitations and assumptions

This thesis focuses on the repair mechanism for the PRAD runtime, specifically the restoration of failed sites. The sites are not terminated abruptly, instead they are programmatically set to a failed state. However, the sites are treated if they have crashed, and are unreachable. We try to reflect on the possible implications of our approach, and the potential limitations of our implementation.

We perform a set of simple experiments aimed at evaluating the viability and the performance of the proposed repair mechanism. The experiments are not exhaustive, and it should be noted that the results are not generalizable to all scenarios. The repair mechanism is only evaluated on insertions, but it could be adapted to accommodate deletions as well.

1.3 Contributions

This thesis is a contribution to the ongoing unpublished work at the Open Distributed Systems (ODS) group on the PRAD runtime. With the design and implementation of a repair mechanism for the PRAD runtime, we aim to provide a fault-tolerant extension for the PRAD runtime. Section 3.1 is written with as part of unpublished work and its inclusion is intended to provide context for the reader.

The primary contributions of this thesis are as follows:

- An implementation of a repair mechanism for the PRAD runtime, that allows sites to restore from failures using replication.
- An implementation of site restoration from upstream or peer sites, utilizing the Causal Length Set CRDT and Lightweight Commit (LWC)'s to more efficiently restore sites.
- Support for both 'project' and 'select' operations in the repair mechanism, to ensure that the data is eventually consistent across sites.
- A Delta Trimming mechanism for the PRAD runtime, that ensures that metadata is not growing indefinitely.

1.4 Context

This thesis was completed within the context of the ODS group at University of Tromsø – The Arctic University of Norway (UiT). The Open Distributed Systems (ODS) group facilitates research in the field of distributed applications of various kinds, with an emphasis on interoperability and adaptability issues. The group's research agenda includes support for next-generation applications, mobility, composition-based web applications, real-time collaboration and information exchange. Specific issues include the adaptability, context-awareness, applied security, privacy, and collaborative editing of these applications.

A recent publication from the ODS group related to CRDT's is 'Toward Replicated and Asynchronous Data Streams for Edge-Cloud Applications' by Qayyum and Yu [6]. This paper presents a novel approach to the design of a replicated and asynchronous data stream system for edge-cloud applications. The system is based on the principles of CRDT's, and presents some interesting challenges in terms of fault-tolerance and recovery for the future.

1.5 Methodology

In their article 'A Framework for the Discipline of Computer Science' Comer et al. [7] present an intellectual framework for the field of computer science. This was subsequently endorsed and approved by the ACM Education Board.

> *Computer science and engineering is the systematic study of algorithmic processes their theory, analysis, design, efficiency, implementation and application that describe and transform information.*

The final report identifies three distinct paradigms of computer science research: theory, abstraction, and design and implementation.

- The **theoretical** paradigm is the examination of the qualities of algorithms and data structures. It is a highly abstract mathematical discipline. Theory is focused on demonstrating the accuracy of algorithms and assessing their time and space complexity. The initial step is to characterize the objects of study by *defining* them. Subsequently, hypotheses are formed about possible relationships between objects with *theorems*. Subsequently, these hypotheses are then subjected to rigorous testing through *proofs* to determine whether the relationships are indeed true. Finally, the results are interpreted.

- The **abstraction** (modeling) paradigm is grounded in the experimental scientific method, which proposes that scientific advancement develops by forming hypotheses, creating models and predictions, and subsequently assessing them through experimentation. Finally, the data is collected and analyzed to determine the validity of the predictions made were correct.

- The **design** paradigm is based on the principle of abstraction, and is concerned with the development of systems that are capable of solving real-world problems. The focus of this paradigm is on the creation of requirements and specifications for a given problem, and evaluating the design through experimentation. In an iterative process, the design is refined and improved until the requirements are met, at which point a specification is produced.

The three paradigms are not mutually exclusive and frequently overlap and are employed in conjunction with one another. The intertwined nature of the paradigms makes it challenging to distinguish between them and identify a single paradigm as the primary driver of a specific research project.

This thesis primarily employs the design paradigm, with some overlap into the theoretical paradigm. The challenge is to design a repair mechanism for the PRAD runtime that is both efficient and fault-tolerant. We state the requirements for the approach, and employ an iterative process to refine the design until the requirements are met as closely as possible.

1.6 Thesis Outline

The remainder of this thesis is organized as follows:

Chapter 2: Theoretical Framework provides the necessary background for understanding the rest of the design and implementation. It covers the domains of CRDT, Consistency, Datalog, Provenance, and Fault-Tolerance.

Chapter 3: Design outlines the design of the repair mechanism and the accompanying trimming algorithm. It also provides an overview of the PRAD runtime architecture.

Chapter 4: Software Engineering Methods describes the methods and tools used to explore the problem. It covers the topics of agile and test-driven development.

Chapter 5: Implementation describes the challenges and solutions encountered during the implementation of the repair mechanism and trimming algorithm.

Chapter 6: Experiments investigates the solutions proposed in the previous chapter, and evaluates them through experimentation.

Chapter 7: Discussion & Future Work discusses the results of the evaluation. It also discusses the challenges of recovering from failures in a distributed system, and the implications of the results. It also outlines potential future work and improvements to the system.

Chapter 8: Conclusion concludes the report by summarizing the results and contributions of the thesis. It also compares the approach to existing work.

2 Theoretical Framework

This chapter presents the theoretical framework necessary to comprehend the inner workings of PRAD. We will begin by introducing the fundamental algebraic structures utilized in the thesis, namely *p-semirings* and *join-semilattices*. Subsequently, the concepts of *local-first software* and the *consistency* guarantees it can provide will be explained. Moreover, the fundamental principles of Conflict-free Replicated Data Type (CRDT)s and an optimization for state-based CRDTs, known as delta-state CRDTs, are explained. This is followed by an introduction to the concepts and terminology of Datalog, and finally, the notion of *fault-tolerance* in distributed systems.

2.1 Algebraic Structures

We will now proceed to introduce and provide definitions for the algebraic structures that serves as the foundation of PRAD.

2.1.1 Lattice

Definition 1. A *poset* P is a set together with a binary relation $\leq$ that satisfies, for all $a, b, c \in P$: (i) $a \leq a$ (reflexivity), (ii) if $a \leq b$ and $b \leq a$ then $a = b$ (antisymmetry), (iii) if $a \leq b$ and $b \leq c$ then $a \leq c$ (transitivity).

Here the relation $\leq$ is called the *partial order*. Further a poset $(P, \leq)$ is called a *lattice* [8] if every subset of P has a Least Upper Bound (LUB) and a Greatest Lower Bound (GLB) with respect to $\leq$.

Definition 2. A *join-semilattice* is a partially ordered set where every subset has a LUB [9]

Further a *join* is defined as the Least Upper Bound (LUB) and written as $\sqcup$, while a *meet* is defined as the Greatest Lower Bound (GLB) and written as $\sqcap$.

It follows that $\sqcup$ is:

- $x \sqcup y = y \sqcup x$ (commutativity)
- $x \sqcup (y \sqcup z) = (x \sqcup y) \sqcup z$ (associativity)
- $x \sqcup x = x$ (idempotence)

Definition 3. A *lattice* is *distributive* if the operations $\sqcup$ and $\sqcap$ distribute over each other. That is: $x \sqcap (y \sqcup z) = x \sqcup y \sqcap x \sqcup z$.

2.1.2 Join Decomposition

The subsequent step is to define a *join-irreducible state*

Definition 4. A lattice state $x \in \mathcal{L}$ is join-irreducible if it cannot be written as the join of any finite set of states, except for itself. In other words an elementary state. Written as $\mathcal{J}(\mathcal{L})$

With this we can define the join decomposition of a state.

Definition 5. A join decomposition of a lattice state $x \in \mathcal{L}$ is a set of join-irreducible states D such that $D \subseteq \mathcal{J}(\mathcal{L})$.

Adding to this, we define a *monotonic* join semilattice.

Definition 6. A monotonic join semilattice is a join semilattice that has the following properties: (i) Merging two states computes the LUB of both states i.e. $s \cdot m(s') = s \sqcup s'$. (ii) States is *inflationary* across updates. $s \leq s \cdot u$.

Finally, assuming eventual delivery and termination, any object that satisfies the monotonic join semilattice property has strong eventual consistency [10].

2.1.3 Provenance Semirings

A *semiring* is an algebraic structure, denoted by the set $\langle S, +, \cdot, \mathbf{0}, \mathbf{1} \rangle$, where S is a set with two binary operators $+$ and $\cdot$, and two elements $\mathbf{0}$ and $\mathbf{1}$ in S. This structure must satisfy the following properties:

1. $(S, +)$ is a monoid with identity element $\mathbf{0}$.
2. $(S, \cdot)$ is a monoid with identity element $\mathbf{1}$.
3. Addition is *commutative*.
4. $\cdot$ left- and right-distributes over $+$.
5. 0 is the annihilator of $\cdot$.

Moreover, a *semiring* is a *commutative-semiring* [11] if for all $a, b \in S$, $a \cdot b = b \cdot a$.

In order to ensure provenance, it is necessary to annotate the facts in a database. This allows the derivations of facts by values of a *commutative semiring* to be tracked. These values get propagated through a query, using the properties of the semiring to keep track of information as follows:

- $+$ interprets as *union* of provenance information.
- $\cdot$ interprets as *joins* of provenance information.
- $\mathbf{0} \in S$ are false assertions and $\mathbf{1} \in S$ are true assertions.

In the context of provenance, our interest lies in the *provenance semiring* [6], which is a commutative semiring with additional properties:

- $+$ and $\cdot$ are idempotent.
- $+$ applies to the empty and infinite sets.

A problem with algebraic semiring operations is that they only offer incomplete support for negation in Datalog. With the *p-semiring* PRAD can express the minimum representation of dependencies of facts in the database.

2.2 Local-first Software

The concept of *Local-first software* represents a significant paradigm shift in the way we think about software. Today the majority of software and services operate within cloud, with data stored on centralized servers. This results in the loss of ownership and agency for users, who become dependent on the service provider. There are no copies of the data stored locally, and the users are at the mercy of the service provider to keep their data safe and available.

In contrast to traditional software architectures, local-first software, prioritizes data storage on the end-user devices. This approach involves minimal replication to the cloud, which is considered a secondary concern. Such software is identified by a set of ideals, which are as follows: *Low latency*, *Multi-device*, *High availability*, *Secure* and *Privacy-preserving*. As previously discussed by Kleppmann et al. [1], CRDT are the fundamental multi-user data structures for realizing local-first software. They allow for data to be replicated across multiple devices, without the need for coordination between the replicas and with the guarantee of eventual consistency.

2.3 Consistency

This section investigates the concept of consistency in the context of replication and partitioning in distributed systems. If data objects A and B are replicated on multiple machines, they are considered consistent if all observers see the same value for A and B. This *strong consistency* can also be understood as *linearizability* or *serializability*, which is traditionally solved by either using consensus algorithms such as Paxos [12] i or the Two-Phase Commit Protocol [13]. However, both of these approaches entail a trade-off in terms of coordination between the replicas, which can significantly impede the system's performance, as evidence by the findings of Hellerstein and Alvaro [14].

A weaker form of consistency, such as *eventual consistency*, allows for replicas to diverge temporarily. Updates are propagated asynchronously between the replicas, and when no new updates are submitted, all replicas will eventually converge to the same value. However, since these updates can be made concurrently, conflicts arise and a conflict resolution mechanism is needed.

Definition 7. A distributed system is eventually consistent if it satisfies the following properties: *Eventual delivery*: If one replica has received an update, then it will be eventually delivered to all replicas. *Convergence*: Replicas that have received the same updates will eventually reach an identical state. *Termination*: All method executions terminate.

If we instead opt for a compromise between strong and eventual consistency, we can use Strong Eventual Consistency (SEC). This guarantees that when two replicas have received the same set of update, they will have the identical state, and any conflicts will be resolved *coordination-free*.

Definition 8. A distributed system has strong eventual consistency[15] if it is eventually consistent, and if all replicas have received the same updates, they will have the same state (Strong convergence).

A coordination-free system enables scalability and performance that are not possible with traditional consistency mechanisms, which slow down computation [16]. As identified by Hellerstein [17], monotonicity is a key property of coordination-free systems to establish consistency. This property is covered in the CALM theorem:

Definition 9. A program has a consistent, coordination-free distributed implementation if and only if it is monotonic.

It originates as a formulation for consistency in distributed logic programs, but can be applied to CRDTs to provide efficient and safe query execution [3].

2.4 CRDT

A Conflict-free Replicated Data Type (CRDT) is a data abstraction designed to be replicated across multiple replicas. Each replica is updated and queried locally, without the need for coordination between the replicas. Updates are asynchronously gossiped between the replicas and merged when received. However, replicas can temporarily diverge. CRDTs guarantee strong eventual consistency, as defined in 8, ensuring that replicas converge deterministically and coordination-free to the same state when the same set of updates have been applied.

Two distinct types of CRDTs can be identified: *state-based* (Convergent) and *operation-based* (Commutative) CRDTs. In the following sections, we will provide explanations for both types. Some parts of our explanations are based on the work by Shapiro et al. [18].

2.4.1 Operation-based CRDT

In operation-based CRDTs, convergence is achieved through the propagation of update operations between the replicas. This process results in the communication of only the most recent operations between replicas. When a replica receives an update operation, it is applied reliably to the local state. In addition to requiring the update operations to be reliably delivered, in certain cases they must also be in a specific order to ensure convergence. This order is typically a causal order.

In order for the replicas to converge in cases without a causal order, all update operations must be *commutative*. If the update operations may be applied in any order, then all update operations must be *commutative* and *idempotent*. To guarantee that the update operations are delivered in a consistent order to all replicas, a reliable multicast protocol is needed. This is typically implemented using the Reliable Broadcast protocol [19], which enables replicas to exchange messages in a causal order.

2.4.2 State-based CRDT

In a state-based CRDT replicas synchronize by sending their entire state to the other replicas, rather than the update operations performed. When a replica receives the state of another replica, it merges the received state with its own. The possible states of the CRDT can be represented as a join-semilattice as defined in 2. The merge function of two states s_1 and s_2 is defined as the LUB of the two states, and written as $s_1 \sqcup s_2$. Consequently, the merge function must be commutative, associative and idempotent. Moreover, the state must be increasing monotonic, which is necessary to ensure that the state is always moving towards the LUB of all states. This is defined as a monotonic join-semilattice in 6. With these properties the replicas will converge as long as they have received the same updates.

One disadvantage of state-based CRDTs are that they require the entire state to be transmitted between the replicas, even if only a small part was updated and replicas already hold most of the information. Which is especially true for large data structures such as graphs and JSON objects. Consequently, state-based CRDTs are usually used for file systems and databases, where updates are less frequent and updates are larger. For smaller and more frequent updates, operation-based CRDTs are more suitable. Conversely, state-based CRDTs work trivially with dynamic systems, where the set of replicas can change over time. While operation-based CRDTs require complex mechanism, and are frequently limited to predefined set of participants (which is compatible with Causal Broadcast).

2.4.3 Delta-state CRDT

As previously stated, state-based CRDTs require the entire state to be sent between the replicas. In particular, for large data structures, and given the ever-increasing state of CRDTs, this can give rise to issues with regard to performance and scalability.

One potential optimization to improve the efficiency of state-based CRDTs is to only send the difference between the states, instead of the entire state. This is called a *delta-state CRDT*. There are currently two main approaches to implementing delta-state CRDTs. Firstly, the join-semilattice state can be handled as the join of multiple smaller delta-states. These are generated by delta-mutators [20], which encode the difference in state since the last synchronization. Provided that all deltas are propagated and joined at the replicas, they will converge to the same state. This approach is employed in the Akka framework [1].

An alternative approach involves the utilization of join-decompositions [21]. In this approach, the deltas are referred to as join-irreducible states, as defined in 4. Every element in a join-semilattice can be decomposed into a set of join-irreducible states. Consequently, any state can be represented as the join of a set of join-irreducible states. This further reduces the redundancies in the state, as the join-irreducible states are the smallest possible states. However, this approach also introduces an overhead in terms of computation of the aforementioned join-irreducible states.

PRAD adopts delta-state CRDTs, which are more efficient and scalable than state-based CRDTs. Moreover, they are more suitable for dynamic systems, where the set of replicas can change over time, which is a must for a distributed system.

Subsequently, we will present a selection of the most common CRDTs used in practice. These are all state-based CRDTs, which can be composed to create more complex data structures such as JSON objects and graphs.

2.4.4 Grow-Only Counter

A typical example of a trivial state CRDTs is the *grow-only counter*. The replicated counter permits only monotonically increasing increments. The state of each replica can temporarily diverge, but will converge to the same state when the state is merged. Similar to vector clocks, a grow-only counter is

1. https://doc.akka.io/docs/akka/current/typed/distributed-data.html#delta-crdt

simply a map of a unique replica identifier and a partial counter value. Each replica maintains the number of increments it has performed locally and stores it as a map entry. The value of the counter is then the sum of all the partial counter values. The merge function of two replicas is defined as the maximum of the two partial counter values for each replica identifier. This ensures that the counter will converge to the same value, as long as the same number of increments have been performed on both replicas. As some readers may have observed, this approach is similar to *Vector Clocks* [22], which are used to track causality between events in distributed systems.

Listing 2.1 shows a simple implementation of a grow-only counter in Elixir.

Listing 2.1: Example implementation of a Grow-Only counter.

```
defmodule GCounter do
  def new, do: %{}

  def increment(c, id, v \\ 1) do
	Map.update(c, id, v, &(&1 + v))
  end

  def merge(c1, c2) do
	Map.merge(c1, c2, fn _id, v1, v2 -> max(v1,
	   v2) end)
  end

  def value(c), do: Map.values(c) |> Enum.sum()
end
```

2.4.5 LWW-Register

A *register* maintains a single value, and can be updated by any replica by writing a new value to it. One of the most common registers is the *Last-Writer-Wins* register CRDT. As the name implies, the register gives precedence to the most recent write operation, which also is the value it stores. To guarantee this precedence, the register holds a timestamp for the latest write operation. It is assumed that these timestamps are unique, totally ordered and generated in an increasing monotonically order. The merge function of two replicas is defined as the greater of the two timestamps, and the value of the replica with the greatest timestamp, according to the defined total order. Ties between replicas timestamps are resolved by comparing the replica identifiers, such as peer IDs. Listing 2.2 shows a straightforward implementation of a LWW-register in Elixir.

Listing 2.2: Example implementation of a LWW register.

```
defmodule LWWReg do

  defstruct v: nil, t: nil

  def new(v, t), do: %__module__{v: v, t: t}

  def merge(r1, r2), do: update(r1, r2.t, r2.v)

  def update(r, v, t) do
    cond do
          r.t == t && r.v != v -> raise "err"
          r.t <= t -> new(r.v, r.t)
          r.t > t -> r
        end
  end

  def get(r), do: r.v
end
```

2.4.6 Grow-Only Set

Sets are among of the most common data structures. A *set* is a collection of unique elements, which can be mutated by adding or removing elements. However, the two operations are not commutative, and we would need a causal order between them to ensure convergence. The most straightforward approach is to utilize a *grow-only set*, which can only be mutated by adding elements. The merge function is defined as the union of the two sets, written as $\text{merge}(S, T) = S \cup T$. Moreover, a union of two sets is commutative, associative and idempotent, which together with a partial order makes the state form a monotonic join-semilattice. This guarantees that the set will converge to the same state, as long as the same set of elements have been added to both replicas. Grow-only sets are an example of an *anonymous* CRDT, as they do not require a unique identifier for each replica and the operations are not specific to a single replica. An example of a *named* CRDT are causal CRDTs such as the *Add-Wins Set* [23]. See Listing 2.3 for an example implementation of a grow-only set.

The next step is the *2P-Set*, which is a set that can be mutated by adding and removing elements. It achieves this by using two grow-only sets, one for adding elements and one for removing elements (the tombstone set). Which shows that CRDTs can be composed to create more complex data structures.

Listing 2.3: Example implementation of a Grow-Only set.

```
defmodule GSet do
  def new, do: MapSet.new()

  def value(s), do: s

  def add(s, v), do: MapSet.put(s, v)

  def merge(s1, s2), do: MapSet.union(s1, s2)

  def lookup?(s, v), do: MapSet.member?(s, v)
end
```

2.4.7 CL-Set

In this context, we introduce the Causal Length set, which was defined in [24]. Similar to the aforementioned CRDTs, the CL-set is a variation designed to address the primary issue of general-purpose CRDTs, namely causality between updates. Each element within the set is associated with a Causal Length, which is used to determine whether an element is present in the set. It's an *anonymous* CRDT, which means that it does not require a unique identifier for each replica. This is ideal for dynamic systems, where the set of replicas can change over time.

An element is present in the set if the Causal Length is odd, and absent if it is even. It can be observed that additions and removals are causally dependent, as an element must be added before it can be removed. Moreover, an element can only be added to the set if it is not already present in the set, and removed if it is present in the set.

An example implementation of a Cl-Set in Elixir can be found in Listing 2.4. Each element in the set is associated with a Causal Length, which is used to determine whether an element e is present in the set s. An element is present in the set if the Causal Length is odd, and absent if it is even. The keys in the map are the elements in the set, while the values are the Causal Lengths. Any element not present in the set has a default Causal Length of 0. We can see that additions and removals are causally dependent, as an element must be added before it can be removed.

The insertion of an element e is done by incrementing the Causal Length of e by 1, if it is not already present in the set. In the event that the element in question is already present in the set, the addition will be ignored.

A removal of an element *e* is done by incrementing the Causal Length of *e* by 1, if it is already present in the set. If *e* is not present in the set, the removal will be ignored. If it is present, then its Causal Length is odd and incrementing it by 1 will result in its removal from the set.

Finally, the merge function is defined as a union of the two sets, where the Causal Length of each element is the maximum of the two Causal Lengths.

Listing 2.4: example implementation of a Cl-Set crdt.

```
defmodule CLSet do
  def in_set?(s, e), do: Map.get(s, e, 0) |> Crdt.
     is_odd()

  def add(s, e) do
    unless in_set?(s, e) do
      Map.update(s, e, 0, &(&1 + 1))
    end
  end

  def remove(s, e) do
    if in_set?(s, e) do
      Map.update(s, e, 0, &(&1 + 1))
    end
  end

  def merge(s1, s2) do
        Map.merge(s1, s2, fn _k, v1, v2 -> max(v1,
           v2) end)
  end
end
```

2.5 Datalog

2.5.1 Logic Programming

Logic programming is a programming paradigm that is based on formal logic. It's used to express facts and rules about a problem domain, and then use a logical inference engine to derive new facts from the existing ones. This enables the implementation of highly sophisticated functionality, including the absence of side effects, no explicit control flow and strong termination and correctness guarantees. A logic program consists of a finite set of facts and rules, which are used to derive new facts. The facts are assertions of the world in the problem domain, while the rules are logical implications that allow us to infer new facts from existing ones.

Logic programming languages can be classified according to their expressiveness, with the least expressive being *Relational algebra*. The next step is *Datalog*, which is often considered as an extension of relational algebra, with the addition of recursion. Furthermore, Datalog can be extended with negations, which is referred to as *Stratified Datalog*. Finally, if we wish to express algorithms beyond polynomial time, we can use *Prolog*. Prolog is Turing complete, but it comes at the cost of losing the guarantees of termination and correctness.

The remainder of this section we will focus on Datalog and its extensions, as this is the language used in PRAD. We will start by introducing the syntax and semantics of Datalog, and then continue by explaining the extensions of negations and stratification.

We will now present the syntax and semantics of Datalog, as well as relevant terminology.

2.5.2 Syntax

In this section, we let **V** be an infinite domain of variables, and **D** be an equally infinite but disjoint domain of values. Datalog consists of *atoms* and *terms*. A *term* x_i is either a variable from **V** or a value/constant from **D**. An *atom* $R(x)$ is a function R along with a tuple of terms $x = (x_1, \ldots, x_n)$. An *atom* with only constants is called a *ground atom*.

A Datalog *schema* is a set of relation names, and every function R has an associated arity n, which is the number of terms in the tuple.

A Datalog *program* is a set of finite Datalog *rules*, each of which is a Horn clause of the form:

$$R_1(x_1) \leftarrow R_2(x_2), \ldots, R_n(x_n) \tag{2.1}$$

Here, $n \geq 1$, $R_1, \ldots, R_n$ are relation names and $x_1, \ldots, x_n$ are tuples of terms. Further, Every variable in x must also appear in at least one of the other tuples x_i [25]. The *head* of the rule is $R_1(x_1)$, while the *body* is formed by $R_2(x_2), \ldots, R_n(x_n)$.

Given a *valuation* $\mathcal{V}$, an *instantiation* of a rule via substitution is defined as:

$$(R_1(\sqsubseteq(x_1))) \leftarrow (R_2(\sqsubseteq(x_2))), \ldots, (R_n(\sqsubseteq(x_n))) \tag{2.2}$$

In this context, $\sqsubseteq(x_i)$ represents the substitution of the terms in x_i according to $\sqsubseteq$.

Continuing, let $\mathcal{P}$ be a datalog program. An *extensional* relation are relations stored only in the body of the rules, since they are stored in the database. An *intensional* relation are relations occurring in both the head and body of the rules, since they are computed from the extensional relations. The *extensional schema*, denoted $edb(\mathcal{P})$ is the set of extensional relations in $\mathcal{P}$, while the *intensional schema*, denoted $idb(\mathcal{P})$ is the set of intensional relations in $\mathcal{P}$. The *schema* of $\mathcal{P}$ is the union of the extensional and intensional schema, denoted $schema(\mathcal{P}) = edb(\mathcal{P}) \cup idb(\mathcal{P})$.

Furthermore, in order to guarantee that the set of facts which can be derived from a program is finite, we require that the rule is *safe*. A rule is *safe* if all of its variable is *range-restricted*, which means that a variable in the head of the rule must also appear in the body of the rule, or be equivalent to a variable that does.

2.5.3 Semantics

The semantics of Datalog can be defined in three different, but equivalent ways.

Model-theoretic

The model-theoretic semantics of Datalog is based on *first-order logic*, where the Datalog rules are interpreted as logical constraints. Consequently, the database instance that satisfies all the constraints is designated as a *model* of the program. Let $\mathcal{P}$ be a Datalog program, and $\mathcal{D}$ be a database instance. Then $\mathcal{D}'$ is a model of $\mathcal{P}$ if and only if $\mathcal{D}'$ extends $\mathcal{D}$ and satisfies all the rules in $\mathcal{P}$. Nevertheless, there can be multiple models of a Datalog program. In this context we are interested in the *minimal model* of the program.

For a Datalog program $\mathcal{P}$ and a database instance $\mathcal{D}$ over *edb (P)*. A model $\mathcal{D}'$ of $\mathcal{P}$ is an instance over *schema (P)* such that $\mathcal{D}'$ extends $\mathcal{D}$ and satisfies all the rules in $\mathcal{P}$. The *semantics* of $\mathcal{P}$ on $\mathcal{D}$, is the minimal model of $\mathcal{P}$, containing $\mathcal{D}$, if it exists [25].

In other words, the results of a Datalog program should satisfy all the rules and be the smallest possible set of facts.

Fixpoint-theoretic

The least fixpoint semantics of Datalog is based on the *immediate consequence operator* $\mathcal{T}_{\mathcal{P}}$ for a Datalog program $\mathcal{P}$. It applies all the rules in $\mathcal{P}$ to a database instance $\mathcal{D}$, and adds the *immediate consequences* to $\mathcal{D}$. An atom R is an immediate consequence of a program P if R is grounded in D or a $R_1 \leftarrow R_2, \ldots, R_n$ is a ground instance of a rule in P, and $R_2, \ldots, R_n$ are in D. T_P is then applied iteratively to $\mathcal{D}$ until no more new facts can be derived, and the result is the least fixpoint of $\mathcal{P}$. This is equivalent to the minimal model of $\mathcal{P}$ in the model-theoretic semantics.

These semantics are useful for bottom-up evaluation of Datalog programs.

Proof-theoretic

Alternatively, if we wish to evaluate the Datalog program top-down, we can use the proof-theoretic semantics. Which defines the result of a Datalog program as the set of atoms that can be *proved* from the rules and the database instance. A proof is a tree of atoms, where the root is the atom to be proved, and the leaves are the facts in the database instance.

The set of atoms that can be proved from a database instance $\mathcal{D}$ using the rules in a Datalog program $\mathcal{P}$ is precisely $\mathcal{P}(\mathcal{D})$ [25].

2.5.4 CALM

As previously stated in definition 9, monotonicity is a key property of coordination-free systems to establish consistency, it's the CALM theorem. It originates from the CALM conjecture, which states that in Datalog, when a program is expressible in queries that are monotonic and the data is inflationary, then there exists an eventually consistent and coordination-free implementation of the program [17]. The CALM conjecture was later formally proven by Ameloot et al. [26].

In PRAD with the *Cl-set* CRDT, we can apply the CALM conjecture to ensure that the program is eventually consistent and coordination-free, since the *Cl-set* is inflationary for both addition and removal of elements.

2.5.5 Datalog with Negation

Negation is a powerful tool in logic programming, allowing us to ask for the *absence* of a fact or inference. However, this approach also introduces some issues with the semantics of Datalog, as it can lead to non-termination and inconsistency especially when combined with recursion. The following query, presented in listing 2.5. It can be observed that the query $Child(alice)$ will result in non-monotonic behavior, as it will infinitely recurse between the two rules.

Listing 2.5: Example query with negation.

```
human(alice).

Adult(X) :- human(X), not child(X).
Child(X) :- human(X), not adult(X).
```

In the light of iterative evaluation of rules, the initial status of the negated atoms is that they are true. However, as the rules are evaluated the negated atoms will become false. To avoid recursion, we can organize the rules into layers or *strata*.

2.5.6 Stratified Datalog

A Datalog program P is *stratified* if its rules can be partitioned into *strata*, $P_1, P_2, \ldots P_n$ such that the following conditions hold:

1. All rules mentioning relation R in a non-negated atom are in the same stratum P_j, with $j \geq i$.

2. All rules mentioning relation R in a negated atom are in the same stratum P_j, with $j > i$.

By applying rules exhaustively within each stratum, it is possible to avoid non-monotonic behavior. However, not every Datalog program can be stratified. The stratified evaluations of rules terminate in a finite number of steps (bounded by the number of possible facts), upholding the monotonicity of Datalog. At present PRAD does not support stratified Datalog, but it is a potential extension to the system.

2.5.7 Evaluation Strategies

This section we will present some evaluation strategies used by Datalog engines. We will start by introducing the naive evaluation strategy, and then proceed to the semi-naive evaluation strategy.

The *naive* evaluation strategy is the simplest evaluation strategy, and is based on the fixpoint semantics of Datalog (bottom-up). It evaluates the rules in a Datalog program *iteratively*, starting from the extensional relations. At each iteration, the rules are evaluated, and the immediate consequences are added to the database instance. This process is repeated until no further new tuples can be derived, at which point the result is the least fixpoint of the program as discussed in the section2.5.3. One potential issue with this evaluation strategy is that we can end up deriving the same tuples on multiple occasions, which could be highly inefficient.

The *semi-naive* evaluation strategy [27] is an optimization of the naive evaluation strategy. Its objective is to reduce the number of redundant derivations. The input tuples computed in the previous iteration of a recursive program are stored in a *delta*, are used as input to the current iteration to compute the new tuples. If and only if the new tuples are not already present in the delta, are they used as input to the subsequent iteration. This ensures that only new tuples are derived, and that the same tuples are not derived multiple times.

Let R be an IDB relation in a Datalog program $\mathcal{P}$, and R_i be the set of tuples of R in iteration $i \geq 1$. Every iteration i it computes the new tuples $\Delta R_i = R_i - R_{i-1}$. Then ΔR_i is used to compute the new tuples in the next iteration $R_{i+1} = R_i \cup \Delta R_i$. This is repeated until $\Delta R_i = \emptyset$. The result is the least fixpoint of the program.

The *semi-naive* evaluation strategy also minimizes the amount of communication between the nodes, as it only sends the new tuples ΔR to the other nodes at every iteration. This strategy is used by PRAD, which permits the valuation of rules in a variety of orders and on multiple occasions, without deriving the same tuples multiple times. This approach allows for the execution of PRAD programs without the need for coordination.

2.6 Fault-Tolerance

In the context of distributed system design, it is important to consider the fault-tolerance of the system. It is inevitable that a system will eventually encounter a failure, and it must be able to handle it. A distributed stream processing system must be capable of handling failures, including those resulting from node crashes, network partitions, and message loss. This section will present some common failure models and approaches to fault tolerance in distributed systems.

2.6.1 Failure Models

In a distributed stream processing system, there are several possible failure models that must be considered. Both the processes and the network may fail, and the system must be capable of handling these failures. In the literature, there are four main categories of failures: *Crash failures*, *Omission failures*, *Timing failures* and *Byzantine failures* [28, p.430].

Crash failures are the most straightforward type of failure in distributed systems. The system halts, but is working correctly up until the point of failure. It is noteworthy that, following the system halts, it will no longer respond to any messages. This is readily detected by the other sites in the system, and recovery is typically straightforward.

Omission failures are more complex than other types of failures, as the system can still work correctly, but it may not receive or send messages. Two distinct types of omission failures can be identified: *send omissions* and *receive omissions*. In a *send omission*, a process fails to send a message, while in a *receive omission*, a process fails to receive a message. In general, these are caused by network partitions or input buffer overflows. The detection of these failures is more complex, as it would require some form of timeout or heartbeat mechanisms.

Next are *Timing failures*, which are caused by the system working, but not responding within a certain time frame. This can be caused by network congestion or overloaded servers, which results in a performance issue rather than a fault. This primarily affects the availability of the system, and is applicable to synchronous systems.

Finally, *Byzantine failures* represent the most complex type of failure, the system *appears* to be working correctly, but is sending incorrect messages. This can be caused by software bugs, hardware failures or malicious attacks. It is the most challenging to detect, and may require complex mechanisms such as voting or consensus algorithms. Examples of Byzantine failures include *IoT sensor malfunction* and *malicious attacks*.

2.6.2 Approaches to Fault Tolerance

There are several approaches to fault tolerance in distributed systems, but the most common methods are redundancy and replication. The system is designed to be resilient failures, and continue to work correctly even if some components fail. Replication can be divided into two categories: *Active replication* and *Passive replication* [28, p.391]. In *Active replication*, all replicas are active and execute the same operations in parallel. A synchronous mechanism is used to ensure that all replicas agree on the result. In *Passive replication*, only one replica is active, while the others are passive and only become active if the primary replica fails. The operations are then propagated to the other passive replicas. The mechanism is asynchronous, and it is frequently used in distributed systems.

3 Design

This chapter provides an overview of the existing approach for a coordination-free parallel and replicated Datalog extension PRAD [6]. This chapter presents a method for augmenting Datalog with semiring (p-semiring) provenance to provide distributed execution. Furthermore, we present a method for equipping the aforementioned p-semiring with Cl-Set CRDT for coordination-free and eventually consistent Datalog. We present how distributed execution plans are generated to meet parallel and fault-tolerant requirements. This will lead us to investigate how it handles faults at the site level. The required properties for a fault-tolerant system are presented. We will present our approach to maintain fault-tolerance requirements in a distributed setting, as well as our strategy for minimizing communication bandwidth.

3.1 PRAD Overview

In order to describe the PRAD runtime, we employ a classical transitive-closure Datalog program. The database schema comprises six Extensional Database (EDB) relations, as shown in Table 3.1.

The *Edge* relation contains the edges of the graph, where each tuple represents a direct edge between stations. The *Path* data structure contains paths traversing the edges. The relations *StationEU* and *StationUS* contain the stations in Europe and the United States of America, respectively. *StationFoo* contains

Relation	**Attributes**
Edge	*From*, *To*
Path	*From*, *To*
StationEU	*Station*
StationUS	*Station*
PathFoo	*From*, *To*
StationFoo	*Station*

Table 3.1: Relations in the database schema.

the stations that the entity *Foo* travels to. The *PathFoo* data structure contains the paths among the stations in *StationFoo*.

The Datalog program *Paths* generates all paths from the edges in accordance with Rules 3.1 and 3.2.

Listing 3.1: Rule 1 Edge

```
Path(X, Y) :- Edge(X, Y).
```

Listing 3.2: Rule 2 Path

```
Path(X, Y) :- Edge(X, Z), Path(Z, Y).
```

Moreover, Rule 3.3 queries the paths from the stations for *Foo*.

Listing 3.3: Rule 3 Query for Foo

```
PathFoo(X, Y) :- StationFoo(X), StationFoo(Y), Path(X, Y).
```

The aforementioned Rules and Relations are run in a distributed setting and are available on the following sites:

Site	**Relations**
ServerEU	*Site*
ServerUS	*Site*
Server	*Site*
ClientFoo	*Site*

Table 3.2: Relations at each site.

The *ServerEU* and *ServerUS* relations contain the stations in Europe and the United States of America, respectively. The *Server* variable contains the servers on which the distributed Datalog program is executed. The *ClientFoo* relation contains the devices associated with the *Foo* entity.

PRAD extension to Datalog programs incorporates a relation for site configuration, which specifies which the sites responsible for each relation. The site configuration relation is defined as follows:

$$\text{Cfg(Relation, Attribute, Site)}$$

With the following site configuration rules:

Listing 3.4: Site Configuration Rules (* denotes a wildcard)

```
Cfg(Edge, To, X, Y) :- StationEU(X), ServerEU(Y).
Cfg(Edge, To, X, Y) :- StationUS(X), ServerUS(Y).
Cfg(PathFoo, *, *, Y) :- ClientFoo(Y).
```

The configuration rules for the site specify that the *Edge* to European stations is available on the *ServerEU* sites, while the *Edge* to American stations are available on the *ServerUS* sites. The final rules specify that the *PathFoo* relation is available on all of *Foo*s devices. Depending on whether each *Server* only contains one site value, the site configuration rules can be used to specify a partition of relation *Edge* on *To* attribute. In the event that the *Server* relation contains multiple site values, the site configuration rules can be used to specify a replication of relation *Edge* on *To* attribute, ensuring that the relation is available on e.g. all European sites.

Moreover, the site configuration rules specify that the *PathFoo* query result is replicate to ensure it is available on all devices belonging to *Foo*. The result of the query is updated as long as the *Path* program is running and *Foo*s devices are online. In the event that a device goes offline, the result of the query is still available on the other devices. Furthermore, *Foo* can specify how it wants to propagate the result of the query to its devices.

Suppose that the relations *ClientHomeFoo* contains the home devices of *Foo*, e.g. IoT devices. Then we can modify the site configuration rules as shown in Listing 3.5.

Listing 3.5: Site Configuration Rules for Home Devices

```
PathHomeFoo(X, Y) :- PathFoo(X, Y)
PathIoTFoo(X, Y) :- PathHomeFoo(X, Y)

Cfg(PathHomeFoo, *, *, Y) :- ClientHomeFoo(Y).
Cfg(PathIoTFoo, *, *, Y) :- IoTHomeFoo(Y).
```

In accordance with the site configuration rules set in Listing 3.5, the result of the query is available on all of *Foo*s home devices. The query result is initially collected at the *Client* and then subsequently propagated to the *IoT* devices. This prevents *IoT* devices from engaging in resource-intensive computations and from actively participating in the network communication.

Suppose that for the *Paths* program, the input facts are in the relation *Edge* and the output facts are in the relation *PathFoo*. Then intermediate relation *Path*, is not included in the input or output facts, and lacks a site configuration. At runtime, we can specify the parallel and fault-tolerant requirements for the program execution in the *Paths* program. The parallel requirements specify how the program can be executed in parallel, and the fault-tolerant requirements specify how many site failure it can tolerate. In accordance with the aforementioned requirements, the system is capable of regenerating the site configuration for network sites as specified in the *Server* relation.

3.1.1 Architecture

The high-level architecture of PRAD-runtime is depicted in Figure 3.1. The system is composed of a two-layer architecture. The upper layer is referred to as the *Application Relation Layer* (APP). The APP layer is comprised of a set of relations analogous to those found in conventional relational database. The relations may be either base relations (EDB) or derived relations (Intensional Database (IDB)). This is the location where applications can update and query the data stored in the relation. This is in accordance with the schema of the application database schema. The bottom layer is referred to as the *Augmented-Relation layer* (AUG). The AUG layer is a superset of the APP layer. It associates each relation in the APP layer with a set of metadata based on the two algebraic structures, the p-semiring and the ClSet CRDT. Moreover, the AUG layer is responsible for maintaining the consistency requirements and to execute incremental maintenance of both base and derived data.

The interrelationship between the aforementioned layers can be described as relatively straightforward. Any query to the APP layer can be satisfied without triggering the AUG layer. Nevertheless, an update to the APP layer will prompt the AUG layer to ensure consistency with the former. In Figure 3.1, applications at sites s_1 and s_2 make concurrent update to the relation R_1 in the APP layer. Subsequently, the updates are then augmented with metadata in the relation $\tilde{R_1}$. The two relations are both locally stored on each site, which allows them to send each other their local updates, when both are online. Synchronization is

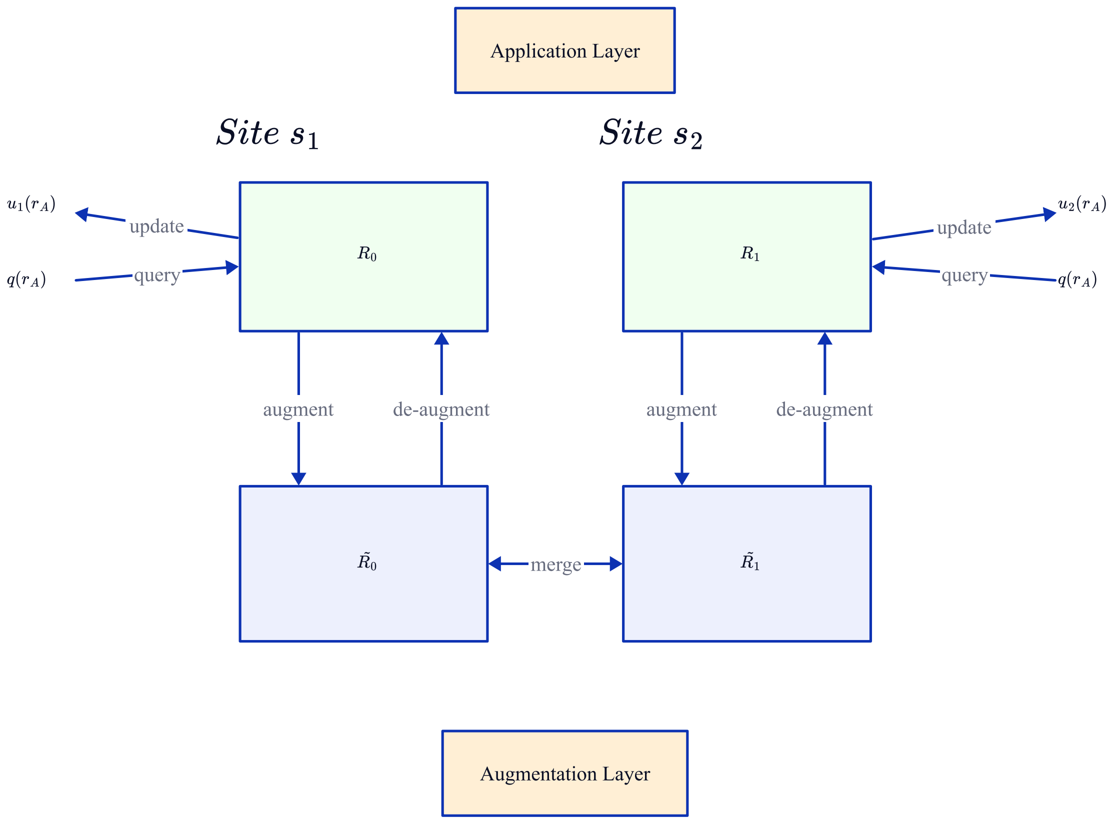

Figure 3.1: PRAD Architecture.

achieved using an anti-entropy procedure at the AUG layer. These updates are either the whole state or a set of join-irreducible states, from Section 4. When a site receives a remote update, it will merge the update into its local relation $\tilde{R_1}$, and then with the de-augmented update the APP layer with the new relation R_1. The system ensures that when two sites s_1 and s_2 have received the same set of updates, they will have the same R and $\tilde{R}$ states, regardless of the order in which the updates were received.

Sites s_1 and s_2 transmit their updates to downstream sites s_u. Upon receipt of the updates, site s_u will incrementally update its local augmented relation, denoted by $\tilde{R_u}$, and then the APP relation, denoted by R_u, with the de-augmented update. The system ensures that the relation state of s_u is consistent with the states of s_1 and s_2 as if they were in a non-distributed setting.

In PRAD a *relation* is a set of tuples, which again contain multiple values. The augmented relations are represented as CL-sets 2.4.7, which are sets of tuples with provenance information. Therefore, the tuples in the relations are CL-set elements. Which means that the tuples are tuples of values and a set of associated provenance information. The implications of this are that the aforementioned operations are defined according to the CL-set semantics.

A more detailed view of the PRAD site is presented in Figure 3.2. It consists of three main components: the *Compilation*, the *Runtime*, and the *Storage Layer*. The Compilation component is responsible for parsing the inputs, which are the user *query*, the EDB (Rules) and the new IDB (Facts). Next, the distribution policy is created, which is a function that maps each site to a set of facts in the schema. Subsequently, it defines a communication strategy, which essentially is the manner in which the upstream and downstream sites communicate. To this end, it gathers information about the sites states from the Storage Layer metadata. The Compilation component then generates the execution plan for the program, and sends it to the Runtime component.

The Runtime component is responsible for executing the program, it does this by applying the rules in the program to the local instance of the site using semi-naive evaluation. The local facts are stored in the Erlang Term Storage (ETS) inside the Storage Layer. Once the site has reached a local fixpoint, it sends the derived facts to the downstream sites or, alternatively, to the user.

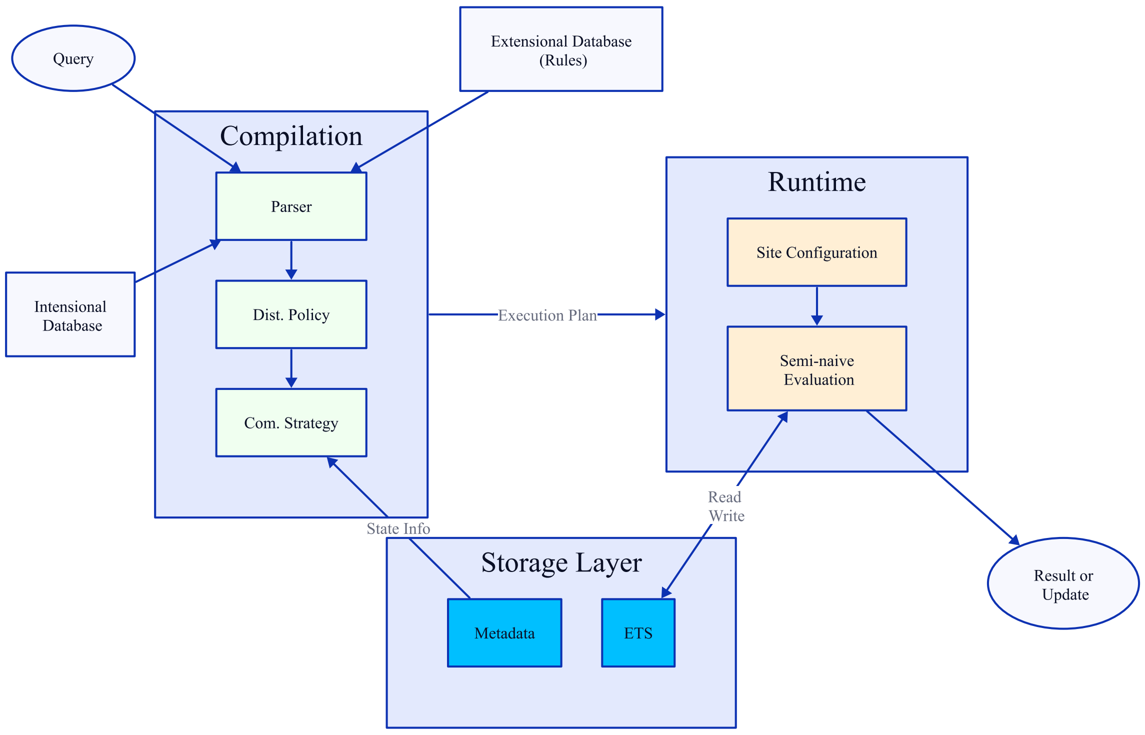

Figure 3.2: Example PRAD Site.

The general methods for interacting with the system are to either update or query tuples. Updates are done by the insertion of a new tuple or the modification an already existing tuple. A tuple can only be inserted if it does not already exist in the relation. This implies that it could already exist in the derived relation with an even causal length. Subsequently, an insertion would then increment the causal length. It is noteworthy that new tuples are always inserted with a causal length of 1. As said above, tuples may only be updated or deleted if it has an odd causal length.

The components are explained in greater detail in the subsequent sections.

3.1.2 Datalog^p

In general, the provenance of an IDB fact indicates the manner in which the fact is derived. Green et al. [29] introduced a general framework of semiring provenance, which $Datalog^p$ is a special instantiation of. $Datalog^p$ augments Datalog with a p-semiring, explained in Section2.1.3. The following section, will demonstrate the transformation of a typical Datalog program P, into a $Datalog^p$ program $\ddot{P}$.

Let prv be the domain of provenance expressions in the p-semiring, disjoint from the var and dom domains. A *p-instance* $\ddot{I} : U \longrightarrow prv$, is a function from the set of all facts U to prv. The *support* of a p-instance is the set of facts whose provenance is not zero. This is denoted by $\mathbf{supp}(\ddot{I}) = \{f \in U \mid \ddot{I}(f) \neq 0\}$. The *support* of a p-instance is identical with the instance of the normal Datalog program, which may be expressed as $I = \mathbf{supp}(\ddot{I})$. For any fact $f \in I$, the mapping $f \mapsto \ddot{I}(f)$ is a $p-fact$ in the p-instance $\ddot{I}$.

Now, we will define the manner in which the provenance values of $p-facts$ are generated. For an EDB fact f, the provenance value is defined as $\ddot{I}(f) = id_f$, where id_f is a unique identifier for the fact f. The domain of identifiers is $uid \subset prv$. For a normal Datalog valuation v of rule τ in instance I, there exists a valuation $\ddot{v}$ in the corresponding p-instance $\ddot{I}$, such that

$$\ddot{I}\left(v\left(\text{ head }_\tau\right)\right) = \prod_{f \in v\left(\text{body}_\tau\right)} \ddot{I}(f) \tag{3.1}$$

For example, seen in Figure 3.3 if p-instance $\ddot{I}$ contains the two $p-facts$ $\textsc{Edge}(1,2) \mapsto l_{12}$ and $\textsc{Path}(2,4) \mapsto l_{24}$, then a valuation of the second Rule in $\ddot{I}$ derives a $p-fact$ $\textsc{Path}(1,4) \mapsto l_{12} \cdot l_{24}$. The union of two p-instance $\ddot{I}_1$ and $\ddot{I}_2$ is defined as:

$$\forall f \in U : \left(\ddot{I}_1 \cup \ddot{I}_2\right)(f) = \ddot{I}_1(f) + \ddot{I}_2(f) \tag{3.2}$$

For instance, if $\ddot{I}_1$ contains the $p-fact$ $\textsc{Path}(1,4) \mapsto l_{12} \cdot l_{24}$ and $\ddot{I}_2$ contains the $p-fact$ $\textsc{Path}(1,4) \mapsto l_{13} \cdot l_{34}$, then the union $\ddot{I}_1 \cup \ddot{I}_2$ derives the $p-fact$ $\textsc{Path}(1,4) \mapsto l_{12} \cdot l_{24} + l_{13} \cdot l_{34}$.

In the case that f is neither in $\mathbf{supp}(\ddot{I}_1)$ nor in $\mathbf{supp}(\ddot{I}_2)$, the provenance value is zero, $\left(\ddot{I}_1 \cup \ddot{I}_2\right)(f) = 0$. If f is in $\mathbf{supp}(\ddot{I}_1)$ but not in $\mathbf{supp}(\ddot{I}_2)$, then $\left(\ddot{I}_1 \cup \ddot{I}_2\right)(f) = \ddot{I}_1(f)$.

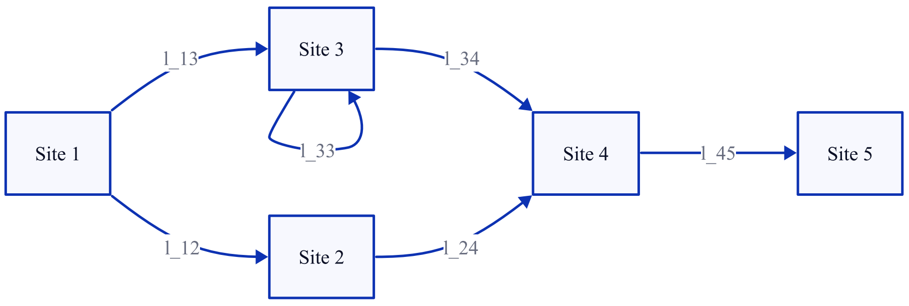

Figure 3.3: Example Edge facts.

The immediate consequence operator for program $\ddot{P}$ is defined as

$$T_{\ddot{P}}(\ddot{I}) = \ddot{I} \cup \bigcup_{\tau \in P, v\left(\text{body}_\tau\right) \subseteq I} \{\} \tag{3.3}$$

with contents

$$\{v\left(\text{head}_\tau\right) \mapsto \prod_{f \in v\left(\text{body}_\tau\right)} \ddot{I}(f)\} \tag{3.4}$$

Now, we define an order $\ddot{\leq}$ on p-instances. For two p-instances $\ddot{I}_1$ and $\ddot{I}_2$, if and only if $\forall f \in U : \ddot{I}_1(f) \leq \ddot{I}_2(f)$, then $\ddot{I}_1 \leq \ddot{I}_2$. Clearly, if $\ddot{I}_1 \leq \ddot{I}_2$, then $\mathbf{supp}(\ddot{I}_1) \subseteq \mathbf{supp}(\ddot{I}_2)$.

Notice that the union of p-instances is identical to the LUB of the p-instances with respect to the order $\ddot{\leq}$. That is $\ddot{I}_1 \cup \ddot{I}_2 = \ddot{I}_1 \sqcup \ddot{I}_2$. The union of p-instances is just like normal Datalog inflationary under $\ddot{\leq}$.

Another way to compare $Datalog^p$ to Datalog, we can define Datalog in terms of $Datalog^A$. A Datalog instance I, a A-instance is the function $I^A : U \longrightarrow P$, for all facts $\forall f \in U : I^A(f) = (f \in I)$. Because there is a semiring-homomorphism h from p-semiring to the A-semirung according to [29]. We can use the homomorphism to define a $Datalog^p$ instance I^p from a $Datalog^A$ instance I^A as

$$h(T_{\ddot{P}}(\ddot{I}) = T_{PA}(h(I^A))) = T_P(I) \tag{3.5}$$

This implies that $Datalog^p$ programs can be translated in a straightforward manner to $Datalog^A$ programs, with provenance values are being generated automatically.

Given that $T_{\ddot{I}}$ is inflationary and the $\cdot$ and $+$ operations are idempotent, for any p-instance $\ddot{I}$ with finite support, the iterative application $T_{\ddot{P}}$, starting $\ddot{I}$ reaches a fixpoint, denoted by $\ddot{P}(\ddot{I})$. The p-semiring provenance of an IDB fact f describes how f depends on EDB facts. For instance, PATH$(1,4)$, depends on either l_{12} and l_{24} or on l_{13} and l_{23}. While PATH$(1,3)$ depends soley on l_{13}.

This facilitates the straightforward monitoring of the dependencies inherent in the derivation process, a capability that will prove invaluable in the restoration of crashed sites.

3.1.3 Distribution Policies for Program Execution

We define a *network* as a finite non-empty set S of values with $S \subseteq dom$ representing the sites. A *distribution policy*, denoted by D over schema σ and network S is a function that maps each site $i \in S$ to subsets of facts in the schema σ. Formally, this is defined as $\text{facts}_D : S \longrightarrow P\left(U_{|\sigma}\right)$, where S is the network and σ is the schema. For a given database instance $I, I_{|i}$ is the local instance at site i, defined as $I_{|i} = \text{facts}_D(i) \cap I$. A Site i is considered *responsible* for fact f if $f \in \text{facts}_D(i)$. $\text{sites}_D(f)$ is the set of sites responsible for f.

Moreover, a PRAD *site-configuration* of a program P is defined as a distribution policy D that encompasses both the external database and the internal database $\text{edb}(P) \cup \text{out}(P)$. A PRAD *execution-policy* of program P is a distribution policy C over schema P that includes the site-configuration D of P and is consistent with the program P.

The distributed execution of a program P is divided into two phases: the *execution phase* and the *communication phase*. The execution phase is the phase during which each site i executes the program P, and applies the rules in P to the local instance $I_{|i}$ to obtain a set of derived facts. The communication phase is the phase during which the sites exchange the derived facts with the responsible sites. The communication phase is conducted asynchronously, and the sites may exchange the facts in any order. Once a site derives no new facts in the execution phase, the site is said to have reached its local *fixpoint*. The program P is said to have reached a *global fixpoint* if all sites have reached their respective local fixpoints.

A program P is said to be *parallel-correct* under a distribution policy D if for any instance I, the union of all facts at all sites at the global fixpoint under the same distribution policy D, denoted by $[P, D](I)$, is equal to the set of facts that would be derived by the program P in a centralized setting, denoted by $P(I)$.

Furthermore, a distribution policy D is said to support a valuation $v(\tau)$ in I if there exists a site i and a valuation $v(\tau) \in I_{|i}$. A policy is said to *strongly support* a program P if its supports all valuations within P. According to Ketsman et al. [30] strong support is sufficient for the program to be parallel-correct.

3.1.4 Coordination-Free Replication and Eventual Consistency

The objective of PRAD is to execute Datalog programs coordination-free and eventually consistent, while the data is partitioned and replicated across network sites. A Datalog^p program P as introduced in Section 3.1.2 is eventually consistent if it is both eventually convergent and parallel correct. We have already defined and proven parallel correctness in the previous section. We will now define eventual convergence. A p-instance I is said to be *convergent* if the provenance expression of the fact is the identical across all sites responsible for the fact.

A program P is said to be *convergent* if for any p-instance I, when P reaches a global fixpoint, $T_P(I)$ is convergent. Further, since the domain of provenance expressions is a distributive lattice, instances of the same relation partition form a Cl-Set CRDT, which means that they are eventually convergent without coordination.

Furthermore, a distributed execution of a Datalog program is *coordination-free* if the sites do not need to communicate in the form of acknowledgement of message delivery or coordination via consensus protocols [12] to achieve, with each other beyond EDB and IDB data to reach a global fixpoint. For a Datalog program using a semi-naive evaluation algorithm discussed in Section 2.5.7, coordination-free execution means that valuations of rules can be applied in any order and at any time, and the program will still reach a global fixpoint.

As previously established in Section 2.5.4, in any inflationary Datalog program, the rules can be applied in any order and time and the program will still reach a global fixpoint [17]. The PRAD runtime is an instance of a compiled and running Datalog program P, where each relation $\tilde{R}$, is associated with provenance metadata.

3.1.5 Replication and Parallel requirements

PRAD programs defines where input and output facts are stored, with the objective to make them available at the specified sites to meet the parallel and replication degree requirements. These are included in the generated distribution policy, as part of the execution plan. A Datalog program uses a bottom-up evaluation strategy, where the rules consist of evaluations of the relational algebra operations *join* and *project*. An evaluation has a parallel degree of N_p if every join operation in the evaluation can be executed on at least N_p sites. This is the foundation for other parallel join evaluations, such as the one seen in the widely used MapReduce [31] framework. In addition, an evaluation has a replication degree of N_r if for each valuation we have at least N_r sites that support it. This would mean that for a strong support of a program, the replication degree would be greater than or equal to 1, $N_r \geq 1$. This implies that currently the PRAD system uses a static configuration of replication, without being able to dynamically change the replication degree.

3.2 Communication Strategies

This section will describe the two communication strategies 1-1 and N-N. In the 1-1 strategy, a single partition replica of a relation joins with a single partition replica of another relation. This implies that a fact derived at an upstream site is sent to a single downstream site. On the other hand, in the N-N strategy, all partition replicas of a relation joins with all partition replicas of another relation. Consequently, a fact derived at the upstream replica is sent down to all downstream replica sites.

Suppose for Rule 3.6 we have generated partition replicas $Q^{n}_{p,r}$, $R^{n_1}_{p,r}$ and $S^{n_2}_{p,r}$ where $1 \leq p \leq N_p, 1 \leq r \leq N_r$.

Listing 3.6: Example Rule Q

```
Q :- R, S.
```

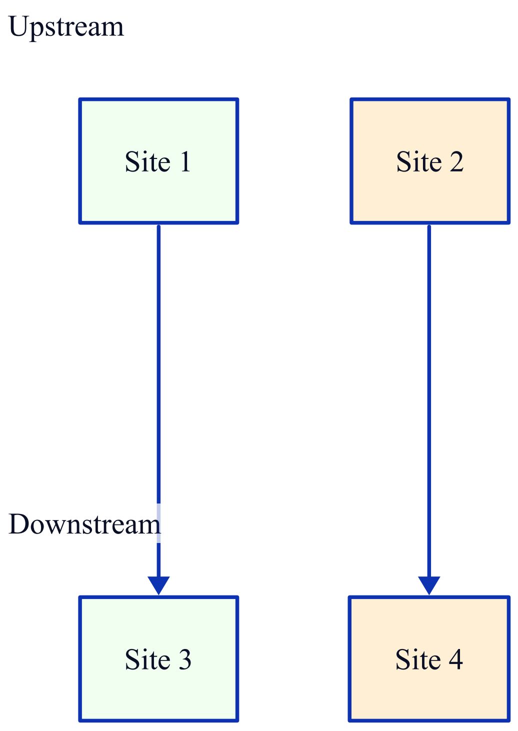

Figure 3.4: Communication Strategy 1-1.

With the 1-1 strategy, the r-th replica of the p-partition of R^{k_1} joins with the same r-replica of the p-partition of S_2^k. The derived facts are then dispatched to the r-th replica of different partitions of Q^k. This is shown in Figure 3.4, where the derived facts from Site 1 are sent to Site 3, and the derived facts from Site 2 are sent to only Site 4.

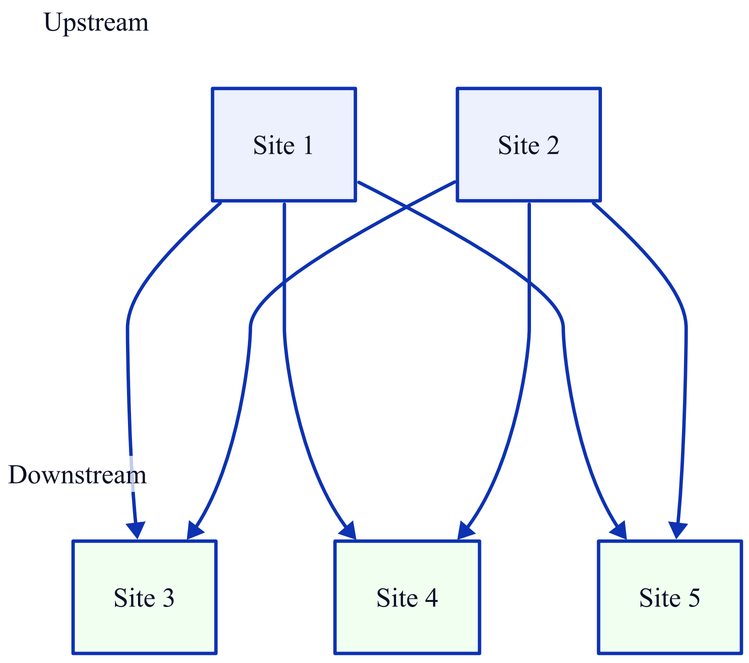

Figure 3.5: Communication Strategy N-N.

As seen in Figure 3.5, the N-N strategy is more complex than the 1-1 strategy. With the N-N strategy, we can replace the different replica relation names $Q^k_{p,r}$, with a single relation partition Q^k_p for each relation partition p of Q^k (same for R^{k_1} and S^{k_2}). Such that the sites of Q^k_p are equal to the union of all sites $Q^k_{p,r}$ for $1 \leq r \leq N_r$, denoted as:

$$\text{sites}\left(Q^k_p\right) = \bigcup_{1 \leq r \leq N_r} \left\{\text{site}\left(Q^k_{p,r}\right)\right\} \tag{3.6}$$

The derived facts are dispatched to all replicas of the different partitions of Q^k. This means that the derived facts from Site 1 and Site 2 are sent to all downstream sites, Site 3, Site 4 and Site 5.

We can see that the 1-1 strategy is effectively the same as running N_r instaces of the program with replication degree equal to 1. However, they have to be treated differently on repair, as seen in the next section on fault-tolerance.

3.3 Fault-Tolerance

Achieving fault-tolerance in a distributed stream processing system is a challenging undertaking that requires careful consideration of the system's architectural framework and the nature of the data being processed. In the context of PRAD, in order to maintain the replication degree a fault-tolerant system must be able to recover from site failures and network partitions, while maintaining the correctness and consistency of the data. Moreover, since the system is designed with local-first principles in mind, the focus is on the edge nodes where the input and output of the system occur.

Our approach to fault-tolerance is twofold. Firstly, the objective is to minimize the impact of site failures by utilizing a combination of lightweight commits, along with repair and restoration mechanisms. It is not possible to relay on the guarantee of FIFO ordering of messages, which means that it is not possible to use only sequence numbers to ensure that the messages are delivered in the correct order. Secondly, we seek to reduce the communication overhead associated with repairing a site by employing our algorithm that minimizes the amount of data that needs to be exchanged between sites to restore state.

In the context of failure detection, we adhere the Erlang philosophy of *let it crash*. This implies that we do not actively attempt to detect failures, but rather focus on failing fast and then restoring the system to a consistent state after a failure has occurred. This approach is based on the assumption that the system can function correctly even in the presence of failures.

The following sections present our approach to fault-tolerance in a distributed setting, utilizing replication and restoration mechanisms. In addition, we investigate the potential for enhancing the efficacy of site repair through the utilisation of lightweight commits. Furthermore, we will provide an illustrative example to demonstrate the concepts presented.

3.3.1 System model

The following system model is assumed. A distributed system is defined as a set of sites, $S = \{s_1, s_2, \ldots, s_n\}$, that do not share memory. Each site, designated by the subscript s_i, is responsible for maintaining a durable state. It is possible that some sites may experience a crash and subsequently restart to a previous state, which may differ from the state that existed prior to the crash. A site is capable of sending a message to any other site via an asynchronous network. It should be noted that the network is unreliable, in that messages may be duplicated and reordered, but will eventually be delivered and not corrupted. This implies that the network may be partitioned, but that disconnected sites will eventually reconnect and that we have a *Crash* failure model.

3.4 Replication & Restoration

A site may be inaccessible for an extended period (or be offline for sufficiently long), and may subsequently become operational again. The execution of a program may continue with the sites that are still operational. No action is required as long as the number of failed sites is less than the replication degree N_r. This can be designated as the *tolerance level* of the system. An execution with a replication degree of N_r can tolerate $N_r - 1$ site failures. The number of failures that the system can tolerate is reduced by one for each additional site failure.

3.4.1 Basic Replication

Our first approach to repair a PRAD program execution to maintain the fault-tolerance requirements, we can utilize passive optimistic replication. This implies that sites propagate updates beyond the current execution to other sites and allows for sites to temporarily diverge. During the compilation of the program, we generate a distribution policy that specifies where and how the data is replicated with the parallel and replication degree requirements. Depending on the replication degree and the communication strategy as detailed in Section 3.2, we can repair a site by replicating the missing data from the other sites. For an N-N execution the site is simply replaced with the new replica site without modifying the other sites. In the case of an 1-1 execution, we also switch the downstream sites of $\bar{s}$ to the new replica site to fully repair the execution.

We first consider the case where a site $\bar{s}$ is down with a replication degree of $N_r \leq 2$ and communication strategy of 1-1. In this case need a complete repair of an execution, since the site is the only one responsible for the data. After site $\bar{s}$ we make a new replica of $\bar{s}$. Suppose that the site $\bar{s}$ holds the relation R (or most likely multiple relation $R_1, R_2, \dots, R_n$ in a partition). We create a new site s and populate R with a running replica of R from site s', let's call it R'. Suppose sites $u_1, \dots, u_n$ are the upstream sites of $\bar{s}$ that generate the facts in R, and sites $d_1, \dots, d_n$ are the downstream sites of $\bar{s}$ that consume the facts in R in their rule bodies. These will then be set as the new upstream and downstream sites of the new replica site s respectively.

If we are using a coordination-free and naive approach, the upstream sites u_i re-generate all derived facts that $\hat{s}$ was responsible for and send them to the new site s. Respectively, s re-generates all facts in R and sends them to the responsible downstream sites d_i. This is done until all involved sites have stabilized. The approach is correct, but it is costly, and is against the principle of asynchronous and incremental execution of Datalog programs.

3.4.2 Stateless Repair

A second approach that does not rely on the site state, is to utilize the provenance information (p-instance) within Cl-Map to determine which facts are missing. Let's focus on communication strategy 1-1, between the new replica s and the upstream sites u. We want to use u to update the state of R at s. First, s sends it Cl-Map cl_s to u. Upstream site u then calculates the difference between its own Cl-Map cl_u and cl_s. This difference is denoted by $\mathrm{cl}^\Delta = \{t, \mapsto \mathrm{cl}_u(t) \mid \mathrm{cl}_u(t) > \mathrm{cl}_s(t)\}$. We suppose that site u has the rule $R \leftarrow R_1, R_2$ in its program, meaning that the upstream site u contains R_1 and R_2. It then filters out the facts in R_1 and R_2 whose provenance identifiers are in $\mathbf{supp}(cl^\Delta)$. Finally, u uses these facts to derive the missing facts in R for s with the semi-naive evaluation algorithm and sends them to s. This process is repeated for all upstream sites u of s.

The newly constructed replica site s is capable of generating the requisite derived facts for the downstream sites d in a manner analogous to that described above. The difference cl^Δ can be reused to determine which facts are missing in the downstream sites d. Subsequently, s can use $\mathrm{cl}^\Delta = \bigcup_{1 \leq i \leq n} \mathrm{cl}^\Delta_{d_i}$ which is the union of the differences between the Cl-Maps of the downstream sites d_i and s, this is equivalent to the most recent state of R at s. To generate new derived facts for all the downstream sites $d_1, \dots, d_n$ of s. Even though a downstream site d may have received the facts from s that it already has received from $\bar{s}$ or other sites before, it will not affect the correctness of the execution. The downstream site d will simply ignore the duplicate facts as discussed in Section 3.1.3. This holds true as each site has either a EDB or IDB table with an associated Cl-Map. We basically build a dependency graph of the Cl-Maps to determine which EDB tables the specific IDB tables on a site depends on.

The second approach allows to further minimization of the communication overhead and the amount of data that needs to be exchanged to repair a downstream site d. To repair a downstream site d, it is sufficient to send the derived facts, the heads to the downstream sites, since they contain all the information needed to generate the bodies.

3.4.3 Lightweight Commits

PRAD adheres on local-first principles, meaning that data is stored and processed at the edge. This means that sites go offline and online frequently and can stay offline for extended periods of time. The system must be capable to restore the state, that is, to stabilize a site $\bar{s}$, when it comes back online. The restoration process is analogous to the repair process, but it needs to support partial restoration of the state. This is because the site $\bar{s}$ may have been offline for an extended period and may not have received all the updates that were sent to replicas.

In order to repair a site $\bar{s}$, we to need to fetch the missing data from upstream u or peer sites p. We employ two strategies for restoration, first the *stateless repair* strategy 3.4.2 which works just as for replication, and secondly the LWC strategy. The need for the lightweight commit strategy arises from the fact that it is currently not possible to determine if an output $p - fact$ will be resent from an upstream or not.

Two alternative solutions to this problem have been identified. The initial idea was to utilize *sequence numbers* for deltas. However, this approach would require messages to be delivered in FIFO order, a guarantee that is not iherent in the system. This renders it impossible to determine if a message has been re-sent or not. Instead, we draw inspiration from the Git [1] and Pijul [2] version control systems. This involves adding commits to the delta outputs at periodic intervals, to keep track of sets of updates that have been sent and received. Unfortunately, commits also exhibit a similar limitation as sequence numbers, as they are not guaranteed to be delivered in the correct order. This implies that messages prior to the commit may be lost or arrive after the commit. We propose the inclusion of Cl-Map deltas, which are join-irreducible states within the commits. As the Cl-Maps represent a summary of the site state, they can be used to determine which facts are missing and where they derive from.

1. https://git-scm.com/
2. https://pijul.org/

This approach entails a reduction in communication overhead, although it necessitates the introduction of a more persistent system state.

A LWC for a $p-fact$ $\ddot{f}$ is structured as a tuple of the form $\langle f, p_f, cl_f, m_f \rangle$. The f represents the $p-fact$ itself, p_f is the provenance identifier for the $p-fact$ and cl_f is the Cl-Map for the $p-fact$. m_f contains additional metadata about the $p-fact$, such as the site that generated it or the target site. It can be modified based on the specific requirements of the system.

Suppose an upstream site s_u sends a sequence of $p-facts$ to a downstream site s_d, followed by a LWC message c_n. The $p-facts$ after the previous LWC message c_{n-1} are included in the same commitment. Subsequently, the downstream site s_d will then merge the received $p-facts$ into its local state. When s_d receives a new LWC message c_{n+1}, it sends an *acknowledgement* message to s_u, if and only if it has merged all neccesary $p-facts$ into its state. To verify that all the necessary $p-facts$ have been merged, cl-maps are compared by calculating the difference between the cl-maps of the LWC message c_n and the current cl-map of the site. If the $p-facts$ are verified, the site s_d sends an acknowledgement to the upstream site s_u, to notify it that the previously sent updates are now included in its state.

The upstream s_u maintains a cl_i for each LWC message i it sends. After sending $\langle f, p_f, cl_f, m_f \rangle$, s_u merges the new Cl-Map cl_f into the current Cl-Map cl_i, to update its local state to reflect the changes. When s_u receives an acknowledgement from s_d, it compares cl_i with cl_{s_d}. If $cl_i \leq cl_{s_d}$, downstream site s_d has successfully merged the $p-facts$ into its state. If $cl_i > cl_{s_d}$, the site s_d has not yet merged the $p-facts$ into its state. In this case, s_u will resend the LWC message c_i to s_d. This process will be repeated until $cl_i \leq cl_{s_d}$.

The site s_u keeps a *commit log* that contains acknowledged LWC messages. If s_u knows that all downstream sites $s_1, \ldots, s_n$ have made merged all the necessary $p-facts$ into their states, it can trim the $p-facts$ from its delta-out buffer. More on this in Section 3.5.

If s_u derives multiple $p-facts$ in a single step, it will bundle them together in a single LWC message. These bundled Cl-Maps are called *composite Cl-Maps* and are not curently implemented in the system. But it is a potential future extension to the system see Section 7.2.

3.5 Lowering recovery overhead

As the system is horizontally sclaed, the introduction of each replica results in an increases the amount of data that must be exchanged to repair a site increases. Consequently, the communication overhead increases linearly with the number of replicas. This is not a significant issue when the system is relatively small, but as the system grows, the communication overhead will become a bottleneck. To address this issue, we need to minimize the amount of data that needs to be exchanged between sites to restore state. Therefore, a strategy is employed to trim the delta-out buffer of redundant data, while maintaining the fault-tolerance requirements.

The trimming algorithm is based on the concept of garbage collection. The objective is to identify the data that is no longer needed and remove it from the delta-out buffer. The algorithm is designed to be stateless and does not require any coordination between sites. Each site is responsible for trimming its own delta-out buffer. Periodically, the site will check the state of the *commit log*, to determine if it has received all the downstream sites acknowledgements. If all have acknowledged, the site will trim (discard) the associated delta-out $p - facts$ from the buffer. It does not discard the LWC records themselves, this ensures that the system can restore from a site failure without losing any data. For any site that comes back online, we can regenerate the missing $p - facts$ with the database instance, at the cost of some additional computation and time.

3.5.1 Choosing the time to trim

An efficient trimming algorithm will result in a reduction in the duration of system operations. This implies enhanced runtime performance, as there is a reduction in the disruption to processing and a faster recovery from failures, due to a reduction in the amount of data that needs to be restored. Therefore, it is crucial to determine the optimal time to trim the deltas. The time to trim is determined by the number of downstream acknowledgments received. It's inherently a trade-off between trimming too often and too seldom. Trimming too often will result in removing deltas that are still needed to restore a downstream site, while trimming too seldom will result in a large amount of data that will be stored at upstream sites. In the case of a N-N communication strategy, a site can have both multiple replicas and multiple downstream sites how can it alone determine when to trim?

The system can't be certain that all downstream sites have received the deltas, as some sites may be offline (a common state in a local-first system), and it will only have a partial or incomplete view of the system. Suppose we have a system with upstream site u and downstream sites $d_1, d_2, \ldots, d_n$. The upstream site u sends a LWC to the downstream sites, and it receives an acknowledgment from the downstream sites $d_1, d_2, \ldots d_{n-k}$ where k is the number of offline sites. The upstream site u can't be certain that the offline sites have received the $p - facts$, and it can't trim the $p - facts$ since new information from the offline sites may be needed to restore the system. In other words, the output (trim) does *not* grow monotonically with the input (acknowledgments), previously received acknowledgments may be invalidated by new acknowledgments. This is akin to the findings of Hellerstein and Alvaro [14] where they discuss the challenges of distributed garbage collection, with the conclusion that coordination-free systems can't guarantee that all nodes have received the same information.

We have chosen a probabilistic approach to solve the problem. Depending on the system configuration, we set up a policy to trim the deltas when we have received acknowledgments from around certain percentage of the downstream sites. This is based on the assumption that the system contains both edge and core sites, where the edge sites are more likely to be offline than the core sites. Once the policy is met, the upstream site can trim the deltas, and the reaming sites that have not acknowledged are regarded as offline.

4 Software Engineering Methods

This chapter will provide a brief overview of the software engineering methods employed during the thesis development process, with a particular focus on the improvements made to the development processes and code quality in PRAD.

The principles and practices of software engineering serve as the foundation for any robust development process. Software engineering is a discipline that applies engineering principles to the creation of software. The objective is to prioritize *efficiency*, *maintainability*, and *reliability* throughout the software lifecycle. These qualities are of importance for any system developed over an extended period of time. Software engineering is a broad discipline encompassing numerous approaches to software development. One of the most popular approaches is the *Agile methodology*, which was created to address the shortcomings of the traditional *Waterfall model*.

4.1 Agile

Agile methodologies, such as *Scrum* and *Kanban*, represent a set of software development practices that are focused on *incremental iterative* development, is a methodology that involves the evolution of requirements and solutions through collaboration between self-organizing and cross-functional teams. These methodologies are based on the Agile Manifesto [32], which outlines the fundamental principles of agile development as follows:

- Individuals and interactions over processes and tools.
- Working software over comprehensive documentation.
- Customer collaboration over contract negotiation.
- Responding to change over following a plan.

In the context of our project, we have implemented a modified version of the Scrum methodology. The *product backlog* consists is a prioritized list of requirements and milestones, created in collaboration with the project supervisor. Subsequently, the product backlog is then divided into *sprints*, which are defined as short cycles during which the requirements are defined and implemented. At the conclusion of each sprint, the requirements are reviewed and evaluated, and the product backlog is updated accordingly. This process is repeated until the project is completed. The project is divided into three sprints, each lasting approximately one month. The initial sprint was focused on the theoretical framework, the second to the design and experimentation and the third to the evaluation and results.

Utilizing an agile methodology has been beneficial to the project, as it has allowed us to adapt to changes in requirements and priorities. And it opens for collaboration with future developers, as the project is well-structured and tested.

4.2 Test-Driven Development

Test-Driven Development (TDD) is a software development process that relies on the repetition of a very short development cycle. Once the requirements have been transformed into highly specific failing test cases, then the software is improved to pass the new tests. Only then can the code be refactored. This process is repeated until the requirements are met. TDD is well-suited to agile methodologies, as it ensures that the code is well-tested. Additionally, TDD facilitates the design of Application Programming Interface (API)s, as it forces the developer to think about the interface before the implementation. This is especially useful for PRAD, as it is a library that will eventually be utilized by other developers.

4.3 DevOps

Development and Operations (DevOps) extends the agile methodology, by efficiently integrating development and operations. It is a set of practices that automates and links the software development process. With the goal of building, testing, and releasing software quickly and reliably. Industry leaders such as Microsoft [33] and Amazon [34] have adopted DevOps, and it has become a popular approach to software development. The DevOps approach builds on the principles of *Collaboration* between the development and operations teams, *Automation* of the software delivery process, and *Monitoring* of the performance of the development process. They are described in detail below.

4.3.1 Collaboration

In order for a team to work efficiently, they must be able to collaborate effectively. For the source code to be available, Version Control System (VCS) such as Git are used. They allow for better collaboration, versioning, branching and can scale to large projects. A popular workflow for Git is *trunk-based development*[1], where all changes are committed to a single branch, the *trunk*. With this, changes are visible for all collaborators, and it reduces the pressure of merging, and allows for more automation.

4.3.2 Automation

Automation is a key component of DevOps, as it reduces the time and cost for integration and deployment. The utilization of such tools such as containerization permits the creation of reproducible environments, which can be used for testing and deployment. Continuous Integration (CI) is a software development practice where developers regularly merge their code changes into a central repository. Automated builds and tests are then run. This approach allows for early detection of bugs and reduces the cost of integration. Continuous Deployment (CD) is an extension of CI, where the software is automatically deployed to production or the package repository after passing the automated tests.

4.3.3 Monitoring

Monitoring is the process of continuously collecting, processing, and displaying real-time information about a system. It provides information about the system

1. https://trunkbaseddevelopment.com/

such as performance, status and usage patterns of the system. This is achieved by collecting *metrics*, *events*, and *logs* from the system. *Metrics* are quantitative measurements, such as CPU usage, memory usage, and network traffic.*Events* are discrete signals that indicate that something has happened, such as a node joining or leaving the network.*Logs* are records of events that have happened in the system, such as a node crashing or a message being sent. These are typically stored in a centralized database, and can be used for debugging and analysis. In Section 6.1, we will employ monitoring techniques to assess and evaluate system's performance.

5 Implementation

This chapter will present how the approach to failure handling with replication and restoration of sites has been implemented. We explain how our approach is integrated into the existing PRAD system. We provide an overview of the rationale behind selecting Elixir as the programming language. Finally, we present the implementation of the trimming algorithm for delta outputs.

5.1 Elixir Background

In this thesis, and more specifically in PRAD, we make use of several language-specific features of the Elixir programming language, which renders it an optimal choice for the development of large distributed systems. In light of the aforementioned considerations, it is beneficial to provide a brief overview of Elixir and the Erlang Virtual Machine, which it runs on.

Elixir [35] is a dynamically typed and concurrent programming language that adheres to the principles of functional programming. This enables the rapid prototyping of highly concurrent systems. It integrates elements from Erlang, Clojure and Ruby to create a language that is both expressive and performant. Its syntax is inspired by that of Ruby, and it makes use of a number of the same features, including *metaprogramming* and *pattern matching*. In contrast to programming languages such as Python and Rust, which employ Async-Await to achieve concurrency, Elixir utilizes a share-nothing *actor* model. These and other features derive from Erlang, which is the reason for now introducing the Erlang programming language.

Erlang [36] is a programming language and platform designed for building reliable and scalable systems. The language was initially developed by Joe Armstrong, Robert Virding and Mike Williams at Ericsson in the 1980s, with the intention of being used in telecommunication systems. Telecommunication systems are inherently distributed, and must be always online, even in the face of unexpected errors or software upgrades. The non-functional requirements of high availability and fault tolerance remain highly relevant in the present day. Erlang is used to build large-scale distributed systems, including those used by WhatsApp [5], Discord [4] and RabbitMQ [37].

5.1.1 BEAM & OTP

The *Erlang Virtual machine*, or Bogdan/Björn's Erlang Abstract Machine (BEAM) is the virtual machine that runs Erlang and Elixir code. It utilizes its own garbage collector, scheduler and memory allocator to distribute the Erlang processes across available CPU cores.

The fundamental unit of concurrency in Erlang is the *Erlang process*. In contrast to operating system processes, Erlang processes are notably lightweight and can be created and destroyed quickly. This capability enables the Erlang runtime to schedule millions of processes on a single machine. The processes are scheduled by the Erlang Run-Time System (ERTS) [38], and are preemptively scheduled. This implies that can continue to execute until they yield control to the scheduler.

Additionally, Erlang processes are *share-nothing*, meaning that they cannot share memory with other processes. In the event of a process crash, it will not affect any other processes, and the runtime will simply destroy the process and reclaim the memory as described in [38]. This contrasts with operating system processes, where a crash can affect other processes and the operating system itself. This enables a 'Let it cash' philosophy, where the programmer does is not required to handle every possible error; instead, the process can be allowed to crash, and a new process can be initiated instead. This is particularly advantageous in distributed systems, where errors are inevitable and difficult to handle. Furthermore, it enables the use of the *actor model* of concurrency, where each process is an actor that communicates with other actors by sending messages.

As Erlang processes are share-nothing, they communicate by asynchronous message-passing. This implies that they do not necessitate the use of intricate synchronization primitives such as locks or mutexes. The message-passing interface between processes is the identical regardless of whether the processes are local or remote. This enables the same code to be executed on a single machine or across a cluster of machines.

The Open Telecom Platform (OTP) is a set of libraries and tools for building distributed systems in Erlang and Elixir. It provides a set of *behaviours* for building fault-tolerant systems, such as *gen-server*. A *behaviour* is generic a code module that implements a set of common functionality. This functionality is made available through a set of callbacks, which the user implements in order to adapt the code to their specific requirements code. This enables the user to focus on the domain logic, while the generic Open Telecom Platform (OTP) code handles the common functionality. The ease of adopting *GenServer* behaviours is one of the main reasons we choose to implement PRAD in Elixir. This allows us to focus on the domain logic of the system instead of the common functionality. *GenServers* provide both a synchronous and asynchronous request-response model, named *call* and *cast* respectively. This allows for a flexible communication model between the processes.

The Erlang Term Storage (ETS) is a built-in database in the Erlang runtime that can be utilized to store considerable amounts of data in-memory and to share state between Erlang processes. It is highly optimized for concurrent access, and can be used to store millions of entries. It is useful for *key-value* stores or counters, such as in the *Cl-set*. PRAD uses ETS to store the EDB and IDB tables of the Datalog program.

5.2 PRAD Extension for Failure Handling

The implementation extends the existing PRAD [6] system with the ability to repair the execution of PRAD programs by recovering from site failures. The implementation has been developed in Elixir as a set of modules, which have been wrapped in an API that provides the necessary failure handling functionality. The currently supported modules in the Elixir implementation are `Replication` and `Restoration`. The `Replication` module is responsible for dynamically replicating the state of one site to a new replica. The `Restoration` module is responsible for restoring the state of a site that has been offline for a period of time. Both modules are implemented as `GenServer` processes in Elixir, which allows for the handling of stateful processes (actors).

The modules are responsible for repairing the state of a crashed, stopped or offline site in the system. As previously discussed in Section 3.4, the objective is to populate the new site with the relation(s) from an existing site. To achieve this, the new site must set up a minimal skeleton state, which requires the instantiation of a configuration, a relation or relations, and a Cl-Map. These are the fundamental elements of a site in PRAD.

Listing 5.1: Example Datalog Program

```
BookPr(T, Y) :- Book(T, Y, A).
```

Suppose we have upstream sites s_1, s_2 with relation the *Book* and downstream sites s_3 and s_4 with the relation *BookPr*. Among these sites, s_3 has experienced a crash and is offline. All of the aforementioned sites are part of the same program, as seen in Listing 5.1. An example of the state of s_1 and s_3 can be seen in Listing 5.2 and Listing 5.3 respectively, taken immediately following the crash of s_3.

Listing 5.2: Example of complete Site 1 state.

```
{
  "name": "s1",
  "configs": [":book"]
  "relations": {
    [":book_s1"]
  },
  "replicas": [":s2"]
  "schema": {[title: :text, year: :num, author: :
     text], [1, 3]}
  "delta": [{"Elixir", 2024, "Author", %{prov...}}]
  "cl": %{...}
  "commit_log": {...}
  "is_on": true
}
```

As illustrated in Listing 5.2, the complete state of s_1 is a JavaScript Object Notation (JSON) object that contains the site's name, configuration, relations, replicas, schema, delta, Cl-Map and a boolean flag *is_on*. While the state of s_3 in Listing 5.3 is largely consistent with that of s_1 with the exception of the *is_on* flag, which has been set to *false* to indicate that the site is offline. Important for the repair process are the *configs*, *replicas*, *cl* and *delta* fields. The *configs* field contains the relations that the site is responsible for, the *replicas* field contains the *names* of the replica sites, the *cl* field contains the Causal Length of the site and the *delta* field contains the updates that the site has received.

The Cl-Map is implemented using a map data structure in Elixir, where the keys are the relation names and the values are sub-maps containing the UUIDs and their associated Causal Lengths. While the commit log utilizes the Erlang

:ORDSET [1] module to store the UUIDs of the commits, allowing for efficient insertion and deletion of unique commits.

Listing 5.3: Example of complete Site 3 state.

```
{
  "name": "s3",
  "configs": [":book_pr"]
  "relations": {
    [":book_pr_s3"]
  },
  "replicas": [:s4]
  "schema": {[title: :text, year: :num], [1, 2]}
  "delta": [{"Elixir", 2024, %{prov...}}]
  "cl": %{...}
  "commit_log": {...}
  "is_on": false
}
```

The process of establishing a skeleton site is initiated by invoking the GENSERVER INIT behavior in Elixir, which is called when a new process is started. The `init` function is responsible for initializing the state of the process. The state, which is represented as a map, includes the site's unique *id* and its configuration. The configuration comprises the relations (EDBs and IDBs) that the site is responsible for, the schemas of the relations and the *Rules* defined by the current program. A common feature of both the `Replication` and `Restoration` modules is that we provide the *id* of the site from which we wish to replicate or restore from. With the *id* of the source site, we can fetch the configuration using a `call` to the source site. Based on the source configuration, we create the required relations either as EDBs or IDBs, and instantiate them at the replica. Finally, we create an empty Cl-Map based on the Site *id*. Once the skeleton state has been established, the replication or restoration process can be initiated.

5.2.1 Basic Replication

Our implementation of the restore and replication mechanisms underwent several iterations. In the initial iteration, we used a replicate (full restore) approach, where the source site would send the relations to the new replica. This approach is illustrated in Listing 5.4 Once the site skeleton has been established, we contact the existing site to pull the relations from it. The

1. https://www.erlang.org/doc/apps/stdlib/ordsets.html

existing site sends the required relations to the new replica. Subsequently, the new replica then stores the relations in their respective states and updates the Cl-Map accordingly. At this point, the new replica is prepared to receive new updates.

Listing 5.4: Minimal example of a basic repair using replication.

```
defmodule Site do
  use GenServer
  # Common intialization for both replication and
     restoration.
  def init(args) do
    %{id: args[:id], relations: %{}, cl_map: %{}}
  end

  def replicate(new, source) do
    call({:init})
    config = fetch_config(source, new)
    create_relations(config)
    pull_relations(source)
    update_cl()
  end
end
```

The full replication approach is costly due to the necessity of transmitting a significant amount of data, even in the event that sites possess some common relations. This is particularly problematic in the case of partial repairs or restores, where only a subset of the relations is required. To address this issue, we developed a stateless restore mechanism, which is more efficient and only requires the sites to transmit the missing relations. The stateless restore mechanism is discussed in the following subsection.

5.2.2 Stateless Restore

A more viable approach is to utilize the Cl-Map to calculate the difference between the two maps and only transmit the missing relations. After receiving the diff, the new site can apply the changes to its state and update the Cl-Map accordingly.

The diff algorithm is composed of two primary functions that operate on Cl-Maps. The keys represent relations, while the values are sub-maps containing UUIDs and their associated Causal Length. The diff algorithm is illustrated in Algorithm 1. The algorithm takes two Cl-Maps as input, the current state map

CL and the delta state map CL_Δ. The algorithm iterates over each relation in the delta state map and compares the Causal Lengths of the UUIDs in the sub-maps. If the Causal Length in the delta state map is greater than the Causal Length in the current state map, the UUID is added to the diff. This ensures that only the facts that are not present in the target map are added to the diff. The diff is then returned as the output of the algorithm.

Algorithm 1 Diff Algorithm for Cl-Maps

```
1:  Input: Current State Map $CL$, Delta State Map $CL_\Delta$
2:  Output: Difference Map $D$
3:  procedure DIFF($CL$, $CL_\Delta$)
4:      $D \leftarrow \{\}$
5:      for all $r \in$ Relations($CL_\Delta$) do
6:          $CL_r \leftarrow$ Get($CL, r, \{\}$)
7:          $CL_{\Delta r} \leftarrow$ Get($CL_\Delta, r$)
8:          $\delta \leftarrow$ DiffInRel($CL_r, CL_{\Delta r}$)
9:          if $\delta \neq \{\}$ then
10:             $D \leftarrow D \cup \{r \mapsto \delta\}$
11:         end if
12:     end for
13:     return $D$
14: end procedure
15: procedure DIFFINREL($CL$, $CL_\Delta$)
16:     $D_r \leftarrow \{\}$
17:     for all $u \in$ Keys($CL_\Delta$) do
18:         $l \leftarrow CL_\Delta(u)$
19:         if $CL(u) < l$ then
20:             $D_r \leftarrow D_r \cup \{u \mapsto l\}$
21:         end if
22:     end for
23:     return $D_r$
24: end procedure
```

5.2.3 Lightweight Commit Generation

A third approach, currently only for the restoration mechanism, is to use Lightweight Commit (LWC)s, to handle the restoration of a site with multiple upstream sites. It builds on the same initial setup as the stateless restore, but instead of relying on the Cl-Map to calculate the difference, we use the LWCs to determine when all upstream sites have sent their updates. Suppose we have an upstream site u, and a downstream site d. The upstream site u sends a sequence of *p-facts* followed by a LWC. As described in Section 3.4.3, the interval of the LWC is determined by the queue length. The downstream site *d* receives the *p-facts* and merges them to its local Cl-Map. When the downstream site *d* receives the LWC, it sends an acknowledgment back to the upstream site *u* if and only if it has merged all necessary *p-facts*. *d* can verify this by comparing the Causal Length of the *p-facts* with the Causal Length in the Cl-Map. If the Causal Length in the Cl-Map is less than the Causal Length of the *p-facts*, the *p-facts* have not been merged, and the LWC is ignored. Currently, the LWC mechanism is only available insertions as deletions require a more complex mechanism to handle the removal of facts, to keep the monotonicity of the system.

LWCs are generated automatically by the system or manually added to the delta output via an API. In either case, however, the generation of LWCs is contingent upon the occurrence of *queue points*. A queue point is defined as a point in the execution where the system is in a stable state, which can be used as a reference point for the generation of LWCs. The system automatically generates queue points based on the user-defined *queue length* parameter. The queue length parameter specifies the number of updates that can be sent before a queue point is generated.

The queue length parameter allows for the fine-tuning off the amount of state that needs to be stored and the efficiency of the restoration process. A queue length that is too short will result in numerous queue points and LWCs that could temporarily saturate the network. Conversely, a queue length that is too long will result in a significant amount of state that needs to be stored. The optimal queue length is contingent upon the specific use case and the requirements of the system. Consequently, there is an inherent trade-off between the overhead in memory usage by storing more in the queue, and reducing the latency of the restoration process.

5.2.4 Repair API Usage

Listing 5.5 shows the usage of the Replication and Restoration APIs. In this example, we have four sites, `:s1`, `:s2`, `:s3`, `:s4` and `:s5`. Site 1 crashes and Site 2 goes offline and the fault tolerance degree of the system is reduced. To repair the fault tolerance degree, we create a new site, `:s5`, which is a replica of site 1. We use site 2 as the source of the replication. The state of site 5 is now the same as site 1, and the fault tolerance degree of the system is restored. Here we see that the replication mechanism extends the system static configuration, we can instantiate new replica sites on-the-fly and dynamically change the number of replicas for a given site.

Listing 5.5: Example of the Replication and Restoration APIs

```
def Prad.Example do
  # Initialize the sites.
  sites = [:s1, :s2, :s3, :s4, :s5]

  # Configuration for the sites.
  upstream_schema = [:s1, :s2, :s3]
  downstream_schema = [:s4, :s5]

  # Sites 1 crashes.
  Site.crash(:s1)

  # Site 2 goes offline.
  Site.stop(:s2)

  # We create a replica of site 1 using site 3.
  :s6 = Site.replicate(:s1, :s3)

  # We restore site 2 using site 3.
  :s2 = Site.restore(:s2, :s3)

  # Sites 2 and 6, can now be used as before.
end
```

All three approaches currently use a synchronous communication mechanism `call`, which means that the calling process will block until the receiver has processed the message. This could have notable performance and scalability implications, this and the different ramifications of each approach will be discussed more in-depth in the Discussion Section 7.2.2.

5.3 Trimming Algorithm

The implementation of the trimming algorithm presented in Section 3.5 is built on top of the GenServer process abstraction in Elixir. The trimming algorithm is presented in Algorithm 2. Each site in the system is represented as a GenServer, which is equipped with a local trim process. Besides the synchronous and asynchronous communication protocols (call and cast), GenServer provides a set of 'regular' callbacks that define the sending and receiving of messages. To support the periodic nature of the trimming algorithm, especially the `send_after/4` function is of interest. It allows us to send a message to the GenServer process after a specified time interval, and handle the message in the asynchronous `handle_info` callback, which is akin to an Async/Await pattern in other programming languages. This allows us to avoid blocking the process while waiting for the next trimming operation in idle periods.

In addition, if no new LWCs are acknowledged, an exponentially back off is implemented using the aforementioned `send_after/4`. The backoff period is determined by the *trimming interval* parameter, which specifies the time between each trimming operation. With a maximum backoff period of 10 minutes, the algorithm will gradually increase the time between each trimming operation until a new LWC is acknowledged. This ensures that the system is not burdened with unnecessary trimming operations when there is no new data to be removed.

Algorithm 2 Trimming Algorithm for Delta Outputs

```
 1: Input: State state, min_timeout := 1000, max_timeout := 10000,
    queue_size := 10
 2: Output: Trimmed updated state
 3: procedure HandleInfo(trim, state)
 4:     (new_deltas, trim_performed) ← PerformTrim(state)
 5:     if trim_performed then
 6:         timeout ← 1
 7:     else if min_timeout * (state.timeout * 2) > max_timeout then
 8:         timeout ← max_timeout
 9:     else
10:         ScheduleTrim(min_timeout * (state.timeout * 2))
11:         timeout ← state.timeout * 2
12:     end if
13:     ScheduleTrim(min_timeout)
14:     return state with updated delta and timeout
15: end procedure
16: procedure ScheduleTrim(interval)
17:     Send :trim message after interval
18: end procedure
```

6 Experiments

This chapter presents the experiments conducted to evaluate the implemented approach. We start by describing the experimental setup, including the hardware and software used. We present the two scenarios, repair and restore, and demonstrates their effect on performance, storage, and communication. We then proceed on to evaluate the lightweight commit cost. Finally, the impact of trimming algorithm on the system is evaluated.

6.1 Experimental Setup

We have created a set of experiments to evaluate our expansions of the PRAD system in terms of performance and scalability. The experiments are implemented and measured using our monitoring system, which is described in Section 4.3.3. The monitoring system enables the measurement of the CPU usage, memory usage, garbage collection, and the number of processes, with minimal overhead.

The experiments were conducted using Elixir [1] version 1.15.7, compiled with Erlang OTP [2] 26.1.2. The PRAD fault-tolerance extension is implemented using *GenServers* for the sites and ETS for data storage. In order to study the

1. https://elixir-lang.org
2. https://www.erlang.org/

performance of the system in a more realistic distributed environment and to overcome the limitations of the shared resources on a local machine, we have added delays simulating network latency and processing time of Database operations. The delays are introduced at the *GenServer* level, and can be configured. For each message transmitted between two sites, a $10ms$ delay was introduced, and for each search conducted on a table $1\mu s$ for every 10 rows processed.

The following hardware was used to run the benchmarks:

- **CPU:** Apple M1 CPU with 8 (4 power and 4 efficiency) cores @ 3.2 GHz
- **MEMORY:** 16 GB LPDDR4 @ 4267 MHz
- **BANDWIDTH:** 68 GB/s
- **STORAGE:** 256 GB PCIe SSD with 3.3 GB/s read and 2.8 GB/s write

Due to the manual insertion of delays, the hardware should not have a significant impact on the results.

The core of the experiment comprises two classical *Datalog* programs for calculating a *transitive closure*, for the two operations *Project* and *Join*. In each experiment, we insert 50, 150, 300, 500, 1000 and 1500 *Book* and *Author* facts for each execution. We then measure the total runtime it takes to derive all new *BookPr* and *Release* facts from the inserted facts. The facts are derived using a recursive Datalog rule, which is applied until a fixed point is reached. The *Project* program is presented in Listing 6.1, while the *Join* program is shown in Listing 6.2.

Listing 6.1: Project Datalog Program

```
BookPr(T, Y) :- Book(T, Y, A).
```

Listing 6.2: Join Datalog Program.

```
Release(T, N) :- Book(I, T, Y, A), Author(A, N).
```

6.2 Repair Time

In the initial experiment, we measure the repair time of an offline or crashed site replicating a peer site, in terms of run-time and memory usage. The run time here is the time it takes to repair the site, from offline to online and including the time to derive all new facts. We then compare the repair time with a restoration of the same site from an upstream site, under the same conditions meaning that 100% of the facts are inserted before it goes offline. In both cases a second site is not going offline, and used to validate the repair or restore operation. The *Project* program is used, with a replication and partitioning factor of 2 for both the *Book* and *BookPr* relations. To circumvent potential anomalies, the experiment is conducted three times and the mean runtime is calculated.

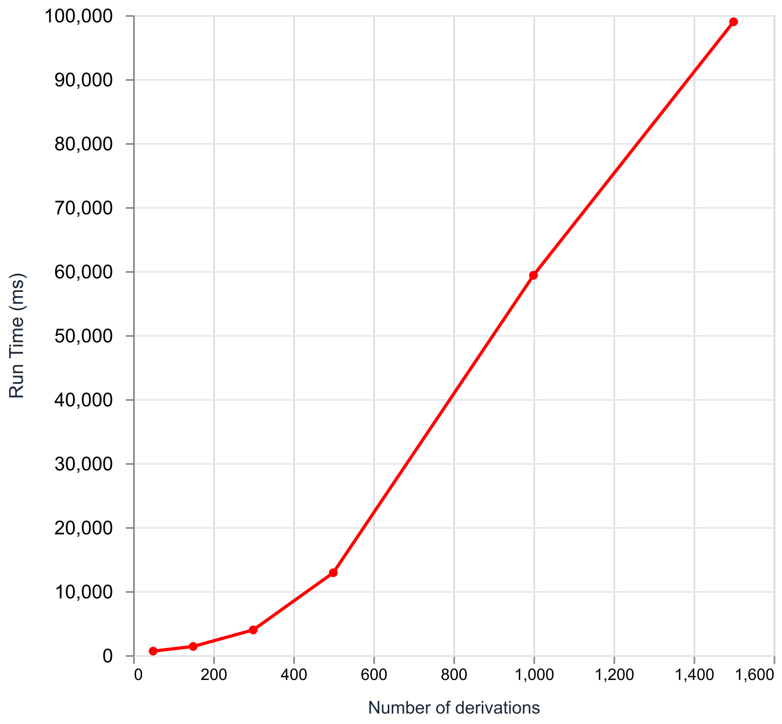

Figure 6.1: Time to repair using Replication.

Figure 6.1 shows the results of the repair using replication. We can see that the run time of repairing small sites with 50 to 300 derivations is quite low, with a run time of under 5 seconds. However, as the number of rows increases, the run time increases exponentially, at 500 derivations the run time is still 20 seconds, but it increases to 100 seconds at 1500 derivations. This is likely due to the increased number of messages sent As previously stated in Section 5.1.1, a GenServer (or any other process) runs concurrently, but its internal mailbox is synchronous. The mailbox is a FIFO queue, which means that the messages are processed in the order they are received. As the number of messages in the mailbox increase, the process will have to go through all the messages in the mailbox to check if they match one of the RECEIVE patterns. We confirm this suspicion by checking the number of messages in the mailbox using PROCESS.INFO(:MESSAGES_QUEUE_LENGHT) [38], which indeed increases drastically with the number of derivations.

The behavior is reflected in the ETS memory usage, which is shown in Figure 6.2. We can see that the memory usage is increasing with the number of derivations, and peaks at 33 MB at 1500 derivations. This is consistent with that each fact is around 15 KB to 20 KB in size depending on the number of attributes. In Figure 6.3, we can see that the size of the site state is increasing with the number of derivations with at peak at 3000 KB, as expected. As the site state contains metadata about the derivations. This explains some increased run time, as the state needs to be updated and sent for each derivation.

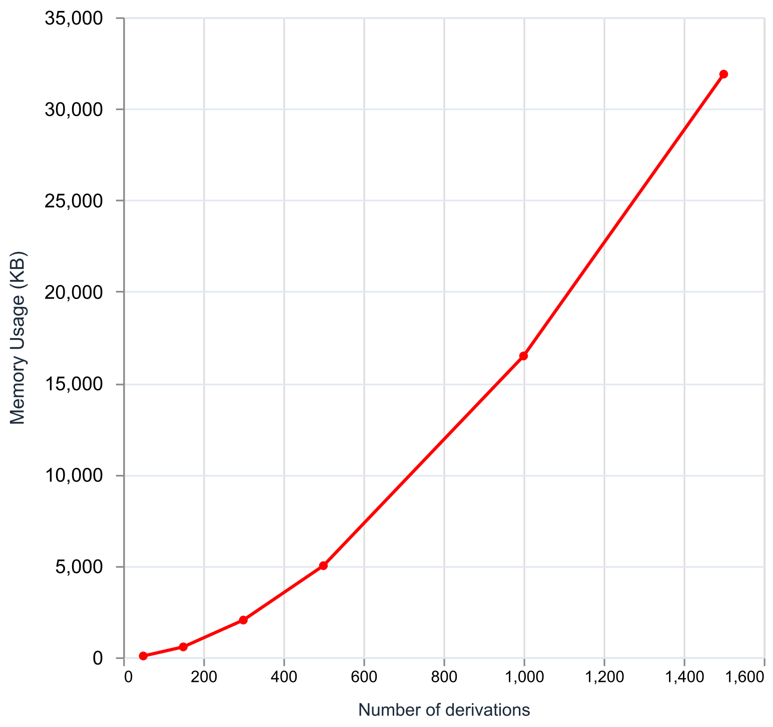

Figure 6.2: Memory usage for repair using Replication.

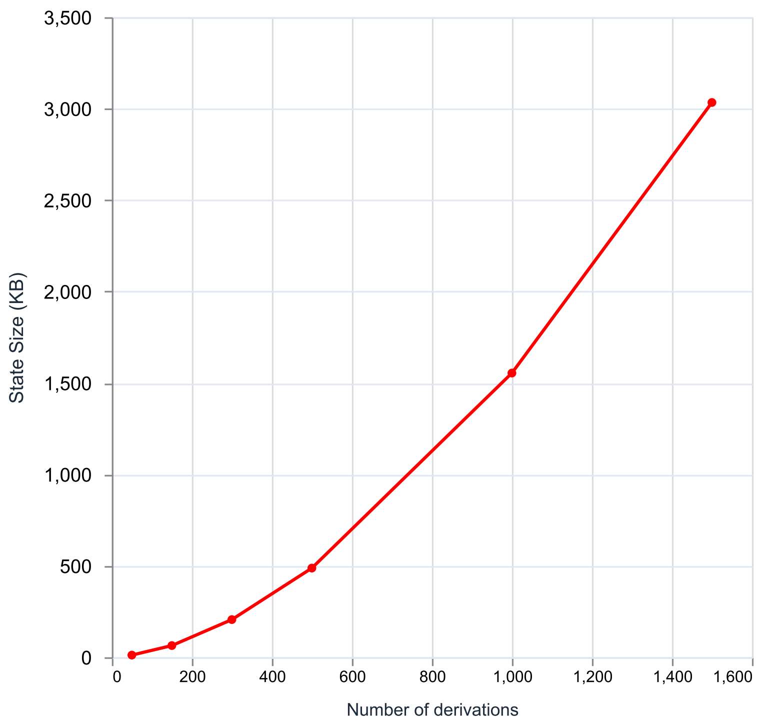

Figure 6.3: State size for repair using Replication.

As described in Section 4.3.3 monitoring is used to gain insights into the performance of the system. These are used to better understand the system behavior, and to detect performance regressions. Below we present the monitoring results for the repair operation using replication.

Next, the Active Process's and the Run Queue length are measured and presented in Figure 6.4. We can observe that the number of processes is on the rise with each run, reaching a maximum at around 600 active processes. Which is well within the limits of the Bogdan/Björn's Erlang Abstract Machine (BEAM), which can handle up to 2000 reductions per scheduler [38]. The length of the Run Queue is also increasing, but from the graphs it appears that *task stealing* is able to keep up, and the queue length per process is kept small.

Subsequently, we take a look at the memory usage, which is shown in Figure 6.5. We can observe that the memory usage, particularly for ETS stands out, and has a steady increase with the number of replicas. It peaks at 2.8 GB total usage, which is consistent with expectations for 1500 table entries. The next step is to examine the impact of garbage collection as the number of tables increases.

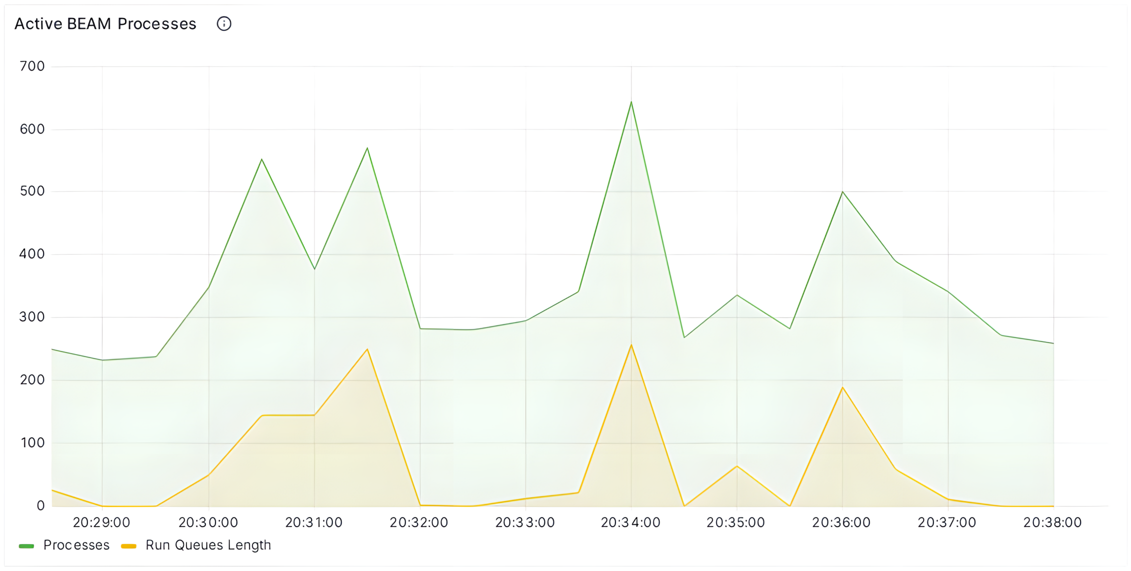

Figure 6.4: Active Processes with Replication.

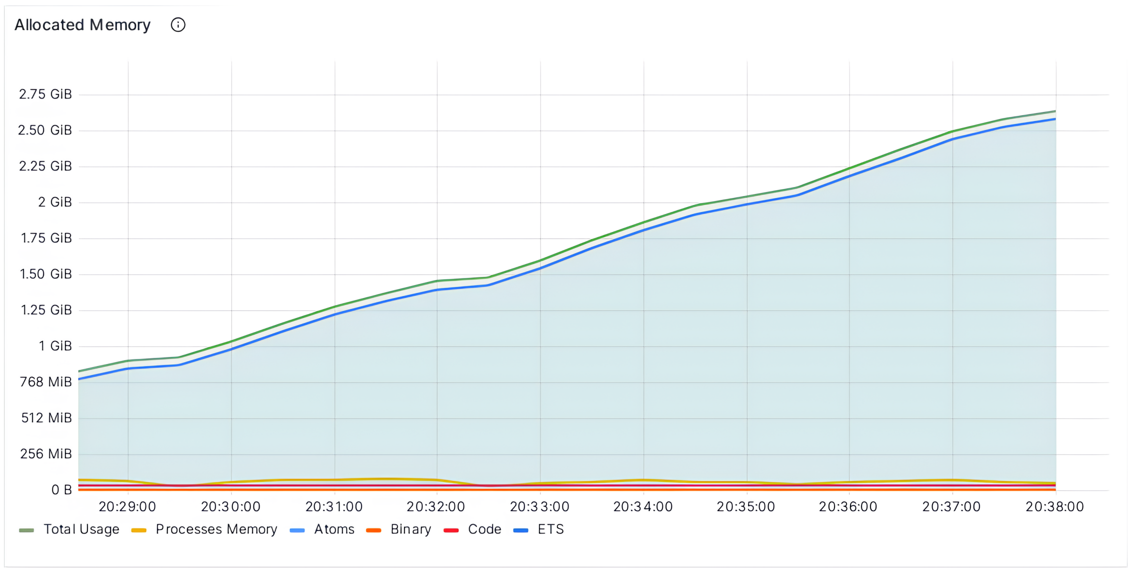

Figure 6.5: Memory Usage with Replication.

Finally, we take a look at the garbage collection, which is shown in Figure 6.6. We can observe that the collector peaks at 800 MB, which is roughly 30% of the total memory usage. Moreover, we can see that the spikes are analogous to the active processes, which suggests that the garbage collector is triggered when the number of processes increases and functions as intended.

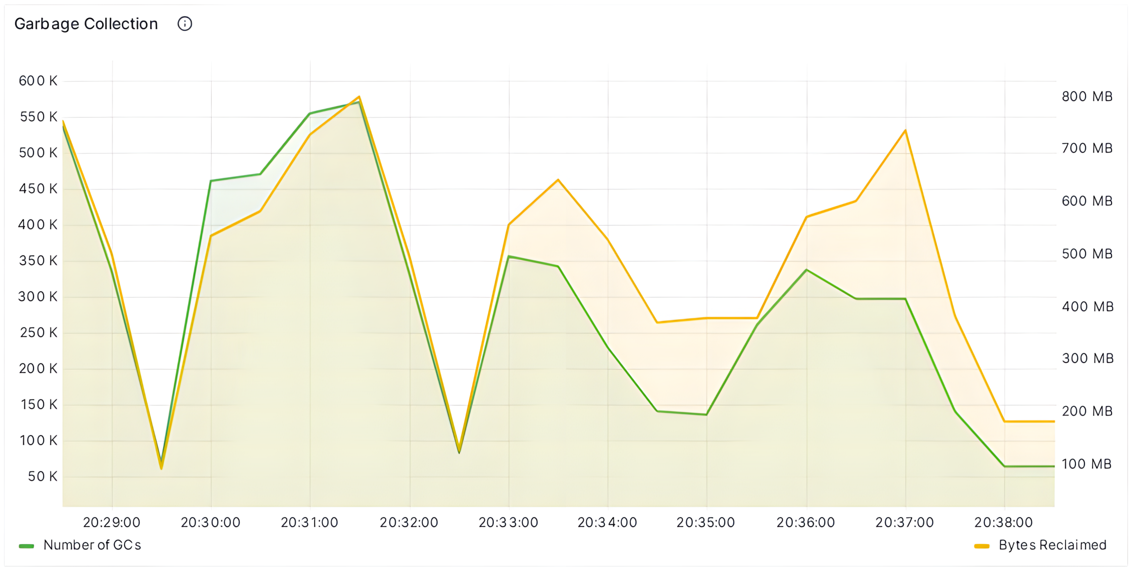

Figure 6.6: Garbage Collection with Replication.

Figure 6.7 shows the comparison between replication and restoration for the repair operation. From the graph, we can see that a full restoration of site is equally fast as using replication, with a slight increase in run time for 1500 derivations and a decrease at 1000 derivations. Which both are within the margin of error. In regard to memory usage, we see no difference between the two methods, which is as expected, as the same amount of data is transmitted in both cases. Meaning that in the case of a full-repair, it is not worth to restore the site and just replicate it. Possible solutions to the run time overhead are outlined later in the discussion Section 7.2.2.

Because the delay is introduced at the GenServer level, it will also be triggered on local messages. This is not ideal, as there should be no delay on local messages in a realistic environment. Therefore, we run the experiment with a delay of 0, to check the implementation. The results are shown in Figure 6.8. We can see that the run time is much lower, with a peak at 20 seconds at 1500 derivations, which is half of the run time with a delay. Nevertheless, we still see the same exponential increase in run time, which indicates that the delay is not the main factor for the increased run time.

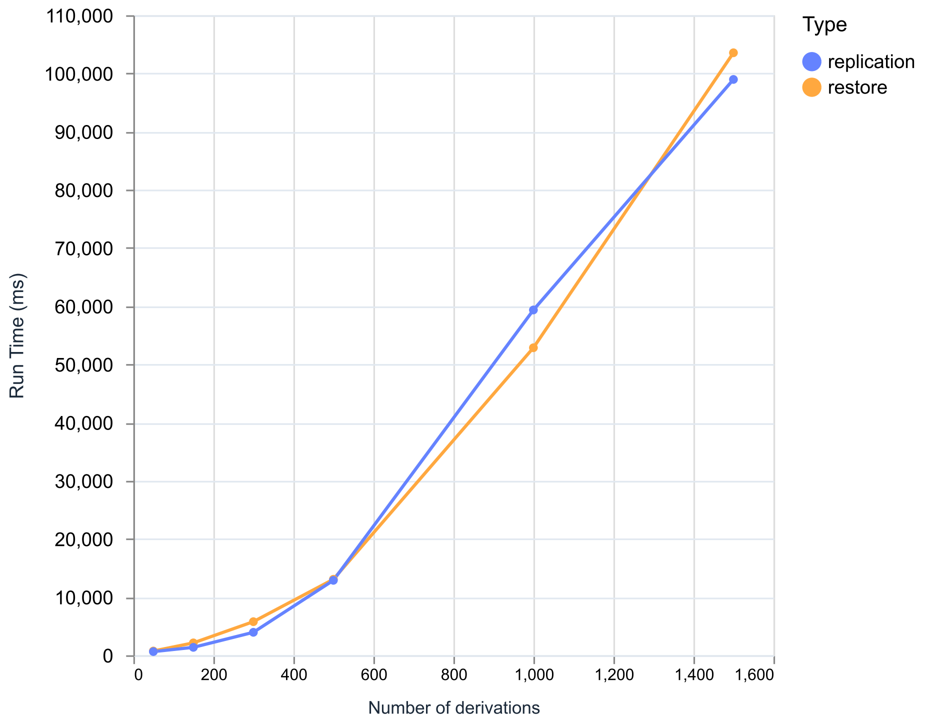

Figure 6.7: Time to repair using Restoration and Replication.

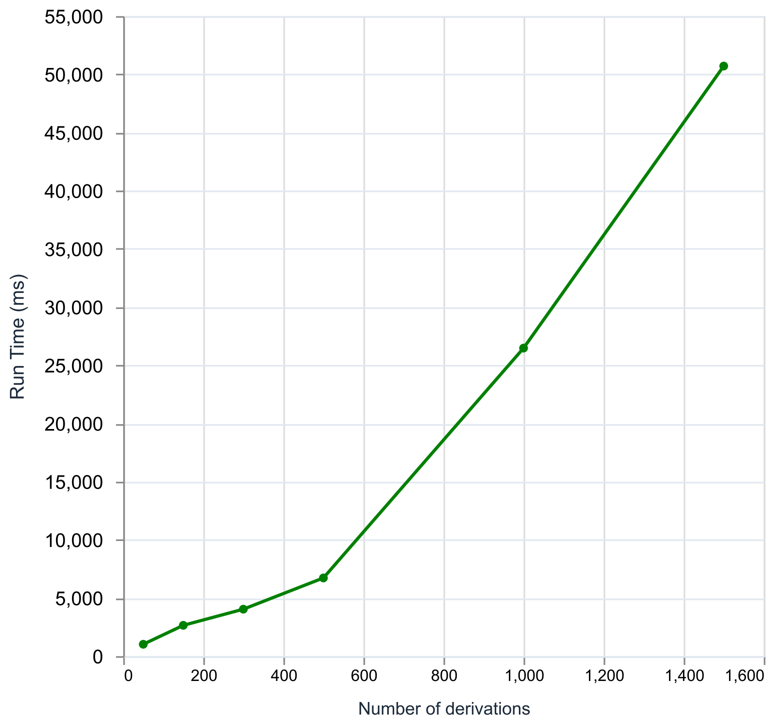

Figure 6.8: Time to repair using Replication (No delay).

6.3 Repair using Cl-Map

We evaluate the performance of the stateless restore utilizing Cl-Maps for repair, as outlined in Section 3.4.2. We use the same setup as in the previous experiment, with a replication and partitioning factor of 2. We measure the runtime of the repair operation for 100, 200, 300, 400, 500, 1000 and 1500 rows, with both the *Project* and *Join* programs.

First we look at the *Project* program, which is shown in Figure 6.9. We can see that the run time closely follows the same pattern as the previous experiment, with a slight increase at 1000 derivations. However, the state is considerably smaller, with a peak at 640 KB at 1500 derivations compared to 3000 KB using replication. This is due to only the Cl-Map being transmitted, which is considerably smaller than the full state. However, the run time remains high, which indicates that the differences' calculation takes a considerable amount of time.

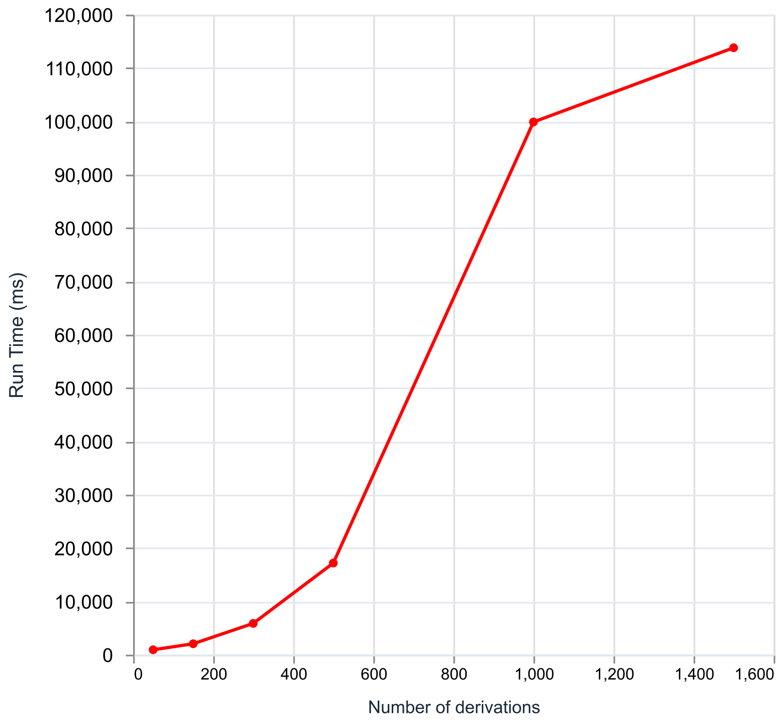

Figure 6.9: Time to repair using Stateless Restore (Project).

Then we take a look at the *Join* program, which is shown in Figure 6.10. We can see that the run time is a bit lower than for the *Project* program, with a peak at 90 seconds at 1500 derivations. And the state size is also smaller, with a peak at 300 KB at 1500 derivations. This is likely due to the fact that we in the case of *join* are restoring from an intermediate relation, which is smaller than the full state.

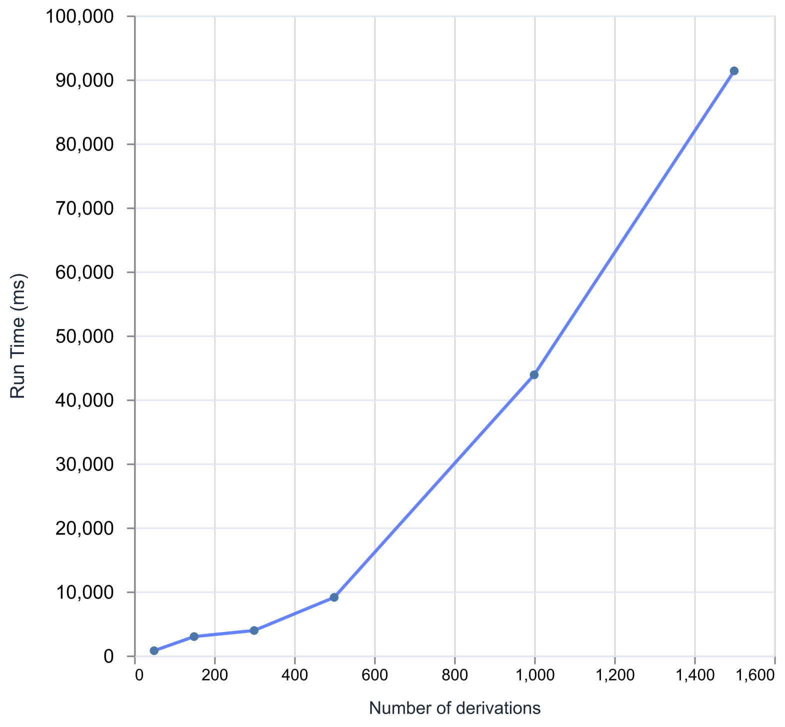

Figure 6.10: Time to repair using Stateless Restore (Join).

The results of the *Project* and *Join* programs indicate that the stateless restore is comparable to the full replication, while being more in line with PRADs coordination-free approach. Additionally, the state size is smaller, which is beneficial for the memory usage. Consequently, we now turn our attention to the final approach, namely the Lightweight Commit (LWC) approach for repair.

6.4 Lightweight Commit Cost

The LWC approach is evaluated using the same experimental setup as the previous experiments, with a replication and partitioning factor of 2. We measure the runtime of the repair operation for 100, 200, 300, 400, 500, 1000 and 1500 rows, with the same exact *Project* program as in Section 6.3. The results are presented in Figure 6.11. We can see that the run time is considerably reduced in comparison to the stateless restore, with a peak at 40 seconds at 1500 derivations, which is less than half of the run time for the stateless restore. The state size is about the same as the first approach, but with much faster operations at the sites. This is likely due to the fact that the LWC approach is more lightweight, and only sends the differences in commits between the sites, which is much smaller than the full difference. This evidence indicates that the LWC approach is a good compromise between the two approaches, with a lower run time and memory usage, while still being coordination-free.

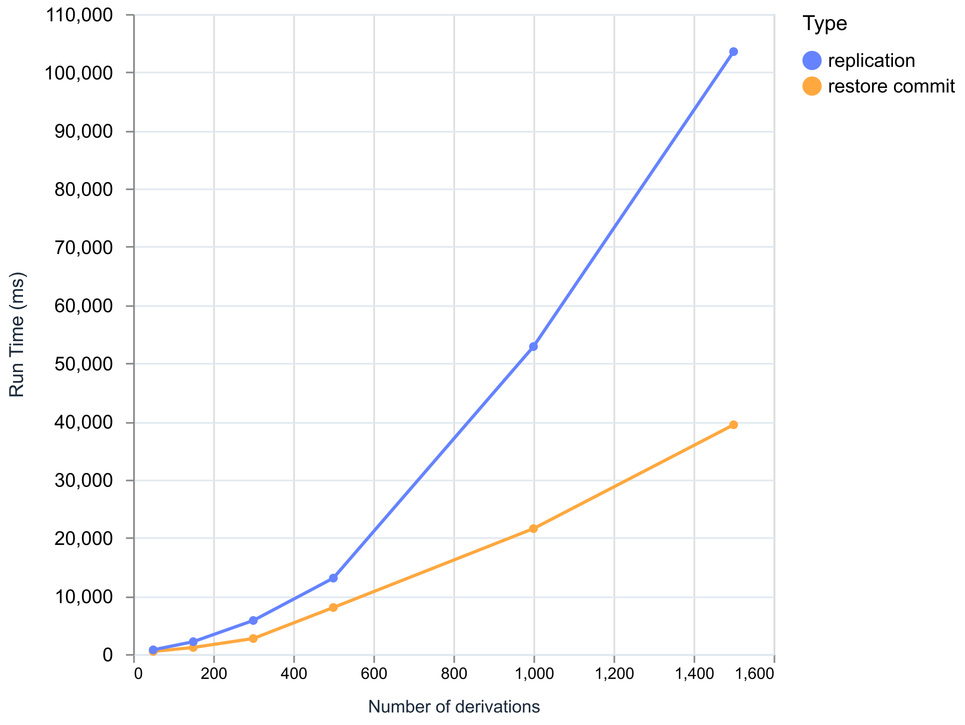

Figure 6.11: Time to repair using LWC (Project).

6.5 Impact of Delta Trimming

We compare the full replica repair run time with trim and without trim. This is illustrated in Figure 6.12. We can observe that the run time is considerably reduced with trim, with a peak at 10 seconds at 1500 derivations, which represents approximately 20% of the run time without trim. This is likely due to the fact that the trim has removed the unnecessary deltas, which are not required for the repair operation. The assertions are corroborated by verifying that the trimmed state of both sites is identical after the repair operation. This indicates that the trimmed site has been replicated correctly and that the trim operation is working as intended.

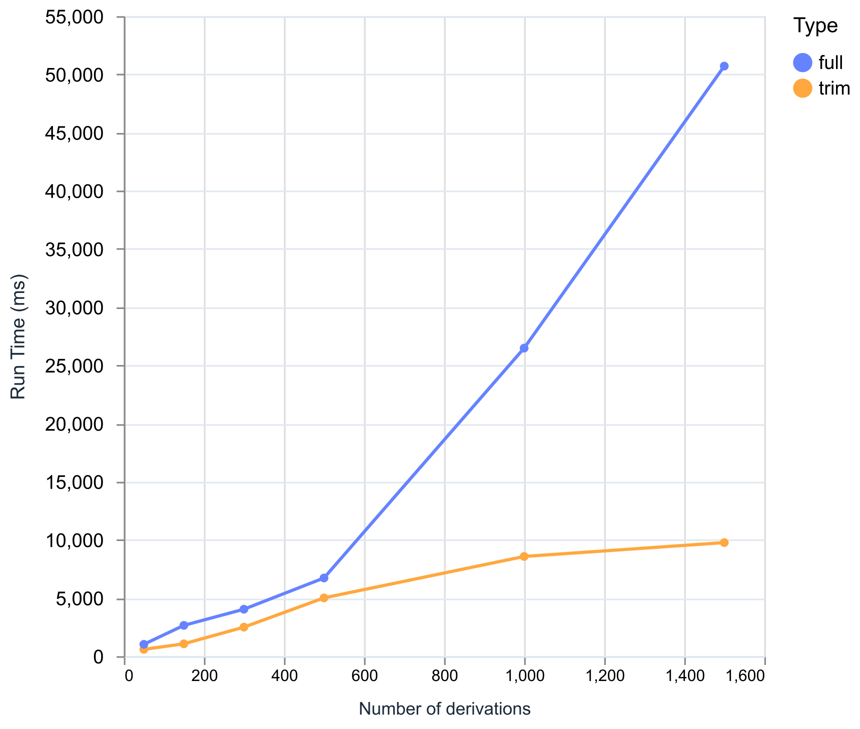

Figure 6.12: Trim versus no trim (Full Replication).

Subsequently, we evaluate the impact of delta trimming on the system. We use the same setup as in the previous experiments, with a replication and partitioning factor of 2. We measure the time to trim deltas, with 100, 200, 300, 400, 500, 1000 and 1500 facts, while running the *Project* program.

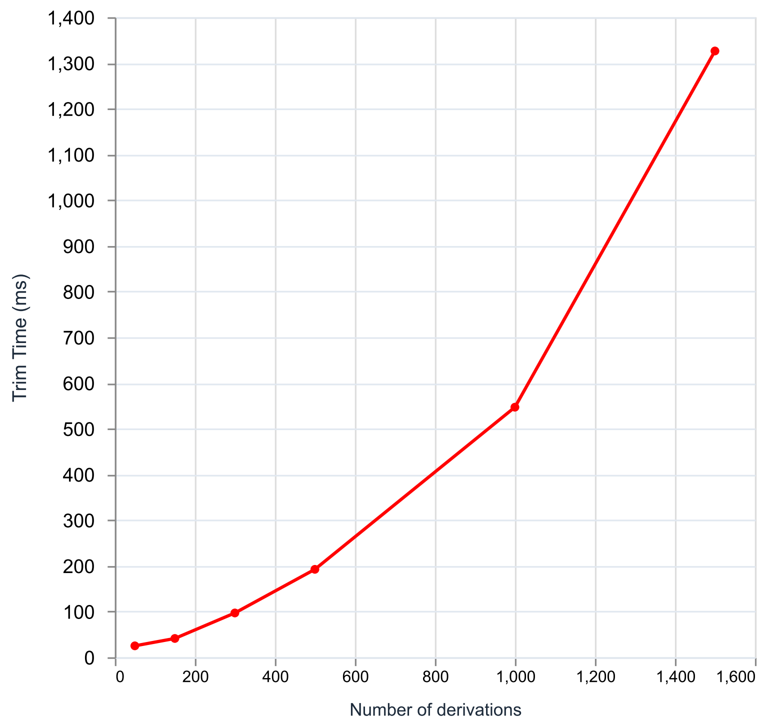

Figure 6.13: Time to trim (Full).

The results are shown in Figure 6.13. We can see that the run time is quite low, with a peak at 1 seconds at 1500 derivations, this includes the time to derive the new facts. In comparison to the repair operation, trimming is run asynchronously, which means that the system can continue to process messages while the trimming is ongoing. Furthermore, we can validate that it trims the correct deltas by checking the memory it collects, which is shown in Figure 6.14. Deltas are only a fraction of the size of full derivations in the ETS tables, which is why the memory reclaimed is much lower at only 1.5 MB at 1500 derivations.

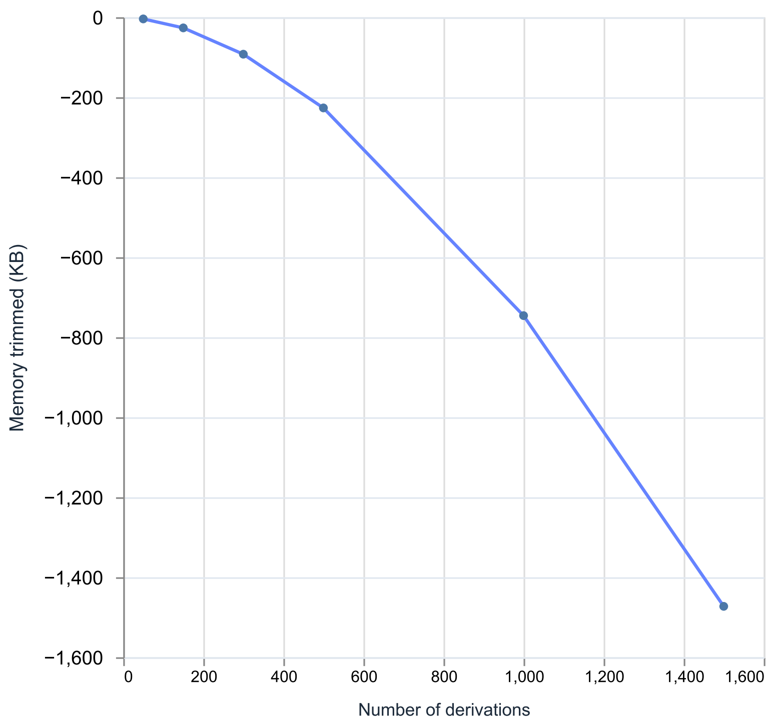

Figure 6.14: Time to trim (Full).

When comparing the reclaimed memory of the full trim to the partial trim, we can see that it has a more linear decrease, which is shown in Figure 6.15. This is likely due to the fact that the BEAM garbage collector is interfering with the trimming, which is not the case for the partial trim. Because the partial trim does not run long enough for the generation to be promoted.

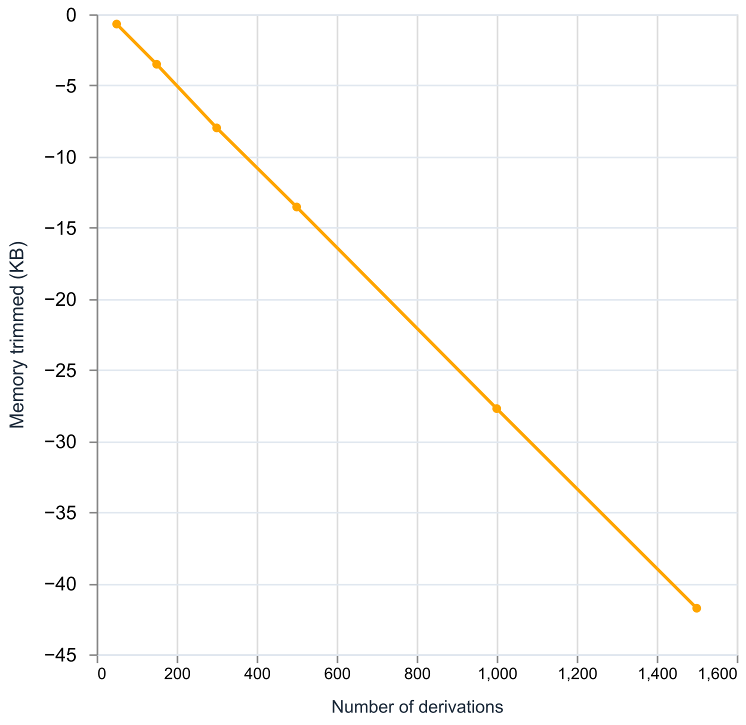

Figure 6.15: Time to trim (Partial).

7 Discussion & Future Work

This chapter will present the implications of our results and evaluations. The potential limitations and possibilities of our approach will be discussed. The difficulties encountered during the development of the system will be highlighted. It is evident that several aspects of the current approach can be improved upon and subject to further research.

7.1 Challenges during Development

During our iterative development process, we encountered some issues with the different approaches we took. This section will discuss their implications and potential solutions.

7.1.1 Invisible or Missing Deltas

We will present two cases where we had to rethink our initial the basic replication and second approach stateless repair approaches. For case one, consider the following example:

Suppose we have a system with three sites s_1, s_2, and s_3. At site s_1 we have relation H with the following facts:

$$\begin{aligned} h_1 &\leftarrow r_1, r_2 \\ h_1 &\leftarrow h_2 \\ h_1 &\leftarrow h_3 \end{aligned}$$

at site s_2 we have relation H_2 with the following configuration:

$$h_2 \leftarrow r_1, r_3$$

and at site s_3 we have relation H_3 with the following configuration:

$$h_3 \leftarrow r_1, r_3$$

we observe that the PRAD partition algorithm will generate the following configuration for the above relations:

$$\begin{aligned} H_1 &\leftarrow R_1^1, R_2^1 \\ H_2 &\leftarrow R_1^2, R_3^2 \\ H_3 &\leftarrow R_2^3, R_3^3 \end{aligned}$$

This implies that the Relations R_1^n and R_1^n may not be collocated at the same site, but at different sites. Another observation is that for two replicas of H_1, one may have received $h_2(r_1, r_3)$, but the other replica never received this update. Depending on the site configuration, this may lead to issues when repairing the sites later on. First, between replicas there are deltas from an upstream or peer replica that are invisible in the local Cl-Map. For instance, if s_1 received r_1, r_2 and $h_2(r_2, r_3)$, then goes offline. It has effectively received $h(r_1, r_2 + r_2, r_3)$, and the Cl-Map contains r_1, r_2 and r_3. When s_1 comes back online, how does it know that it is missing r_1, r_3, and has to restore to $h(r_1, r_2 + r_1, r_3)$? Second, between the upstream and downstream sites, if s_1 first sends $h(r_1, r_2, +r_2, r_3)$ and at a later point sends, but loses, $h(r_1, r_2 + r_1, r_3)$. The upstream site will be unable to detect whether at downstream site has applied the latter delta or not.

For the second case, we have an even simpler example, based on the first case.

Suppose we have at site s_1 the relation H with the following configuration:

$$
\begin{aligned}
h_1 &\leftarrow r_1, r_2 \\
h_2 &\leftarrow r_1, r_3 \\
h_3 &\leftarrow r_2, r_3
\end{aligned}
$$

If a downstream site s_2 has applied the updates h_1 and h_2, how does it know that it is missing h_3?

The two cases indicate that there are invisible delta updates which may come from a single upstream site. Essentially, the problem is that $\ddot{I_1} > \ddot{I_2}$, but $cl(\ddot{I_1}) = cl(\ddot{I_2})$. Therefore, Causal Lengths per site, are not sufficient to determine the state of a site.

As shown in Section 3.4.3 our proposed solution is to use a Cl-Map for each input relation, with bundle generated facts into a composite Cl-Map. This gives the property that for input relations $\mathbf{H} = H_n$ with EDB facts r_n, a site s has not merged the facts $\ddot{f} = \rangle f, p_f, cl_f, m_f \langle$, if and only if there exists some EDB r_i in p and $cl_f^{H_m} \in cl$, such that $cl_f^{H_m}(r_n) > cl_s^{H_m}(r_n)$. Consequently, if $cl_f^{H_m}(r_n) > cl_s^{H_m}(r_n)$, then we know that there is no need to restore the facts $\ddot{f}$.

7.1.2 Communication Overhead

During the design of the communication setup between the sites, we encountered an issue that depended on the communication strategies used. These strategies could potentially introduce an overhead. Suppose we have a system with the communication strategy N-N, where each site communicates with every other site. Two methods exist for transmitting the deltas between the sites. The first method is a single trip. We pull all the deltas from the upstream site, and calculate the difference between the local Cl-Map and the received Cl-Map. The second method is a round-trip. We push the local Cl-Map to the upstream site, which calculates the difference. This difference is then sent back to the local site. The two methods have different trade-offs in terms of the data transmitted and the messages sent. The round-trip method will result in a reduction in the amount of data being transmitted, as only the difference and the small local Cl-Map is sent. However, this comes at the cost of more messages being sent. Furthermore, in the roundtrip method, the upstream site has a smaller difference to calculate. The single trip method will result in potentially more data being transmitted, as the entire Cl-Map is sent. After careful consideration, we have opted for the roundtrip method, which offers greater efficiency in terms of data transmitted per message, while also requiring less processing at the upstream site.

7.1.3 Upstream or Peer for Repair

In the current implementation, both upstream and peer sites can be utilized for the repair process. Between an upstream site s and a peer site s', if $cl_s \leq cl_{s'}$, then there are no missing deltas at s' and peer s' can be used for repair. However, if $cl_s > cl_{s'}$, then s has more deltas than s', and the upstream s should be used for repair, as it possesses more recent information. The process of repairing using peer sites is the identical to that of using upstream sites, but it also has certain advantages. Firstly, there is no necessity for re-deriving the facts, as the peer site has already derived them. Secondly, there is no necessity to compare multiple upstream sites, instead, the peer site can be utilized directly. Finally, the peer sites are often situated closer in terms of network latency, which could lead to faster repair times. Nevertheless, for shorter offline periods, the peer sites have a larger comparison space, which could lead to longer repair times. In the current implementation we utilize both upstream and peer sites for repair, and decide on a case-by-case basis which to use.

7.1.4 Latency for Local Operations

In the current implementation, we have added a latency of a few milliseconds to the *GenServer* calls. This is due to wanting a more realistic simulation of the network. However, this is also added to local GenServer calls, which are performed on the same site. Which is not realistic, as the calls would be instantaneous. This is taken into account in the performance evaluation in Section 6.2, but it is important to note that this is not a realistic scenario. Instead of utilizing the manual :TIMER.SLEEP/1 function inside the business logic to add the delay, we could use the *Process.send_after/4* function inside the GenServer callbacks from to other sites. This would permit us to simulate the latency without affecting the local calls.

7.2 Future Work

There are several areas where the current implementation could be improved upon. The potential of the PRAD runtime has yet to be fully realized. Some edge cases have not been considered, and the system still exhibits limitations. The system is not yet ready for deployment in a production environment. This section will discuss some of the potential improvements that could be made to the system.

7.2.1 Storage Options

The current implementation stores the Cl-Maps and Commits as Elixir Maps. This approach offers a straightforward and effectiuve means of storing data, as Cl-Maps are relatively compact and can be utilized to leverage the integrated MAP functions. For instance, the MAP.MERGE/2 [1], which allows us to merge two maps into a single entity, with the resolution of conflicts being handled by the specified function. However, as the system grows, the Cl-Maps will inevitably become larger, which could potentially lead to performance issues. One way to mitigate this is to utilize the built-in in memory storage solution in Erlang, known as ETS.This is a key-value store, which has been optimized for fast reads and writes. This would permit us to store the Cl-Maps in memory, albeit with a more efficient data structure. However, this would also introduce additional complexity, as we would have to manage the ETS tables ourselves. Furthermore, ETS tables are also not subject to garbage collected, which means that we would also have to manage the memory ourselves. In this trade-off, we must consider the simplicity of using Elixir Maps, versus the performance of using ETS. As PRAD is still in the early stages of development, and it allowed us to quickly iterate on the design. But for a production system, it would be necessary to switch to a more performant storage solution.

Another analogous trade-off is in the implementation of the IDB and EDB, which are currently in memory using ETS. This will ultimately result in a limitation of the available memory. Currently, this is not a problem as the IDB and EDB are sufficiently small to fit in memory. However, as the system grows, the memory usage will also increase. It would be advantageous to transition to an on-disk storage solution, such as Erlang's Mnesia[2]. This would permit the storage of larger databases, but would also introduce additional processing overhead. As the data would have to be read from disk, which is slower than reading from memory. An important balancing act is to find the right balance between storage and performance. We choose to rely on the in-memory storage for the current implementation, as it is more performant, and we don't need to clear the Database manually, but it will be necessary to switch to a disk-based solution as the system grows. This could provide an interesting area for future research, as it would allow us to deeper explore the trade-offs between memory usage and performance.

1. https://hexdocs.pm/elixir/1.16.3/Map.html#merge/2
2. A distributed DBMS in Erlang

7.2.2 Asynchronous vs Synchronous messages

The current implementation of PRAD employs GenServer calls, synchronous messages, to facilitate communication between the sites. This implies that the calling process will block until the receiver has processed the message. This could have notable performance and scalability implications. And also goes against the PRAD principles of being an asynchronous coordination-free system. An alternative approach would be to utilize GenServer casts, asynchronous messages, could be used. This would allow the calling process to continue without waiting for the receiver to process the message. However, this would also mean that the calling process would not know if the message was successfully processed. Which could work with the coordination-free nature of the system. A third option would be to use a combination of both. However, we would lose the capability to use call as a backpressure mechanism. As the calling process would lack the ability to know if the receiver is overloaded.

In addition to the performance implications, there are also implications for the development process. Asynchronous messages are more challenging to debug, as the calling process does not know if the message was successfully processed. This could result in the occurrence of undetected bugs, as the calling process would continue without knowing if the message was processed. Conversely, synchronous messages are more straightforward to debug, as they provide direct feedback which is invaluable for development.

The following Figure 7.1 provides a summary of the interactions between upstream and downstream sites, as facilitated by GenServers. The processes and communication via messages are indicated by the continuous line. The calls to our module made by the GenServer are indicated by the dotted lines.

7.2.3 System Structure

In the current implementation, we have a lot of duplicated code. This is a consequence of the iterative development process, during which new replication and restoration features were added incrementally after passing the initial tests. The modules EDB, IDB and *join* are illustrative examples of this phenomenon. The aforementioned modules share similar functions, yet exhibit slight differences in their respective implementation.

This is an undesirable practice, as it leads to code duplication, which is more challenging to maintain and debug. It would be more prudent to adhere to the *DRY* principle, which stands for Don't Repeat Yourself. This implies that we should strive to write code that is reusable, and to refrain from the duplication of code when it is not necessary. This would make the code easier to maintain,

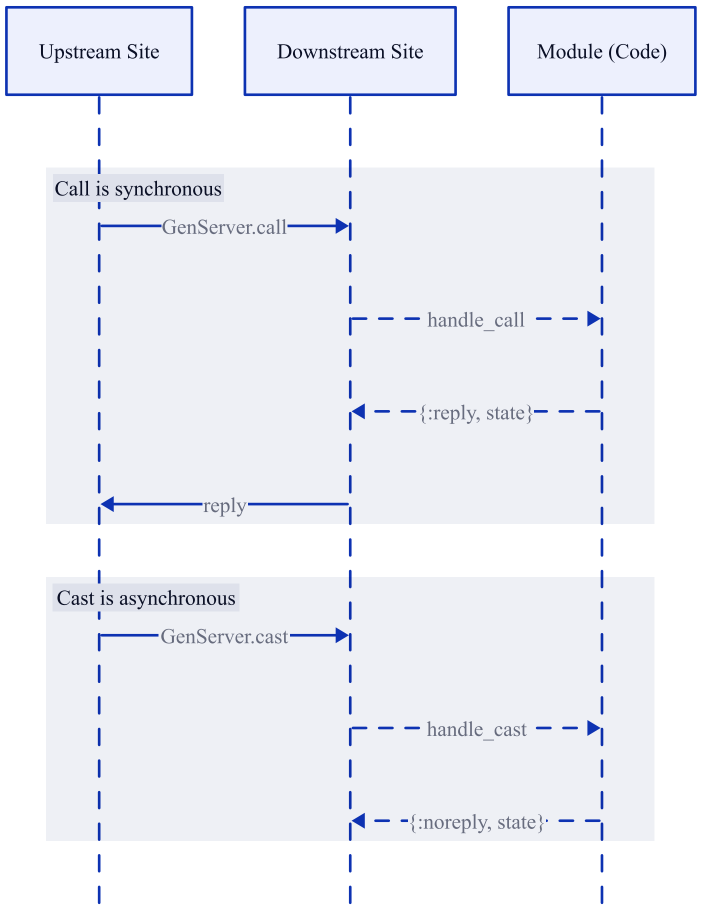

Figure 7.1: GenServer Communication Example.

as we lessen the burden on future developers. One way to achieve this is to use Elixir protocols [3] which is a built-in mechanism to achieve polymorphism in Elixir. In contrast to the limitations of traditional approaches, the use of protocols would allow us to define a set of functions that change behaviour based on the data type. This would enable the development of modular systems that can be readily adapted to accommodate an expanding range of data types and enable a future decoupling of the existing modules.

7.2.4 Composite Cl-Map

At present, a provisional solution has been devised for the problem of invisible deltas as discussed in Section 7.1.1. This solution involves the use of a Cl-Map for each input relation. This allows us to determine if a site is missing deltas from an upstream site. Nevertheless, we are still encountering certain edge cases, where the Cl-Map insufficient for determining whether a site is missing deltas. For instance, in the case of sites with multiple tables, it is not possible to ascertain whether the site is lacking deltas for a specific table. A composite Cl-Map could be employed here, comprising the Cl-Map for each update. The creation and propagation of composite Cl-Maps would permit the resolution of these edge cases, thereby providing a more comprehensive solution to the issue of invisible deltas.

7.2.5 BEAM GC interference

As evidenced by the results of the performance evaluation presented in Section 6.5, the garbage collection of the BEAM VM has an impact on partial trimming. Further experiments could be conducted to investigate the impact of BEAM garbage collection on system performance in greater depth. One potential avenue for further investigation is the use of the SPAWN_OPT [4] function to increase the initial stack and heap size of the processes. This would permit the execution of processes without invoking the BEAM garbage collection.

3. https://hexdocs.pm/elixir/protocols.html
4. https://www.erlang.org/doc/apps/erts/erlang.html#spawn_opt

8 Conclusion

To conclude the thesis we first present some related work relevant to the research presented in this thesis, and then summarize our contributions and provide some concluding remarks.

8.1 Related Work

We have a unique approach to the problem of fault tolerance using LWC with Cl-Map CRDT, which is not directly comparable to other systems. Therefore, we will present the closest related work that we could find.

8.1.1 Fault Tolerance Approaches

Fault tolerance is a crucial aspect of distributed systems, ensuring that systems remain available and consistent even in the presence of faults. A variety of approaches have been developed to achieve fault tolerance, each with its own trade-offs between availability and consistency. Availability is often measured by processing latency, while eventual consistency ensures that all replicas process the input in the same order and produce the output in the same order. One of the primary challenges in fault tolerance is managing the availability-consistency trade-off.

Various fault tolerance approaches exists that implement different trade-offs between availability and consistency, to achieve an improved fault-tolerance degree. Among others classifications, Martin [39] present categorizes fault tolerance approaches into the following categories:

- Active replication: Merges the outputs of upstreams to maintain total order.
- Upstream backup: Buffer output messages and replay them when needed.
- Passive replication: Combines checkpointing with upstream backup to restore the state.
- Active standby: Active replica does not send output.
- Passive standby: Receive state updates.

Furthermore, they present a replication method is based on a per-operator basis, whereby each operator replica can switch between active and backup modes.

In our approach, we have identified the same fault tolerance categories, but we have chosen to focus on the active and passive replication approaches. We have chosen to focus on these approaches because they are the most common fault tolerance mechanisms used in distributed systems, and they provide a good balance between availability and consistency. Our approach differs from their adaptive fault-tolerance per operator replication, in that we utilize our delta output trimming mechanism to reduce the amount of state that needs to be replicated, to achieve similar efficiency improvements to their dynamic approach.

8.1.2 Checkpoints and State Reconciliation

In order to restore the state of a failed replica, checkpointing is employed to save the state of the system at regular intervals. The creation of consistent snapshots of the system's state, checkpointing enables the recovery from faults by reverting to the last known good state. Techniques such as boundary messages (punctures and heartbeats) as presented by Balazinska [40], to help ensure that tuples from duplicate streams, with timestamps smaller than the smallest boundary timestamp, are totally ordered.

Our approach differs from theirs in that it utilizes Cl-Map CRDT to maintain the state of the system. This allows us to avoid the additional complexity of managing timestamps and boundary messages. Since our Cl-Map CRDT is commutative, associative and idempotent, we can merge the state of the system without the need for additional synchronization. Finally, as we rely on Erlang's 'let it crash' philosophy, we can avoid the need for heartbeats, we have instead opted for a rapid repair mechanism, which allows us to recover from failures faster.

In their work, Martin [39] present a system called StreamMine3G, which employs checkpoints to restore the state of the system. Checkpoints are created by locking the state, serializing it, and writing it to disk or sending it to another node. Non-replication optimizations such as sorted merge using epochs are employed to improve efficiency. They utilize a two-phase consensus protocol for replicated operators to achieve consistency.

Our work differs from that described above in that we write the commits to an in-memory map, rather than to on-disk storage. This allows us to avoid the overhead of disk I/O. Additionally, they employ the punctures to sort the merged tuples and a two-phase consensus protocol for consistency among replicas, which we circumvent by using Cl-Map CRDTs to maintain the state of the system. This approach allows us to guarantee strong eventual consistency without the need for additional synchronization.

Hwang [41] presents a system which eases the replication process by merging duplicate upstreams into a single stream. This enables nodes to function independently. The merged input streams comprise non-duplicate results, although they may be presented in a different order. The consistency of the system is guaranteed by the production of identical tuples, regardless of their order. Punctures are employed to constrain the size of metadata and to guarantee that merged replicas are sorted correctly when necessary.

Our work is analogous to theirs, in that we both utilize a mechanism to limit the size of metadata by trimming it at regular intervals. However, our approach has a policy-based decision-making to determine when an upstream should trim its delta output buffer. Additionally, their approach also considers the current resource usage of the sites, which we do not consider. Finally, our approach is not dependent on the order of the tuples, as we rely on Cl-Map CRDTs to maintain the state of the system.

8.2 Concluding Remarks

This thesis has presented an approach to repair PRAD programs at runtime using replication and restoration mechanisms. It combines traditional fault tolerance mechanisms with Lightweight Commit (LWC), and Conflict-free Replicated Data Type (CRDT) research. Our approach is designed to be lightweight and efficient, ensuring that the system can recover from failures at low cost and without the need for state reconciliation. Existing fault tolerance solutions often rely on vector clocks, sequence numbers or consensus protocols, which can be expensive and complex to implement. Additionally, it differs from existing fault tolerance solutions in that it employs a trimming mechanism to reduce the amount of state that needs to be repaired.

We have gained insights and built an understanding of PRAD. First, we undertook an exploration into the theoretical foundations of PRAD, encompassing the domains of CRDT, Consistency, Datalog, Provenance and Fault Tolerance. We chose to implement our PRAD extension in Elixir, due to it being a functional programming language that runs on the BEAM, because of its fault-tolerance, scalability and performance characteristics. During the development process, we employed software engineering practices to streamline the development process and ensure code quality. Test cases were created using TDD, to ensure that the code is well-tested and that the API is well-defined.

A series of experiments was conducted to evaluate the performance, storage, and communication capabilities of the system. Monitoring was implemented to provide further insight into the performance and behavior of the system. The results indicate that the system is capable to repair PRAD site failures through replication and restoration with minimal overhead. Furthermore, our Trimming algorithm is shown to be effective in reducing the amount of state that needs to be repaired, and the time it takes to repair the state.

There is still a considerable amount of future work that can be done to improve the implementation. One area of potential improvement is to optimize the storage and communication of the implementation. Allowing for asynchronous communication between replicas and utilizing ETS for storage, could potentially improve the performance of the system. Another area of improvement is to implement a composite Cl-Map CRDT, which would allow for the representation of more complex site states. Finally, the implementation could be moved to utilize the Elixir Protocols, which would allow for a more modular and extensible implementation.

In conclusion, this thesis has presented a novel approach to fault tolerance in PRAD programs, utilizing LWC with Cl-Map CRDT. The implementation is shown to be lightweight and efficient, and to be capable of repairing PRAD programs from failures with minimal overhead.

Bibliography

[1] Martin Kleppmann et al. "Local-first software: you own your data, in spite of the cloud." en. In: *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas , New Paradigms, and Reflections on Programming and Software*. Athens Greece: ACM, 2019, pp. 154–178. ISBN: 9781450369954. DOI: 10.1145/3359591.3359737. URL: https://dl.acm.org/doi/10.1145/3359591.3359737 (visited on 11/19/2023).

[2] Seth Gilbert and Nancy Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services." en. In: *ACM SIGACT News* 33.2 (June 2002), pp. 51–59. ISSN: 0163-5700. DOI: 10.1145/564585.564601. URL: https://dl.acm.org/doi/10.1145/564585.564601 (visited on 12/07/2023).

[3] Shadaj Laddad et al. "Keep calm and crdt on." In: (2022). DOI: 10.48550/ARXIV.2210.12605. URL: https://arxiv.org/abs/2210.12605 (visited on 11/19/2023).

[4] *How Discord Scaled Elixir to 5,000,000 Concurrent Users*. URL: https://discord.com/blog/how-discord-scaled-elixir-to-5-000-000-concurrent-users (visited on 12/07/2023).

[5] Rick Reed. "That's Billion with a B: Scaling to the next level at WhatsApp." In: *Erlang Factory* (2014). URL: https://www.erlang-factory.com/static/upload/media/1394350183453526efsf2014whatsappscaling.pdf (visited on 12/06/2023).

[6] Owais Qayyum and Weihai Yu. "Toward replicated and asynchronous data streams for edge-cloud applications." en. In: *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*. Virtual Event: ACM, Apr. 2022, pp. 339–346. ISBN: 9781450387132. DOI: 10.1145/3477314.3507687. URL: https://dl.acm.org/doi/10.1145/3477314.3507687 (visited on 11/19/2023).

[7] D. E. Comer et al. "Computing as a discipline." en. In: *Communications of the ACM* 32.1 (Jan. 1989). Ed. by Peter J. Denning, pp. 9–23. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/63238.63239. URL: https://dl.acm.org/doi/10.1145/63238.63239 (visited on 12/04/2023).

[8] George Grätzer and Friedrich Wehrung. *Lattice Theory: Special Topics and Applications. Volume 1*. Vol. 1. Sept. 2014. ISBN: 978-3-319-06412-3. DOI: 10.1007/978-3-319-06413-0.

[9] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. 2nd ed. Cambridge University Press, Apr. 2002. ISBN: 9780521784511 9780511809088. DOI: 10.1017/CBO9780511809088. URL: https://www.cambridge.org/core/product/identifier/9780511809088/type/book (visited on 11/28/2023).

[10] Marc Shapiro et al. "A comprehensive study of Convergent and Commutative Replicated Data Types." en. In: (Jan. 2011). URL: https://inria.hal.science/inria-00555588 (visited on 12/04/2023).

[11] Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. "Semiring-based constraint satisfaction and optimization." en. In: *Journal of the ACM* 44.2 (Mar. 1997), pp. 201–236. ISSN: 0004-5411, 1557-735X. DOI: 10.1145/256303.256306. URL: https://dl.acm.org/doi/10.1145/256303.256306 (visited on 11/29/2023).

[12] Leslie Lamport. "Paxos made simple." In: *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (2001), pp. 51–58.

[13] Butler Lampson and David Lomet. "A new presumed commit optimization for two phase commit." In: *19th International Conference on Very Large Data Bases (VLDB'93)*. 1993, pp. 630–640.

[14] Joseph M. Hellerstein and Peter Alvaro. "Keeping CALM: When Distributed Consistency is Easy." In: (2019). DOI: 10.48550/ARXIV.1901.01930. URL: https://arxiv.org/abs/1901.01930 (visited on 12/14/2023).

[15] Marc Shapiro et al. "Conflict-Free Replicated Data Types." In: *Stabilization, Safety, and Security of Distributed Systems*. Ed. by Xavier Défago, Franck Petit, and Vincent Villain. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 386–400. ISBN: 978-3-642-24550-3.

[16] Ken Birman, Gregory Chockler, and Robbert van Renesse. "Toward a Cloud Computing Research Agenda." In: *SIGACT News* 40.2 (2009), pp. 68–80. ISSN: 0163-5700. DOI: 10.1145/1556154.1556172. URL: https://doi.org/10.1145/1556154.1556172.

[17] Joseph M. Hellerstein. "The Declarative Imperative: Experiences and Conjectures in Distributed Logic." In: *SIGMOD Rec.* 39.1 (2010), pp. 5–19. ISSN: 0163-5808. DOI: 10.1145/1860702.1860704. URL: https://doi.org/10.1145/1860702.1860704.

[18] Nuno Preguiça. "Conflict-free Replicated Data Types: An Overview." In: (2018). DOI: 10.48550/ARXIV.1806.10254. URL: https://arxiv.org/abs/1806.10254 (visited on 11/19/2023).

[19] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. en. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. Chap. 3. ISBN: 9783642152597 9783642152603. DOI: 10.1007/978-3-642-15260-3. URL: http://link.springer.com/10.1007/978-3-642-15260-3 (visited on 11/20/2023).

[20] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. “Efficient State-based CRDTs by Delta-Mutation.” In: (2014). DOI: 10.48550/ARXIV.1410.2803. URL: https://arxiv.org/abs/1410.2803 (visited on 11/25/2023).

[21] Vitor Enes et al. “Efficient Synchronization of State-based CRDTs.” In: (2018). DOI: 10.48550/ARXIV.1803.02750. URL: https://arxiv.org/abs/1803.02750 (visited on 11/26/2023).

[22] Colin J Fidge. “Timestamps in message-passing systems that preserve the partial ordering.” In: (1987).

[23] Nuno Preguiça, Carlos Baquero, and Marc Shapiro. “Conflict-free Replicated Data Types (CRDTs).” In: (2018). DOI: 10.48550/ARXIV.1805.06358. URL: https://arxiv.org/abs/1805.06358 (visited on 11/19/2023).

[24] Weihai Yu and Sigbjørn Rostad. “A low-cost set CRDT based on causal lengths.” en. In: *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*. Heraklion Greece: ACM, Apr. 2020, pp. 1–6. ISBN: 9781450375245. DOI: 10.1145/3380787.3393678. URL: https://dl.acm.org/doi/10.1145/3380787.3393678 (visited on 11/26/2023).

[25] S. Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases*. Reading, Mass: Addison-Wesley, 1995. ISBN: 9780201537710.

[26] Tom J. Ameloot et al. “Weaker Forms of Monotonicity for Declarative Networking: A More Fine-Grained Answer to the CALM-Conjecture.” In: *ACM Trans. Database Syst.* 40.4 (2015). ISSN: 0362-5915. DOI: 10.1145/2809784. URL: https://doi.org/10.1145/2809784.

[27] I. Balbin and K. Ramamohanarao. “A generalization of the differential approach to recursive query evaluation.” en. In: *The Journal of Logic Programming* 4.3 (Sept. 1987), pp. 259–262. ISSN: 07431066. DOI: 10.1016/0743-1066(87)90004-5. URL: https://linkinghub.elsevier.com/retrieve/pii/0743106687900045 (visited on 11/29/2023).

[28] Maarten Van Steen and Andrew S Tanenbaum. *Distributed systems*. Maarten van Steen Leiden, The Netherlands, 2017.

[29] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. “Provenance semirings.” In: *Proceedings of the Twenty-Sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. PODS ’07. New York, NY, USA: Association for Computing Machinery, 2007, pp. 31–40. ISBN: 9781595936851. DOI: 10.1145/1265530.1265535. URL: https://doi.org/10.1145/1265530.1265535.

[30] Bas Ketsman, Aws Albarghouthi, and Paraschos Koutris. “Distribution Policies for Datalog.” en. In: *Theory of Computing Systems* 64.5 (July 2020), pp. 965–998. ISSN: 1432-4350, 1433-0490. DOI: 10.1007/s00224-019-09959-3. URL: http://link.springer.com/10.1007/s00224-019-09959-3 (visited on 05/12/2024).

[31] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: simplified data processing on large clusters.” In: *Commun. ACM* 51.1 (2008), pp. 107–113.

ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. URL: https://doi.org/10.1145/1327452.1327492.

[32] *Manifesto for Agile Software Development*. URL: https://agilemanifesto.org/ (visited on 12/04/2023).

[33] mijacobs. *How Microsoft plans with DevOps - Azure DevOps*. en-us. Nov. 2022. URL: https://learn.microsoft.com/en-us/devops/plan/how-microsoft-plans-devops (visited on 12/13/2023).

[34] Abdullahi Olaoye. *Beginning DevOps on AWS for iOS development: Xcode, Jenkins, and Fastlane integration on the cloud*. eng. New York: Apress, 2022. ISBN: 9781484280232 9781484280225.

[35] José Valim. *Elixir*. URL: https://elixir-lang.org/ (visited on 12/03/2023).

[36] Joe Armstrong. *Making reliable distributed systems in the presence of software errors*. 2003. URL: https://erlang.org/download/armstrong_thesis_2003.pdf (visited on 12/03/2023).

[37] *RabbitMQ Erlang Version Requirements — RabbitMQ*. URL: https://www.rabbitmq.com/which-erlang.html (visited on 12/14/2023).

[38] Erik Stenman. *The Erlang Runtime System*. URL: https://blog.stenmans.org/theBeamBook (visited on 12/03/2023).

[39] André Martin. “Minimizing Overhead for Fault Tolerance in Event Stream Processing Systems.” In: (2016), pp. 1–190.

[40] Magdalena Balazinska et al. “Fault-tolerance in the borealis distributed stream processing system.” In: *ACM Trans. Database Syst.* 33.1 (2008). ISSN: 0362-5915. DOI: 10.1145/1331904.1331907. URL: https://doi.org/10.1145/1331904.1331907.

[41] Jeong-Hyon Hwang. “Fast and highly-available stream processing.” In: (2009). AAI3377136.