# Algebraic Approaches to Distributed Data Systems

*Conor Power*

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2025-103
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2025/EECS-2025-103.html

May 16, 2025

# Algebraic Approaches to Distributed Data Systems

by

Conor Power

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Joseph M. Hellerstein, Chair
Associate Professor Joseph Gonzalez
Associate Professor Paraschos Koutris
Associate Professor Matei Zaharia

Spring 2025

Algebraic Approaches to Distributed Data Systems

Abstract

Algebraic Approaches to Distributed Data Systems

by

Conor Power

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Joseph M. Hellerstein, Chair

With the rise of cloud computing, software systems have become increasingly distributed. Distributed systems offer myriad benefits such as scalability, availability, and fault tolerance. However, they introduce complexity for the programmers of these systems to ensure correctness and hide non-determinism from the end-user. To address this challenge of programming cloud-scale systems, the Hydro project at Berkeley explores bringing declarative programming to the distributed systems space. Declarative programming has had enormous success in the field of databases in the form of SQL. Its benefit is that it allows developers to specify their goals at a high level and leave complex implementation decisions up to the database system.

In this thesis, we explore the marriage of these two worlds: distributed systems programming and declarative database systems. In pursuit of this marriage, we study independent trends towards algebraic models in distributed systems and database systems. This thesis extends these works, explores the relationship between them, and demonstrates the practical applicability of algebraic properties to optimizing distributed data systems.

In particular, we study four lines of research on algebraic properties for distributed data systems: conflict-free replicated data types (CRDTs), algebraic models of incremental view maintenance (IVM), parallel database aggregates, and the CALM Theorem. While these topics have been studied under different formalisms across different research communities, we are able to build bridges between them. We are able to bring the system model and mathematical model of CRDTs, studied in the distributed systems and programming languages communities, to these three other topics that have been studied entirely within the databases research community. The result is a foundation on which to support the benefits of declarativity in distributed systems programming.

To my wife, Laura.

# Contents

# List of Figures

# List of Tables

## Acknowledgments

I have a lot of people to thank. First, I would like to thank the numerous mentors that have encouraged me throughout my career prior to the PhD: Jennifer Welborn, David Mix Barrington, Neil Immerman, Kedar Dubhashi, Carlo Curino, and Rich Draves. I would not be where I am today without all of your support. Next, I would like to thank the people that took the time to mentor me during the last four years: Mae Milano, Natacha Crooks, Paris Koutris, Val Tannen, Dan Suciu, and Hein Meling. I really appreciate all of the time you all have spent with me and I've learned so much from each of you. The last mentor I would like to thank is my advisor, Joe Hellerstein. Thank you for teaching me how to be a scientist. You've taught me how to do good research and how to communicate that research to a wide range of audiences. You've also taught me how to build a research culture that thinks outside the box and that always prioritizes the people behind the science.

The thing that has made the last years so special, without question, is the students I've been surrounded by at Berkeley. I can't imagine what my PhD would have been like had I not sat in the Berkeley Sky Lab. The constant flow of ideas and energy that comes with being surrounded by so many brilliant people has been truly amazing. First, I want to thank the Hydro team for endless hours both thinking deeply about hard problems and living PhD life to the fullest. David Chu, Shadaj Laddad, Chris Douglas, Mae Milano, Tiemo Bang, Mingwei Samuel, Justin Jaffray, and Lucky Katahanas - you have all had enormous influence on the way that I think about computers. I also want to thank the other people that I got the chance to collaborate with on research: Lakshya Agrawal, Alvin Cheung, Tathagata Das, Tyler Griggs, Tyler Hou, Paras Jain, Darya Kaviani, Sam Kumar, Shu Liu, Raluca Popa, Laura Power, Peter Kraft, Samyu Yagati, Ion Stoica, Tenzin Ukyab, Justin Wong, and Matei Zaharia. I would also like to thank the students that I had the opportunity to mentor during the PhD: Sai Achalla, Ryan Cottone, Chae Lee, and Nathaniel Macasaet.

There are a lot of other people that have had enormous influence on the experience of my PhD. Although we never got the chance to write papers together, I am forever appreciative for the energy they've brought to the Sky Lab and to the Berkeley EECS department: Ale Escontrela, Alok Tripathy, Altan Haan, Ameesh Shah, Anna Yoon, Audrey Cheng, Carolyn Dunlap, Carson Young, Charles Packer, Daniel Rothchild, Devesh Rathee, Emily Marx, Emma Dauterman, Gabriel Matute, Grace Dinh, Isabelle Lee, Jae Hong, Jean-Luc Watson, Jiwon Park, Julien Piet, Justin Kalloor, Kate McElroy, Kevin Lin, Laurenz Heppding, Lily Liu, Lisa Dunlap, Louis Lafair, Manish Shetty, Mayank Rathee, Medhini Narasimhan, Micah Murray, Michael Luo, Mick Kittivorawong, Naman Jain, Neil Giridharan, Nithin Chalapathi, Norman Mu, Parker Ziegler, Paul Crews, Peter Schafhalter, Preetum Nakkiran, Reggie Frank, Ritwik Gupta, Rolando Garcia, Ryan Almeddine, Sarah Wooders, Sarah McClure, Shishir Patil, Shm Almeda, Shreya Shenker, Slivery Fu, Soujanya Ponnapalli, Stephanie Wang, Sukrit Kalra, Suyash Gupta, Tess Despres, Tianjun Zhang, Vivian Fang, Wanda Mora, Wen Zhang, Zhuohan Li, and Ziming Mao. These folks, along with dozens of other people I've shared lunches with, hung out at DSF with, traveled to conferences with, and just generally orbited each other, made the last four years what they were - very very special. Thank you so much to everyone.

Finally, I'd like to thank my family. Mom, Dad, Ciara, Gramps, and Pammy - I love you guys. Maura - you didn't really help all that much with the PhD, but you're my favorite anyways. Laura - I'm glad we both like math and fun; thank you for being my partner in all things and supporting me through all my dreams and schemes.

# Chapter 1

# Introduction

### 1.0.1 Motivation

Computer systems have become increasingly distributed over the past twenty years. There are a variety of trends that have driven this paradigm shift. The first two drivers of this are infrastructural. The end of Moore's law means that scaling up a system cannot be reasonably achieved by vertical scaling (larger individual machines). Instead, scaling must be horizontal (adding more machines to the system). In parallel to this shift away from vertical scaling, Cloud Computing has arisen as the defacto way that developers manage computing infrastructure. This has made it operationally trivial to add more machines to the system and achieve this horizontal scaling.

Concurrent with these infrastructural trends towards horizontal scaling are application trends that drive the need for this scaling. The rise of the internet drove the need for massive "web scale" systems and also came with new user expectations of these systems. The internet is shared across the globe and online twenty four seven. This has driven extreme requirements on the availability of software systems. It has also driven the need for systems that are spread across the globe and able to serve low-latency reads and writes on every continent simultaneously.

Despite this rapid growth in globally distributed cloud systems, the way that developers program distributed systems has made little progress to keep up with this trend.

### 1.0.2 State of the Art

The topic of distributed systems has been studied across different communities, each bringing a unique perspective, mental model, and formalism. In the database community, e.g SIGMOD and VLDB, the world is often divided into read-heavy workloads (OLAP) and write-heavy workloads (OLTP). OLTP is studied in terms of ANSI Consistency levels, based on the distinguishing feature that state mutations are in the forms of atomic transactions that touch multiple pieces of data.

The OLAP view of databases and distributed systems has extensively studied the conditions for distributed optimization of SQL queries, both through the lens of logic [8] and through the lens of algebra [35, 79]. The database theory community, e.g. PODS and ICDT, introduced a new model for studying distributed systems called relational transducer networks [2, 10]. This

line of work proved the CALM Theorem, stating an equivalence between queries expressible in monotonic logic and queries being computable without coordination.

The systems community, e.g. SOSP, OSDI, and EuroSys, often study distributed systems in terms of the consistency hierarchy. The guarantees of the system are broken up into categories like linearizable, sequential, causal, or read your writes. For weaker consistency levels (e.g. causal consistency), this community introduced the study of coordination-avoidance through convergent state mutation. Early work in this space such as Bayou [106] paved the way for the most common form of weak-consistency research today which is conflict-free replicated data types (CRDTs).

CRDTs offer an algebraic model of weak consistency based on guaranteeing convergence of replicated state without any central authority for ordering of operations and without risk of interference from network nondeterminism. The earliest use of this algebraic model of weak consistency was by Carlos Baquero and Francisco Moura in 1999 [15]. CRDTs have caught traction in the programming languages community and are an active area of research there. Common topics of interest are the verification of CRDT correctness [53, 70, 113] and designing languages that can take advantage of CRDTs [12, 68, 81, 83]. CRDTs are also notable for their significant adoption in industry. Apple Notes, League of Legends Chat, Sound Cloud, and the Delta Airlines and Lufthansa Airlines apps are all built on top of CRDTs.

While each of these research communities have made great progress in the study of distributed data systems, their approaches remain largely siloed from one-another due to the severe differences in perspective and in mathematical formalism. In this thesis, we take steps towards unifying these disconnected branches of scientific study.

## 1.1 Thesis Overview

This thesis is presented in four parts. In part one, we explore the relationship between CRDTs and the CALM theorem. To do this, we introduce a simple mathematical formalism based on semi-automata. This formalism allows us to reason about future states of the system based entirely on the question of graph reachability. We then are able to model the guarantees offered by the CALM Theorem in terms of graph reachability, and can extend this study to arbitrary functions and data types. This extends the previous state of the art on the CALM Theorem whose study was restricted to programs expressed as logical formulas being executed over *set* data types (the relational data model). This allows us reason about the guarantees on queries for any kind of CRDT or even non-CRDT data structures. In doing this, what we found was that the guarantee of the CALM Theorem [10] is not a natural user-facing guarantee outside of the realm of set data types. We remedy this by defining a more general property called Free Termination that the CALM Theorem is a special case of. The Free Termination property gives a simple and desirable guarantee: if you query a replica state and the replica has not converged, will the query result be the same when it does converge? If so, then this state is free terminating.

In part two, we look at the algebraic properties of CRDTs that make eventual consistency possible in the face of asynchronous networks. We question the notion that developers need to

design CRDTs that satisfy all three properties and observe that properties can often be enforced programmatically via metadata. We call this process of enforcing properties programmatically *algebraic upgrades* and explore in depth the limits of this technique for the *idempotence* property of CRDTs. Our exploration brings us to the development of a novel distributed protocol for idempotence upgrades called OnceTree that achieves constant-space metadata overhead. This improves on the prior state of the art tricks which require linear space in the number of replicas to enforce idempotence.

In part three, we widen our view beyond the algebraic model of coordination-freeness (CRDTs) to algebraic data models that are popular in the databases community. We look specifically at two algebraic models of incremental view maintenance, which can be thought of as the database perspective on incremental computation. Given the importance of both coordination-avoidance and incremental computation in the design of real-time distributed applications, we explore how these different algebraic models interact with one-another. Despite appearing to be incompatible, we show that by utilizing different algebraic structures at different layers of the system, we can achieve both an algebraic model of incremental computation and an algebraic model of coordination-free consistency. The tricks that make this cohabitation possible come from the *algebraic upgrades* view of the world introduced in part two. We will see that while the algebraic requirements are mathematically incompatible, we can translate between the two structures using metadata wrappers i.e. algebraic upgrades.

In part four, we look at the practical considerations of how to make it easy for developers to get the benefits of all the work on algebraic properties for program optimization. There is a tension here between wanting the developer to program in a way that is comfortable to them and wanting the system to have all of the information it needs about algebraic properties to perform optimizations. We see this same tension in SQL systems, where programs expressed using built-in SQL operators like SELECT, PROJECT, and JOIN are highly amenable to optimization, but it is often awkward or even impossible for developers to express the programs they want using relational algebra. To enable developer ergonomics, SQL supports user-defined functions in a wide range of turing-complete programming languages. This makes things easy for the developer, but completely intractable for the optimizer to reason about what properties the code satisfies. In this section, we demonstrate the applicability of formal verification tools for detecting the algebraic properties that are satisfied by arbitrary user code in the Rust programming language.

# Chapter 2

# Background

In this chapter, we provide relevant background on algebraic properties, CRDTs, and CALM that will show up throughout this thesis. Feel free to skip sections you are already familiar with or return to them later for reference.

## 2.1 Algebra Background

Abstract algebra is the study of properties of mathematical functions. Given that computer programs are mathematical functions, algebra applied to computer science can be thought of as the study of what properties code satisfies. Throughout the history of computer science, properties of code have been used to guarantee correctness of programs as well as equivalence of programs. Combined, these allow for program optimization. We can consider the set of equivalent programs made possible by the properties and then identify which of those programs will perform best in our setting.

In this section, we will define the various terms from abstract algebra used in this thesis and throughout the research literature. We will also provide examples and intuition for how each algebraic property is useful in the study of distributed data systems.

### 2.1.1 Commutativity

**Definitions and Examples**: Formally, a binary operator $\circ$ is commutative if for all inputs $x, y$, it holds that $x \circ y = y \circ x$.

In computer science, with a function $f$ that takes in multiple inputs of the same type, we can swap any two inputs $x$ and $y$ and $f$ will return the same result, i.e., $f(x, y) = f(y, x)$.

A pair of functions $f, g$ can also be commutative, meaning the order in which the functions are applied doesn't change the output. That is, $f(g(x)) = g(f(x))$. For example, in relational algebra for database queries, the ordering of selection operators and projection operators doesn't change the query result, so an optimizer can choose whichever ordering will give better performance.

Common examples of functions with commutative inputs are addition, multiplication, and set union. Some examples of non-commutative input functions are matrix multiplication and string concatenation.

In English, commutativity is the indifference to ordering. This indifference is valuable in settings like distributed systems where ordering of events is expensive to achieve, or compilation and query optimization where it means the compiler or optimizer has more degrees of freedom in finding an optimal program execution plan.

**Distributed Systems**: Two of the fundamental challenges of distributed systems are that there are no globally synchronized clocks and that messages over the network may arrive out of order. The lack of globally synchronized clocks is the challenge that consensus algorithms such as Paxos and Raft aim to solve. The ordering in which events in the system occur is determined by a leader node. This introduces a single-node bottleneck in the processing of requests in the system. In many applications, different orderings of events will result in different outputs. For example, one user updates a database row to the value 5 and another updates it to the value 8. Should the row's value be 5 or 8? It depends on which update is considered to have happened last. In some applications, the ordering does not influence the outcome which enables the applications to avoid the bottleneck of a consensus protocol. For example, if the two updates are to different rows in the database then their ordering is unimportant. For another example, consider an update operation that does not overwrite the row with the new value, but rather to add the values 5 and 8 respectively to the current total of the row. In this case, the ordering of the updates does not matter because addition is a *commutative* operation.

A variety of techniques in distributed systems make use of this commutativity. One of the most popular is conflict-free replicated data types (CRDTs), which are an object-oriented way of specifying a distributed object that can be modified without the need for consensus algorithms. There are two ways to model CRDTs, state-based on operation-based. The state-based version is based on semi-lattices, whereas the operation-based approach is simply about commutativity [102]. If the application of the update method you define on your CRDT object is commutative, then replicas are guaranteed to achieve *strong eventual consistency* via disorganized gossip communication. This consistency guarantee roughly means that all replicas will eventually converge to the same state regardless of the order in which messages are delivered over the network.

**Query Optimization**: Query optimizers leverage both forms of commutativity to achieve more efficient query plans. The commutativity of function application is used to reorganize query plans to put, for example, column projections before row filters. The commutativity of inputs to a function are used to avoid the use of *ordering enforcers* [44]. Note that while the relational algebra is defined over sets, an unordered data type, database query planners use physical operators that are sometimes order sensitive. Merge Join is an example of a physical operator that is non-commutative with respect to input order.

### 2.1.2 Associativity

**Definitions and Examples**: Formally, a binary operator $\circ$ is associative if for all inputs $x, y, z$, it holds that $(x \circ y) \circ z = x \circ (y \circ z)$. In English, associativity is the indifference to grouping of terms.

That is, any parenthesization of the expression will give the same result. We will see that both partitioning and batching in computer science can be seen as changes to the parenthesization of the function we are computing.

Common examples of associative functions include addition, multiplication, matrix multiplication, and string concatenation. Examples of non-associative functions include floating point addition, averaging, and exponentiation.

**Query Optimization**: Associativity is a property needed to enable parallel computation of partitioned input data sets. It allows for SIMD parallelism and is used heavily in OLAP Databases, Map Reduce, MPI, and multi-core computation. Intuitively, we can think of the partitioning of inputs as a parenthesization of the function application.

**Distributed Systems**: Associativity allows arbitrary batching together of operations which is an important optimization in distributed systems and streaming systems. Batching can be thought of as doing partial work on partial inputs and is often used to amortize fixed costs of operations, to allow work to be done in parallel without waiting for the full input, or to reduce the size of data before sending it over a networked channel. For this reason, it is one of the three properties required of a state-based CRDT.

### 2.1.3 Idempotence

**Definitions and Examples**: Formally, a binary operator $\circ$ is idempotent if for all $x$, it holds that $x \circ x = x$. In other words, performing the operation any arbitrary number of times with the same input value has the same effect as performing it only once. For a unary operator, idempotence is defined as for all $x$, $f(f(x)) = f(x)$. In English, idempotence is indifference to repeated function calls. No matter how many times a function is called, the result will match that of when the function is only called once. This allows for indifference to the duplication of messages.

Common examples of idempotent functions include set union, min, and max. Common non-idempotent functions include addition, multiplication, and string concatenation.

**Distributed systems**: Idempotence is commonly used in distributed systems where messages can be dropped or duplicated by an unreliable network. In the presence of an unreliable network, applications may need to "retry" sending messages and idempotence ensures that if the original message and the retried message both arrive then the outcome is the same as if it arrived once. Idempotence is one of the three requirements of merge operations in CRDTs to ensure states convergence regardless of network non-determinism. In that setting, idempotence is also needed to handle duplicate delivery of messages along multiple paths in the network via gossip protocols. We explore the role of idempotence in CRDTs in depth later in this thesis.

**Recursive computation and fixpoints**: Idempotence is also used to ensure the convergence of recursive computation. For example, the idempotence of set union is used to guarantee that recursive Datalog programs always converge in finite time.

Idempotence is a special case of a more general property called k-stability, defined as $f^{k+1} = f^k$, which can also be used to guarantee convergence of Datalog-style programs [3].

### 2.1.4 Invertibility

**Definitions and examples:** Formally, inverses come in two major forms. The first is for a binary operation, where we say the operator satisfies the *inverse element* property if for all $x$ there exists a $y$ such that $x \circ y = 0$ where 0 stands for the identity element. In other words, you can go from any value $x$ back to 0 by applying $\circ$ with the inverse value of $x$. In this sense, the inverse of 5 with respect to addition is $-5$. The inverse of 3 with respect to multiplication is $1/3$.

The second form is inverse functions. Formally, for a function $f$, $f^{-1}$ is the inverse of $f$ if for all $x$, $f^{-1}(f(x)) = x$ and for all $y$, $f(f^{-1}(y)) = y$. In this sense, subtraction is the inverse function of addition and division is the inverse function of multiplication.

Common examples of structures with inverses include addition over the integers or multplication over the rational numbers. However, addition or multiplication over the natural numbers do not have inverses.

In computer science, invertibility is a property that allows operations to undo themselves. This is important for both user experience and program optimization. For user experience, having a mathematically exact inverse means that a user pressing an "undo" button will have intuitive semantics. We will see throughout this thesis that "undo" semantics are actually a major challenge in coordination-free distributed systems because of the incompatibility of inverses with other relevant algebraic properties.

For program optimization, we can think of having inverses as giving us the ability to perform speculative work on partial inputs and then undo any part of the speculation that turned out to be wrong. This use of inverses is an essential component of designs for incremental computation such as DBSP [24].

### 2.1.5 Monotonicity and Inflationary Behavior

**Definitions and Examples**: In the context of functions over sets like databases (database tables are just sets of tuples), monotonicity is defined as $A \subseteq B \rightarrow F(A) \subseteq F(B)$ . For arbitrary data types, monotonicity is defined as $x \leq y \rightarrow f(x) \leq f(y)$. If the domain and range of the function $f$ are the same then a similar property called $inflationary$ is often used which states that $\forall x : f(x) \geq x$ for a unary function $f$ [20]. Common examples of monotone functions include set union or squaring over natural numbers. Common examples of non-monotone functions include xor and set difference. Common examples of inflationary functions include set union, addition over the natural numbers, and the immediate consequence operator of a Datalog program. Common examples of non-inflationary functions include modular addition, addition over the integers, or sine.

Observe that a function can be monotone but not inflationary, such as $f(x) = x/2$ over the positive real numbers. Further, a function can be inflationary but not monotone such as *absolute value* over the integers.

In English, *Monotonicity* is the property of a function that as the input grows, the output also grows. The related *inflationary* property says that a state will always grow upon application of a function. These guarantees about a function allow for strong system guarantees such as con-

vergence of replicated state, convergence of recursive computations, and guarantees on partial outputs of queries that they won't be retracted in the future. These properties show up frequently in the study of CRDTs, the CALM Theorem, query optimization, and Datalog.

**Query Optimization**: Of the five relational algebra operators *select, project, join, union*, and *set difference*, the only non-monotone operation is *set difference*. This enables streaming computation ie *pipeline parallelism* inside of database query executors. The guarantee that an operator is monotone gives us that the tuples output by running the operator on a portion of the input will be a subset of the tuples output by running the operator on the full input. For example, a join operator receiving tuples for the two input tables one at a time can store the state of the tuples it has seen so far and compare any new incoming tuples to check if they match the join equality condition. If they match, then the join can output the resulting tuple to downstream operators. The monotonicity of join guarantees that any such tuples output by the join will be included in the final result of the join, so there is no chance of needed to retract a tuple that was output earlier in the computation.

**Distributed Systems**: Monotonicity plays a large role in techniques for avoiding coordination in distributed systems. The CALM Theorem, which stands for Consistency as Logical Monotonicity, applies the same intuition about monotonic operators being safe to compute in a streaming manner to the distributed setting. In that setting, machines don't have enough information to figure out if they have seen all of the messages that have been sent by other machines. This setting is common in decentralized distributed computing and the CALM Theorem shows that the only queries that can be safely computed in this setting without coordination and without retracting outputs are monotone queries.

CRDTs leverage the inflationary property to guarantee convergence of replicated state without coordination. The guarantee they offer is strong eventual consistency under the requirement that state modifications be inflationary and that merging together states of replicas is done using an operation that is associative, commutative, and idempotent. Mathematically, a function that is associative, commutative, and idempotent induces a partial ordering and is inflationary with respect to that ordering. Thus, replica state can be seen as always increasing from both the update operation and the merge operation. The basic intuition for why this helps with convergence in CRDTs is that the larger states in the partial ordering will always be the newer states temporally, so the newer states will replace the older states.

### 2.1.6 Algebraic Structure Definitions

Terms for different combinations of properties are provided here for reference.

$+$ has an *additive identity* if $\exists 0 \in S : \forall a \in S : a + 0 = a$. We call this element "0" the additive identity.

$+$ has an *additive inverse* if $\forall a \in S : \exists b \in S : a + b = 0$. We say b is the additive inverse of a in this case, and will often denote this element b as $a^{-1}$.

$\times$ has a *multiplicative identity* if $\exists 1 \in S : \forall a \in S : a \times 1 = 1 \times a = a$. We call this element "1" the multiplicative identity.

$\times$ is *left-distributive* over $+$ if $\forall a, b, c \in S : a \times (b + c) = (a \times b) + (a \times c)$

$\times$ is *right-distributive* over + if $\forall a, b, c \in S$: $(b + c) \times a = (b \times a) + (c \times a)$

*invertibility*: $\forall g \in G \exists -g \in G : g + (-g) = 0$

A *semi-lattice* is an set $L$ equipped with a binary operation $\sqcup$ that satisfies three properties: associativity, commutativity, and idempotence.

Any valid semi-lattice operator induces the following partial ordering on the set $L$: $a \sqcup b = b \longrightarrow a \leq b$.

A *monoid* is a set $M$ equipped with a binary operation + that is associative. A monoid is also required to have an identity element, 0, satisfying $\forall m \in M : 0 + m = m + 0 = m$.

A *group* is a set $G$ equipped with a binary operation + that satisfies two properties: associativity and invertibility. A group is also required to have an identity element, 0, satisfying $\forall g \in G : 0 + g = g + 0 = g$.

A *abelian group* is simply a group where + is also commutative.

A *ring* is a set S equipped with two binary operations + and $\times$ satisfying the following algebraic properties: + is associative, commutative, has an additive identity, and has additive inverses, $\times$ is associative and has a multiplicative identity, and $\times$ is distributive over + (both left-distributive and right-distributive). On their own, the + operation forms an *abelian group* over S and the $\times$ operation forms a *monoid* over S.

a *semi-ring* is just a ring where the + operation does not need to be invertible.

a map $\varphi$ from one ring $R$ to another ring $S$ is a *ring homomorphism* if it satisfies: (1) $\varphi(a+b) = \varphi(a) + \varphi(b)$, (2) $\varphi(ab) = \varphi(a)\varphi(b)$, (3) $\varphi(1) = 1$, and (4) $\varphi(0) = 0$.

a ring homomorphism $\varphi$ is a *ring isomorphism* if and only if $\varphi$ is bijective. Two rings are *isomorphic* if and only if there exists a ring isomorphism between them.

We summarize which properties are required for each structure in Table 2.1.

| Structure | Associative | Commutative | Idempotent | Identity | Inverse | Distributive |
|---|---|---|---|---|---|---|
| Semi-lattice ($\sqcup$) | ✓ | ✓ | ✓ | | | |
| Monoid (+) | ✓ | | | ✓ | | |
| Group (+) | ✓ | | | ✓ | ✓ | |
| Abelian Group (+) | ✓ | ✓ | | ✓ | ✓ | |
| Semi-ring (+, $\times$) | ✓,✓ | ✓,– | | ✓,✓ | –,– | ✓ |
| Ring (+, $\times$) | ✓,✓ | ✓,– | | ✓,✓ | ✓,– | ✓ |

Table 2.1: Comparison of algebraic structures

## 2.2 CRDT Background

### 2.2.1 Semi-Lattices and Conflict-Free Replicated Data Types

Conflict-free replicated data types (CRDTs) are a popular interface for designing systems with strong eventual consistency of writes. In simple terms, this consistency guarantee means that

all replicas in a system will converge to the same state as long as all updates eventually propagate to them. The interface that CRDTs provide is an object-oriented one with roots in abstract algebra. A *state based CRDT* requires that a user defines two operations on their state: an *update* operation for changes to the state from outside the system (e.g. from users), and a *merge* operation for replicas to combine the updates they have seen into one state. By satisfying certain algebraic properties in the selection of update and merge operations, a developer is guaranteed strong eventual consistency of the replica state via gossip communication over an asynchronous computer network [102, 103].

The requirements on the CRDT operations are that merge is *associative*, *commutative*, and *idempotent* and that update is *inflationary* with respect to the ordering induced by the merge operation. An algebraic structure that is associative, commutative, and idempotent is called a semi-lattice, so the state-based CRDT model offers us an algebraic lens on eventual consistency in distributed systems. This model is sometimes referred to as the "semilattice data model" and was dubbed "ACID 2.0" by Helland, Campbell, and Finkelstein [48].

The reason for these three requirements on the merge operation of CRDTs is that they each provide protection against different sources of nondeterminism on the communication network. Associativity gives the system robustness to arbitrary batching, commutativity gives robustness to different interleaving or reordering of messages on the network, and idempotence gives robustness to messages being delivered multiple times or being "retried". The power of this robustness is that without coordination replicas can process updates and guarantee eventual convergence. This allows geo-replicated systems to service requests with local latency without sacrificing availability or consistency under network partitions (circumventing the CAP Theorem [21]).

A common example of a CRDT is a set. Merge is performed via set union which is associative, commutative, and idempotent. The update operation is also set union, so an update can add an element to the set and all replicas will eventually converge to exactly the set of individual updates applied across the replicas.

Below, we present some common examples of state-based CRDTs that we will refer back to throughout this thesis.

***Grow-Only Set CRDT.*** This CRDT is defined as follows:

**state:** Set $S$ over the universe $\mathbb{U}$

**update:** Set $S$ over $\mathbb{U}$ and $m \in \mathbb{U}$. Then, $\text{update}(S, m) = S \cup \{m\}$

**merge:** $\text{merge}(A, B) = A \cup B$

**query:** $Q(S) = S$ (identity query)

This merge operation, set union, forms a semi-lattice.

***Two-Phase Set CRDT.*** We define it as follows:

**state:** Two sets over the universe $\mathbb{U}$, INSERTS and DELETES.

**update:** The operation "delete($m$)" gives DELETES = DELETES$\cup\{m\}$. The operation "insert($m$)" gives INSERTS = INSERTS $\cup\ \{m\}$.

**merge:** To merge two states, we set-union the INSERTS and DELETES sets respectively

**query:** INSERTS − DELETES (set difference)

The internal state of this CRDT is the product of two grow-only set CRDTs or a "free product semi-lattice" of two set semi-lattices.

***Grow-only Counter CRDT.*** This is defined as follows:

**state:** A map k from unique replica IDs to natural numbers.

**update:** An update at replica with ID $j$ increments the value of $\mathsf{k}[j]$.

**merge:** Element-wise max of the map elements using the natural ordering of the natural numbers.

**query:** Sum over the values in the map.

The *max* operation forms a semi-lattice and the internal state of this CRDT is a map of these *max semi-lattices*. The query translates this internal structure into the desired output to the user - the total number of increments that have occurred globally in the system.

***Positive-Negative Counter CRDT.*** We define this CRDT as follows:

**state:** A map k from unique replica IDs to pairs of natural numbers.

**update:** An update at replica with ID $j$ increments one of the two natural numbers in the pair at key $j$ in the map. An update meant to decrement the global counter will increment the right number and an update meant to increment the global counter will increment the left number.

**merge:** Element-wise max of the map elements using the natural ordering of the pairs of natural numbers ($\mathbb{N}\times\mathbb{N}$).

**query:** Compute a new map $\mathsf{k}'[j] = \mathsf{k}_1[j] - \mathsf{k}_2[j]$ for all $j$. Then sum over values in this map.

The internal state of this CRDT is a map of free product semi-lattices of two *max* semi-lattices. The query translates this internal structure into the desired output to the user - the total number of increments that have occurred globally in the system minus the total number of decrements.

## 2.3 CALM History Background

The CALM Theorem is a research area that studies the role of the monotonicity property in the correctness and optimization of distributed systems. Originally conjectured by Joe Hellerstein at the PODS 2009 Keynote [49], the CALM Conjecture was based on observations while implementing distributed systems in Datalog-like logic languages such as Overlog [78] and NDLog [78]. The original conjecture states "A program has an eventually consistent, coordination-free execution strategy if and only if it is expressible in (monotonic) Datalog" [49].

A version of the CALM Conjecture was proven by a group of researchers in Belgium shortly after its original statement. The result is a theorem that proves a relationship between monotonic logic programs over relational data and the global invariant on outputs over time that tuples will not be retracted. In many settings, this guarantee on outputs can additionally be thought of as an opportunity to make partial progress on partial inputs. We give relevant background on the mathematical framework and proof techniques of this work in relevant sections of this thesis.

In the systems literature, the term "CALM Theorem" is often used to refer to a broader set of intuitions around the relationship between monotonicity and distributed systems preformance. To a theoretician, the "CALM Theorem" might refer specifically to the result proven about relational transducer networks in [10]. This creates a lot of confusion in the study of monotonicity for distributed systems. For example, the terms CRDT and CALM Theorem are sometimes used interchangeably. Further, the implications of calling this broader topic a "theorem" has discouraged further theoretical study in this space as the use of the term implies there is nothing left to prove. However, it is clear from this broader use of the term that there is appetite for a broader investigation beyond the original statement of the conjecture. To alleviate this problem, we now propose a rephrasing that distinguishes between the results that have already been mathematically proven in work on the CALM Theorem from the broader agenda of monotonicity for distributed systems. We hope that in clarifying the aspects of the larger agenda that are still conjecture, it will encourage further research in this area.

There are a few constraints in the phrasing of the original CALM Conjecture that we suggest to relax in the broader study of the topic that we dub the **Generalized CALM Agenda**. We propose a rephrasing of this research agenda to encompass a broad and ambitious set of problems in the spirit of the original conjecture. First, we wish to generalize to not just the monotonicity property, but also the inflationary property. Second, we wish to relax the sepecific use of "Datalog" in the conjecture. Monotonicity and the inflationary property can be defined for arbitrary languages and data structures, so there is no need to restrict the study to logic languages or the relational data model. The original conjecture being in the realm of Datalog also made it difficult to disentangle guarantees that are the result of the monotonicity property itself from guarantees that come from the use of the relational data model based on sets. Recall that sets (with respect to *set union*) are associative, commutative, and idempotent (ACI) structures and each of these properties has a role to play in distributed correctness and optimization.

Lastly, we believe that monotonicity has a role to play in the power of coordination-free systems beyond just being "eventually consistent". Observing the world of monotonic Datalog, we see numerous optimizations that are important for distributed systems that stem from mono-

tonicity and related properties. To state it broadly, the research agenda we propose is to study the role of monotonicity and the inflationary property as they relate to the optimizations and guarantees that can be offered in distributed systems. A non-exhaustive list of examples to start with include guarantees of convergence to fixpoint, semi-naive evaluation, incremental computation, horizontal partitioning, pipeline parallelism, sideways information passing, partial progress on partial inputs, disorderly evaluation, invariants on global outputs over time, and convergence of replicated states. Given that CRDTs are monotonically growing structures, their study is included in the Generalized CALM Agenda.

In this thesis, we do not attempt to prove all facets of the Generalized CALM Agenda. In Section 1, we clarify the relationship between the CALM Theorem, as proven [10], and CRDTs. We also build a bridge to the realm of arbitrary algebraic structures rather than logic programs over sets, which lets us further disentangle the roles of individual properties such as associativity, commutativity, and monotonicity in distributed data systems. The Free Termination Property that we introduce studies the relationship between monotonicity and the invariants on global outputs over time. This is the benefit of monotonicity that the CALM Theorem work [10] focused on, but just one of the many optimizations that the Generalized CALM Agenda is concerned with. We hope that the more general mathematical framework we introduce will ease the future study of the Generalized CALM Agenda.

# Part I

# Bridging Approaches to Coordination Avoidance

# Chapter 3

# The Free Termination Property of Queries Over Time

Building on prior work on distributed databases and the CALM Theorem, we define and study the question of *free termination*: in the absence of distributed coordination, what query properties allow nodes in a distributed (database) system to unilaterally terminate execution even though they may receive additional data or messages in the future? This completeness question extends beyond the soundness questions studied in the CALM literature. We also develop a new model based on semiautomata that allows us to bridge from the relational transducer model of the CALM papers to algebraic models that are popular among software engineers (e.g. CRDTs) and of increasing interest to database theory for datalog extensions and incremental view maintenance. Under this more general model, we are able to reason about both soundness and completeness in a unified light.

## 3.1 Introduction

A central technical challenge in distributed computing and databases is the use—and avoidance—of coordination mechanisms [21, 52]. Coordination in distributed systems is both slow and susceptible to unavailability via the CAP Theorem [21]. In response, theoreticians have studied the question of what programs are computable in a distributed fashion without the use of coordination, most notably in the CALM Theorem [9, 10, 14, 118] for coordination-free queries. Meanwhile, language designers and systems researchers have begun building systems that encourage coordination-free programming, exemplified by Conflict-free Replicated Data Types (CRDTs) [94, 102], which are popular data structure libraries based on semi-lattices.

In this work we push beyond CALM in two directions. The first is to consider a different proof goal. The work on CALM is fundamentally about a soundness property called *coordination-free eventual consistency*: in the absence of coordination, what properties of a program ensure that each node in a distributed system will emit only correct program outputs over time? Here we pose a complementary completeness question, *free termination*: in the absence of coordination, what properties of a program ensure that each node in a distributed system can unilaterally terminate after producing all its (correct) results, even if updates may arrive from other nodes in the future? From a practical perspective, free termination is critical to any user or client that requires a complete answer before proceeding.

The second ambition of this paper is to generalize the theory in this domain from its roots in relational transducers as explored in the CALM papers, and extend it to the context of the algebraic frameworks that are native to CRDTs, and of increasing interest in database research [3, 24, 61, 64, 66]. To that end, we introduce a general model that can capture what guarantees can be offered without coordination in both settings. Our model is based on queries over *semiautomata* and the guarantees to users are captured as properties of those semiautomata.

The main contributions of the paper are as follows:

1. Using semiautomata, we introduce the notion of free termination (Section 3.3) in a state transition system, and show how it can be used to model different types of applications, including incremental view maintenance [24] and the pre-semiring data model used to extend Datalog [3].

2. We then explore how the algebraic properties of the system and the query affect free termination (Section 3.4.1). Among our results, we show that under acyclic state modifications (commonly found in CRDTs), the only queries with free termination are a particular class of threshold queries over the natural partial order of states. We also show that if updates form a group or ring (e.g., in incremental view maintenance models), free termination cannot be achieved.

3. We study how to model coordination-free query computation in a distributed setting via the lens of free termination (Section 3.5). Interestingly, we show that by using the notion of free termination we can achieve a stronger and more fine-grained notion of coordination-freeness that applies to a pair of a query and an input. We show how coordination-freeness

for network transducers as defined in [10] is a specific case of our more general notion of coordination-freeness. This allows us to characterize other queries as coordination-free, for example, antitone queries.

4. Finally, we look into free termination when the state space is finite (Section 3.6). We give a linear-time algorithm for deciding all free termination states in a transition system and also study how to perform state minimization.

### 3.1.1 Motivating Examples

Before we proceed, we provide some motivating examples from the literature. We start with the domain of CRDTs, which have become quite popular with software engineers and shed light on both of our goals.

**Grow-Only Set CRDT:** A common CRDT is the "grow-only set" CRDT: this is a replicated distributed set where each machine propagates its local set to other machines in the system. Upon receiving a message containing the local set at another machine, the local machine will apply its "merge" operation, modifying its local state to be the union of its current state and the incoming set. This process of propagation over an asynchronous network introduces non-determinism as messages might be delivered in a different order than they are sent and messages may arrive multiple times. The fact that set union forms a semi-lattice ensures that, regardless of these sources of network nondeterminism, the state at each machine will eventually converge to the same value. That eventual value is the union of all of the initial states at each machine and the time at which it is reached is called the *quiescence point*.

Grow-only sets illustrate the point that CRDTs provide coordination-free consistency, but do not support free termination. In the absence of coordination, we do not have a mechanism for determining locally whether we have received all the elements in the network—i.e. whether we have reached a quiescence point. Without a guarantee of quiescence, what should we do to answer a query from a user? CRDTs allow arbitrary queries at any time and make no guarantee on the relationship between the query result at time $t$ and the query result at the quiescence point. For example, consider the query $Q(x) = R(x) - T(x)$. A value $(a)$ may be returned at time $t$, but eventually be excluded from the final output via subsequent "merges" into $T(a)$. This is not a particularly satisfactory contract between the system and the user: what good is distributed state if you do not know when you can query it reliably?

**Threshold Queries:** Threshold queries characterize a class of queries over CRDTs in which a machine is able to unilaterally detect that the output will never change regardless of future applications of the merge operation (set union in our example). For example, consider the query $Q() = |R(x)| > 10$. This boolean query is monotone with respect to the partial order of our semilattice (which is ordered by subset containment). Since the merge operation can only increase the position in the partial order, once the local state contains more than ten tuples in $R$, the result of the query is guaranteed to be the same at the quiescence time. By contrast, if the quiescence-time answer to this query is false—i.e. $\neg(|R(x)| > 10)$—then no machine will ever return an answer to this query. From a local perspective, a machine cannot know if there is some additional

element out there that it has not heard about and will someday need to union into its local state. Threshold queries, while useful, only offer free termination for database instances where the answer is true!

**CALM and Relational Transducers:** The CALM Theorem [10] uses relational transducer networks [2] to prove the relationship between queries expressed in monotonic logic and coordination-freeness. This formalism allows for the expression of distributed programs in terms of logical formulas that are evaluated iteratively on each machine and communicate state between machines. Similar models have been used by developers to implement distributed programs (e.g., Webdamlog [2], Bloom [7]). While both the CALM theorem and CRDTs use proofs based on notions of monotonicity, there remains a gap between the logic formalism of the CALM theorem and the state-based formalisms of other work on coordination-freeness. In this paper, we are not interested in the programmability differences between these approaches but exclusively in reasoning about guarantees on query results without coordination. In departing from the realm of logic programming, we want to reason naturally about programs beyond the boolean or relational setting. For example, consider a system that takes on integer values and supports the multiplication operation by a user-specified input integer. Now consider the query "is the running product divisible by two?". This query is not monotone with respect to the traditional ordering of the integers, but it can be computed coordination-free on input instances that contain any even number under the free-termination framework we introduce throughout this paper. Once we have multiplied by an even number, all possible future states will return True on this query.

As mentioned above, transducers were used to prove coordination-free consistency guarantees, but they do not directly address termination. As an example, consider a network of transducers supporting the Datalog language, and a simple program like transitive closure. Transducers accumulate knowledge about the set of paths in the global input, and once they learn about a path they can output it with certainty, but they will never conclude that they have finished finding new paths. Indeed, this depends on the database instance! We will return to this point in Section 3.5.

## 3.2 Related Work

The theory of what is computable without coordination in a distributed setting has been studied across different research communities and different theoretical models. In database theory, this has been studied in terms of the CALM Theorem ("Consistency as Logical Monotonicity") [9, 10, 14, 49, 118] using the relational transducer model of computation [2].

In related work in the programming languages community, conflict-free replicated data types (CRDTs) [103] achieve coordination-free programs via the algebraic properties of functions that can modify the state of data. CRDTs have found popularity amongst practicing software developers and are used in a variety of production software systems including Redis, Riak, ElectricSQL, SQLSync, Ditto, JupyterLab, and SoundCloud.

CRDTs have been criticized for the guarantees that they offer [69]. To resolve this gap, several systems have combined semilattice state convergence with monotonic queries or functions over that state, including Bloom$^L$ [31], Lasp [28, 81], Datafun [12] and Hydro [50].

For coordination-free termination detection, LVars [68] was early in proposing the use of monotone threshold functions from any semilattice $L$ to a smaller lattice like the booleans $(\mathbb{B}, \vee)$. If a computation exceeds a threshold in $L$, the threshold function evaluates to `true`. Since `true` is the top (supremum) of $(\mathbb{B}, \vee)$, nothing can change the result, so computation can safely terminate. This idea can be used in the languages mentioned above.

Efforts at adding non-monotonic functions or queries to these languages have typically fallen back to the use of coordination. Languages like Gallifrey [83] and Bloom$^L$ only guarantee consistency when non-monotone constructs are preceded by a round of coordination.

Beyond semilattices, recent research has explored alternative algebraic structures for data systems that each offer their own potential optimizations. Pre-semirings have been shown to offer semi-naive fixpoint evaluation in Datalog$^o$ [3, 61]. Abelian groups [24] and rings [64, 66] have been shown to enable efficient incremental computation of materialized views. While each of these algebraic structures has shown promise in database contexts, one cannot always make use of them simultaneously (Section 3.4.3).

In this work, we formalize the *free termination* property for certainty of query answers, generalize it beyond the case of monotonic state with threshold queries, and show directly how this property manifests in non-CRDT settings like Datalog, relational transducer networks, and algebraic models of incremental view maintenance.

## 3.3 Definitions of Free Termination

We will capture different scenarios in distributed computation using the general computational model of a *semiautomaton* [43]. We assume that each node keeps a *state* that is represented by an element $s$ in a (finite or infinite) state space $D$. Computation at each node is modeled by a modification of the state $s \in D$: the *transition (or update) function* $U : D \times L \to D$ takes a state $s$, a parameter $\ell \in L$ from some domain $L$ and outputs a state that $s$ can transition to. We will often write $s \xrightarrow{\ell} s'$ to denote that $U(s, \ell) = s'$ (we will often omit $\ell$ if it is not of importance).

**Definition 1** (Semiautomaton)**.** *A* semiautomaton *is a triple $\mathbb{S} = (D, L, U)$, where $D$ is a set called the state space, $L$ is a set called the parameter space, and $U : D \times L \to D$ is a total function called the update function.*

The update function $U$ can take different forms. A common scenario is when $L = D$, and then $U : D \times D \to D$ is a binary function. Another interesting case is when the update is *parameter-independent* , meaning the state transition is independent of the parameter $\ell \in L$. When the state space $D$ is finite, then $\mathbb{S}$ can be thought of as a deterministic finite automaton (DFA), but without the initial state or accept states.

A *computation trace* in a semiautomaton with *initial state* $s_0$ is a (possibly infinite) sequence of states $s_0 \to s_1 \to s_2 \to \ldots$. We say that $s \in D$ reaches a state $s' \in D$ if there is a finite sequence

$s \to \cdots \to s'$; we will use the shorthand notation $s \twoheadrightarrow s'$ for this and often say that $s$ reaches $s'$. We define $U^k(s)$ to be the set of states reachable from $s$ via a sequence of at most length $k$. We also define the *closure* of $s$, $U^\infty(s) = \bigcup_{k \geq 0} U^k(s)$ to be the set of states reachable from $s$.

**Definition 2** (Transition Graph)**.** *Given a semiautomaton $\mathbb{S} = (D, L, U)$, its transition graph $G[\mathbb{S}]$ is a labelled directed graph with vertex set $D$ and edge set $\{(s, s') : \ell \mid s' = U(s, \ell)\}$.*

The output of a computation in a semiautomaton model is captured by a query $Q$, which is a total function $Q : D \to R$ that maps each state to an element of a (finite or infinite) query range $R$. We can now define the main notion in this paper.

**Definition 3** (Free Termination State)**.** *Given a semiautomaton $\mathbb{S} = (D, L, U)$ and a query $Q : D \to R$, we say that a state $s \in D$ is a free termination state if for all states $s' \in U^\infty(s) : Q(s) = Q(s')$.*

In other words, if $s$ is a free termination state, any computation trace with initial state $s$ will leave the query result unchanged. This means that the distributed system can output the value of $Q$ without the need to continue the computation.

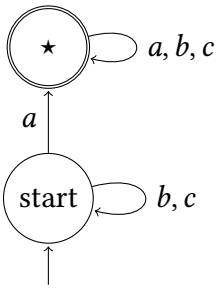
This model is inspired by algebraic models of data systems such as CRDTs and the group and ring models of incremental view maintenance [24, 64]. It is also inspired by application-level consistency [6, 33] which considers only the observable state to users (our query result) as important for application guarantees rather than the internal system state.

The behavior of free termination states can vary significantly. In the example below, we will present some common scenarios of free termination using DFAs; we will see some of these cases throughout the paper.

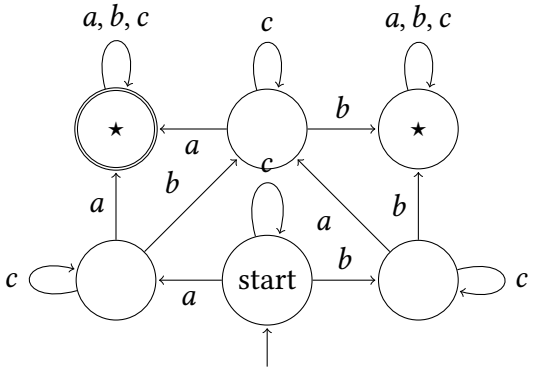
**Example 4.** To map a DFA into the model of queries over semiautomatons, the transitions are the same in both models, but the query $Q$ for the semiautomaton returns true if the DFA state is accepting, otherwise false. Figure 3.1 shows four different scenarios of free termination:

1. A free termination state is reachable from each state and all free termination states return the same query result (see Figure 3.1a).
2. All paths lead to a free termination state but different free termination states return different query results (see Figure 3.1b).
3. There are no free termination states (see Figure 3.1c).
4. Some paths lead to free termination states and others do not (see Figure 3.1d).

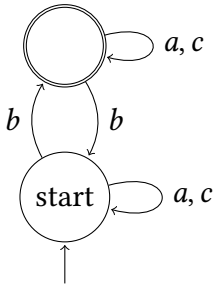
We will see that certain algebraic properties of the semiautomaton and the query imply that we have to fall in one of these cases. For example, if the query is commutative (Appendix 3.4.2), then free termination falls in the first scenario, while if the semiautomaton forms a group structure this means that no free termination state exists (Section 3.4.3).



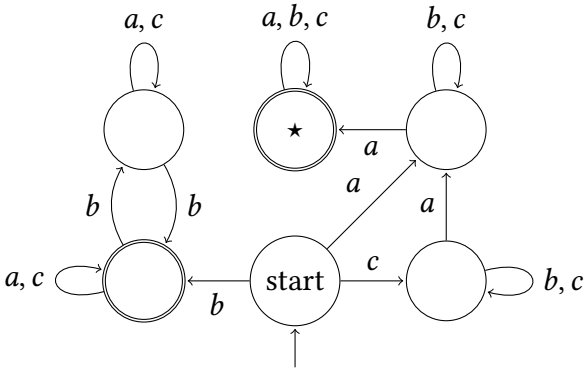

(a) Contains an $a$.




(b) Two $a$'s occur and if there are two $b$'s then the two $a$'s occur before the second $b$.




(c) Contains an odd number of $b$'s.




(d) Starts with a $b$ and has an odd number of $b$'s or doesn't start with a $b$ and contains two $a$'s.

Figure 3.1: Four DFAs with labels $\{a, b, c\}$ showing four different categories of free termination. The doubly-lined states represent the accepting states of the DFA (query is true).
The stars represent free terminating states.

### 3.3.1 Applications of Free Termination

We will next describe a few concrete instantiations of our computational model, which will be used throughout this paper.

***Incremental Query Computation.*** In the first instantiation, we consider $D$ to be the set of all instances over a fixed relational schema $\mathbf{R}$. The update function $U$ allows modification only by inserting a tuple to the instance; in other words, the set $L$ is the set of all tuples over $\mathbf{R}$ and $U(I, t) = I \cup \{t\}$. We will refer to this semiautomaton as $\mathbb{S}_{\cup}$. A trace in $\mathbb{S}_{\cup}$ is a sequence of single-tuple insertions to the initial database. Let us consider a query $Q$ that is a Boolean Conjunctive Query, hence it maps the state of the database (i.e. the "database instance") to its range $R = \{\mathsf{T}, \mathsf{F}\}$.

The characterization of free termination states for $Q$ is fairly straightforward. Indeed, take any instance $I$ in the state space such that $Q(I)$ is true; then, because $Q$ is a monotone query and the updates are only tuple insertions, any $I'$ reachable from $I$ will also have $Q(I') = \mathsf{T}$. Thus, such an $I$ will be a free termination state. On the other hand, if $Q(I) = \mathsf{F}$, free termination is not possible since we can always insert a sequence of tuples to make $Q$ true. As we will see later on, this is a special case of a more general characterization of free termination.

***State-based CRDTs.*** Recall that a state-based CRDT defines three functions over its state $D$: an *update* operation that allows state to be mutated from outside the system (update : $D \times L \to D$), a *merge* operation that determines how the states of two replicas can combine to converge to the same state (merge : $D \times D \to D$), and a *query* operation that defines what is visible to the user from the internal state ($Q : D \to R$).

To capture a state-based CRDT in our model, we define it as a semiautomaton $\mathbb{S}_{\square} = (D, L, U)$, where $D$ is the set of states, and the update $U$ captures both the update and merge operation of the CRDT. Specifically, $U(s, \langle \mathsf{u}, \ell \rangle) = \mathsf{update}(s, \ell)$ and $U(s, \langle \mathsf{m}, s' \rangle) = \mathsf{merge}(s, s')$. A crucial property of a state-based CRDT is that the partial order induced by the merge operation forms a join-semilattice. Observe that because of this, the state transition system for a state-based CRDT will always be acyclic - a property that we will explore in detail in Section 3.4.1.

In Section 3.7, we discuss how the four popular CRDT examples from the thesis background section relate to free termination.

***Fixpoint Computation.*** Consider a partial order $\sqsubseteq$ over a domain $D$, and take $f : D \to D$ to be a monotone function w.r.t. $\sqsubseteq$. Further consider a (parameter-independent) semiautomaton and assume that it captures a fixpoint computation from a starting state $s_0 \to f(s_0) \to f(f(s_0)) \to \ldots$ that eventually reaches a state $s$ with $f(s) = s$ after finitely many steps. For any query $Q$ over $D$, the fixpoint state $s$ is a free termination state (no other state is reachable from it). However, the structure of $Q$ may allow us to find an earlier free termination state before we even reach the fixpoint.

As an example, consider the case where the fixpoint computation is an iterative (naive or semi-naive) evaluation of a Datalog program $P$ on an instance $I$. Here, the starting state is the set of EDB facts and each iteration during evaluation is an update that adds to the current state the newly produced IDB facts via the rules of the program. Our query $Q$ is a view over the IDB facts of the program $P$, which can be thought of as the "target" of $P$. Concretely, consider the following Datalog program that determines whether there exists a path between vertices $s$ and $t$

in a graph.

$$\begin{aligned} P(x,y) &\leftarrow Edge(x,y) \\ P(x,y) &\leftarrow P(x,z), Edge(z,y) \\ Q() &\leftarrow P(s,t) \end{aligned}$$

For this program, we can freely terminate as long as the state contains the IDB fact $P(s,t)$, even before we have reached a fixpoint. This is because the fixpoint computation is monotone w.r.t. set containment (so the update $U$ is inflationary) and $Q$ is a monotone query as well w.r.t. the partial order $\mathsf{F} \sqsubseteq \mathsf{T}$. If we replaced $Q$ with another query $Q'() \leftarrow P(x,x)$ which detects the presence of a cycle in the graph, we also can freely terminate earlier as long as the state contains a cycle. As we will see in the next section, both of these are examples of Boolean threshold queries.

Datalog$^o$ [3] is an extension of Datalog to support recursive queries over partially ordered pre-semirings (POPS). This can also be viewed as a parameter-independent fixpoint computation. Datalog$^o$ requires that both semiring operations be monotone w.r.t. the partial order of the POPS. This tells us that the update transition for any Datalog$^o$ graph is inflationary, which we show in Proposition 13 implies any Boolean monotone query will have a free termination state. For instance, the query $Q$ below that computes whether the distance between vertices $s, t$ in a graph is at most 10.

$$\begin{aligned} P(x,y) &\leftarrow min(Edge(x,y), min\{(P(x,z) + Edge(z,y)))\} \\ Q() &\leftarrow P(s,t) \leq 10 \end{aligned}$$

## 3.4 Algebraic Properties and Free Termination

In this section, we explore how the algebraic structures of the semiautomaton and the query affect free termination.

### 3.4.1 Partial Orders

Consider the example in Figure 3.2 of a semiautomaton in which update labels are singleton sets ($\{a\}$, $\{b\}$, or $\{c\}$) and the update applies set union of the incoming singleton to the current state. Labels on edges depict the incoming singleton that transitions from the source state to the destination state. We can think of this system as inputs from ($\{\{a\}$, $\{b\}$, or $\{c\}\}$) streaming in over time and we wish to compute some query over this stream of inputs without ever knowing if the stream has ended. The figure depicts the query "contains an $a$" over this semiautomaton, illustrated by the dotted green line that is the "threshold" after which states in the partial order return True in the query. Observe that the same graph with the edge labels and self-loops removed is the Hasse diagram [19] of the set/subset partial ordering over atoms $\{a, b, c\}$.

We will utilize properties of this example to reason about its free termination states. In this particular case, the update function is *inflationary*. If we view the ordering of $\mathbb{B}$ as $\mathsf{F} \sqsubseteq \mathsf{T}$ then

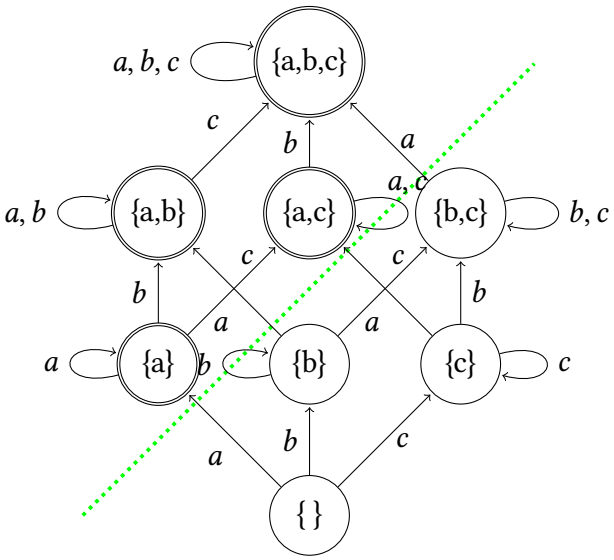



Figure 3.2: A state transition system for the set union state transition over the universe $\{a, b, c\}$. The green dotted line across the states indicates the threshold line for the query "the set contains an $a$". The doubly-lined states return True and single-lined states return False in the query.

we can see that $Q$ is *monotone*. Formally, to define what it means for $Q$ to be monotone we must have some partial order on the sets $D$ and $R$. Recall that a binary relation $\sqsubseteq$ is a partial order if it is reflexive, transitive and antisymmetric.

**Definition 5** (Inflationary)**.** *Let $\mathbb{S} = (D, L, U)$ be a semiautomaton equipped with a partial order $\sqsubseteq_D$ on $D$. $\mathbb{S}$ is* inflationary (resp. deflationary) *w.r.t $\sqsubseteq_D$ if whenever $s \to s'$ then $s \sqsubseteq_D s'$ (resp. $s \sqsupseteq_D s'$).*

In an inflationary semiautomaton, any update of a state will always follow the underlying partial order of the state space. A deflationary system will follow the partial order in reverse.

In many cases, we can define a "natural" partial order on $D$ via the update function $U$. We say that $U$ is *acyclic* in $\mathbb{S}$ if the transition graph $\mathrm{G}[\mathbb{S}]$ is acyclic (excluding self-loops). The following proposition follows from the fact that the transitive closure of a directed acyclic graph is a partial order.

**Proposition 6.** *Let $\mathbb{S} = (D, L, U)$ and $U$ be acyclic. Then, the relation $s \sqsubseteq_U s' \Leftrightarrow s \twoheadrightarrow s'$ is a partial order for $D$. Moreover, $\mathbb{S}$ is inflationary w.r.t. $\sqsubseteq_U$.*

Whenever $U$ is acyclic, we will refer to the partial order $\sqsubseteq_U$ as the *natural partial order* of $\mathbb{S}$. This is akin to the notion of natural orders for algebraic structures such as pre-semirings [3]. If the update transition is a binary operation $U : D \times D \to D$ forming a monoid (is associative and has an identity) then the natural ordering would be the standard $x \sqsubseteq_U y$ iff $\exists z : x \xrightarrow{z} y$.

**Example 7.** Consider again the incremental query computation semiautomaton $\mathbb{S}_{\cup}$. The update operation is acyclic (since an update only adds tuples to the instance), and the natural partial order is set containment of the states.

**Definition 8** (Monotone Query)**.** *Let $\mathbb{S}$ be a semiautomaton equipped with a partial order $\sqsubseteq_D$ on $D$. Let $R$ be a set with partial order $\sqsubseteq_R$. A query $Q : D \to R$ is* monotone (resp. antitone) *w.r.t. $\mathbb{S}$ if $s \sqsubseteq_D s'$ implies $Q(s) \sqsubseteq_R Q(s')$ (resp. $Q(s) \sqsupseteq_R Q(s')$).*

***Free Termination Conditions.*** Intuitively, if the input state will only increase in the partial order because $U$ is inflationary, and $Q$ is monotone, then over time the output of $Q$ will always stay the same or increase. However, free termination is concerned specifically with when the output of $Q$ will *stay the same*. We identify two general conditions that guarantee free termination in this case.

**Proposition 9.** *Let $\mathbb{S} = (D, L, U)$ be a semiautomaton equipped with a partial order $\sqsubseteq_D$. If $\mathbb{S}$ is inflationary (resp. deflationary) and $s \in D$ is a maximal (resp. minimal) element of $\sqsubseteq_D$, then $s$ is a free termination state.*

*Proof.* Consider any state $s' \in D$ such that $s' \in U^{\infty}(s)$. Then, because $\mathbb{S}$ is inflationary and $\sqsubseteq_D$ is transitive, we must have that $s \sqsubseteq_D s'$. But $s$ is a maximal element of $\sqsubseteq_D$, hence $s = s'$. Thus, $Q(s) = Q(s')$. The proof for when $U$ is deflationary and $s$ a minimal element is symmetrical. □

In some cases, we will consider partial orders $\sqsubseteq_D$ with a bottom element $\bot$ or a top element $\top$: then, we have that for every $s \in D$, $\bot \sqsubseteq_D s$ or $s \sqsubseteq_D \top$ respectively. Note that if a top element exists, then $\top$ is a free termination state by the above proposition.

**Proposition 10.** *Let $\mathbb{S} = (D, L, U)$ be a semiautomaton with a partial order $\sqsubseteq_D$. Let $Q : D \to R$ be a query, with $R$ equipped with a partial order $\sqsubseteq_R$. If $\mathbb{S}$ is inflationary, $Q$ is monotone (resp. antitone), and $Q(s)$ is a maximal (resp. minimal) element of $\sqsubseteq_R$, then $s \in D$ is a free termination state.*

*Proof.* Consider any state $s' \in D$ such that $s' \in U^{\infty}(s)$. Then, because $\mathbb{S}$ is inflationary and $\sqsubseteq_D$ is transitive, we must have that $s \sqsubseteq_D s'$. From the monotonicity of $Q$, this implies that $Q(s) \sqsubseteq_R Q(s')$. But because $Q(s)$ is maximal, we must have that $Q(s') = Q(s)$. The case where $Q$ is antitone and $Q(s)$ is a minimal element is symmetric. □

**Example 11.** A classic example of an antitone query over $\mathbb{B}$ is the use of a universal quantifier $\forall$. Consider the natural inverse of the existential quantifier $\exists$ used in our first example. If we let $Q$ be "every element of the input stream is an $\{a\}$" then we can freely terminate when the answer becomes $\mathsf{F}$ (as soon as any non-"a" streams in).

***Threshold Queries.*** In this part, we ask the following question: what classes of queries have free termination states? We will restrict our attention to settings where the behavior of $\mathbb{S}$ is inflationary as threshold queries are naturally valuable in this setting. Further, all state-based CRDTs are examples of $\mathbb{S}$ that are inflationary. We define first an important class of queries, inspired by the LVars [68] work on coordination-free programming languages, called Threshold

Queries. Recall that an *antichain* in a partial order $\sqsubseteq$ with domain $D$ is a subset $C \subseteq D$ such that no two distinct elements of $C$ are comparable under $\sqsubseteq$.

**Definition 12** (Boolean Threshold Query)**.** *Let $\sqsubseteq$ be a partial order with domain $D$, and $C \subseteq D$ be an antichain. We will refer to an antichain $C$ as a* threshold line*. A Boolean threshold query $Q_C : D \to \mathbb{B}$ with threshold line $C$ is the Boolean query that returns the value $Q_C(s) = \bigvee_{c \in C}(s \sqsupseteq c)$.*

Boolean threshold queries are monotone and this allows us to obtain the free termination states in this case.

**Proposition 13.** *Let $\mathbb{S} = (D, L, U)$ be an inflationary semiautomaton equipped with a partial order $\sqsubseteq_D$. Let $Q_C$ be a Boolean threshold query. Then, the free termination states are exactly the elements of $D$ that are at or above $C$.*

*Proof.* We first show that $Q_C$ must be monotone. Indeed, let $s \sqsubseteq_D s'$ and assume $Q_C(s)$ is true. Then, there exists $c \in C$ such that $s \sqsupseteq_D c$. But since $s \sqsubseteq_D s'$, $s' \sqsupseteq_D c$ and $Q_C(s')$ must also be true. Since $\mathsf{T}$ is the maximal element of $\mathbb{B}$, all states $s$ such that $Q_C(s)$ is true are free termination states. Note that any element of $D$ that is at or above the threshold line $C$ has this property. □

We should note here that any monotone Boolean query must be a Boolean threshold query (excluding the trivial query that is always false). In this work, we do not wish to limit ourselves to threshold queries that return only true, or even to thresholds where each $c$ returns the same value $Q(c)$. For example, queries that freely terminate to True on some inputs but freely terminate to False on other inputs. We can generalize the concept of Boolean threshold queries with the following result.

**Proposition 14.** *Let $Q$ be a query with at least one free termination state for a semiautomaton $\mathbb{S} = (D, L, U)$ where $U$ is acyclic. Then, there exists an antichain $C \subseteq D$ w.r.t. the natural partial order $\sqsubseteq_U$ such that whenever $c \sqsubseteq_U s$ for some $c \in C$, $Q(s) = Q(c)$.*

*Proof.* Let $F \subseteq D$ be the set of free termination states. $F$ cannot be empty. Let $F_\tau$ be the minimal states in $F$ w.r.t. $\sqsubseteq_U$. By construction, $F_\tau$ forms an antichain and will be our set $C$. Take now any state $s \in D$ such that $c \sqsubseteq_U s$ for some $c \in C$. Since $c$ is a free termination state and $c$ reaches $s$, it must be that $Q(c) = Q(s)$. □

In other words, there exists a threshold (formed by the antichain $C$) such that at or above the threshold the behavior of $Q$ is governed completely by the threshold states. However, the behavior of $Q$ outside of this threshold space has no restriction. This is unlike the case of Boolean monotone queries where the behavior outside of the threshold space must also respect monotonicity.

***Join Semilattices.*** When the partial order has further algebraic structure, we can characterize the behavior of free termination states more precisely. Here, we consider the case where $\mathbb{S}$ has a natural partial order $\sqsubseteq_U$ and this order is a join-semilattice (this means that there is a least upper bound for any nonempty finite subset of $D$).

**Proposition 15.** *Let $\mathbb{S} = (D, L, U)$ be a semiautomaton where $\sqsubseteq_U$ forms a join-semilattice. Then, for any query $Q$, all free termination states return the same value.*

*Proof.* Indeed, take any two free termination states $s_1, s_2$. Since $\sqsubseteq_U$ is a join-semilattice, there exists a least upper bound $s$ such that $s_1 \sqsubseteq_U s$ and $s_2 \sqsubseteq_U s$. Since $\sqsubseteq_U$ is the natural partial order, $s_1 \twoheadrightarrow s$ and $s_2 \twoheadrightarrow s$. But $s_1, s_2$ are free termination states, so it must be that $Q(s_1) = Q(s)$ and $Q(s_2) = Q(s)$. Thus, $Q(s_1) = Q(s_2)$. □

As a corollary of the above proposition, any free termination state in a state-based CRDT must have the same value. A special case is when the natural partial order has a top element $\top$; in this case, all free termination states must take the value of $Q(\top)$.

**Proposition 16.** *Let $\mathbb{S} = (D, L, U)$ be a semiautomaton where $\sqsubseteq_U$ forms a join-semilattice. If a query $Q$ has a free termination state in $\mathbb{S}$, then any state can reach a free termination state.*

In other words, if there exists a free termination state, it is possible to reach a free termination state from whichever state we currently are in (with an appropriate set of updates).

*Proof.* Let $s$ be the current state and $s_t$ be a free termination state. Since $\sqsubseteq_U$ is a join-semilattice, there exists a least upper bound $s'$ such that $s \sqsubseteq_U s'$ and $s_t \sqsubseteq_U s'$. Since $\sqsubseteq_U$ is the natural partial order, $s \longrightarrow s'$ and $s_t \longrightarrow s'$. But then $s'$ must also be a free termination state. □

One would be tempted to think that monotone (or antitone) queries are the only ones that have free termination states in a join-semilattice, but this is not true even for Boolean queries. Take for example the semiautomaton $\mathbb{S}_\cup$ with the Boolean query $Q() = R(c) \wedge \neg S(c)$ for some constant $c$. This query is neither monotone nor antitone, since, for example, it returns false on $\{R(a)\}$, true on $\{R(a), R(c)\}$, and false on $\{R(a), R(c), S(c)\}$. However, it has free termination states: these are the states in which the tuple $S(c)$ is in the instance.

### 3.4.2 Commutativity

Another algebraic property that many distributed systems satisfy is that of commutativity. It will be convenient to switch from polish prefix notation to infix notation for our update transitions. We use $\cdot$ for the application of an update transition so $U(s, a)$ becomes $s \cdot a$ and $U(U(s, a), b)$ becomes $s \cdot a \cdot b$. Given an ordered sequence of labels $\mathbf{a} = a_1, a_2, ..., a_k$ we will also use the shorthand $s \cdot \mathbf{a}$ to mean $S \cdot a_1 \cdot a_2 \cdot ... \cdot a_k$.

**Definition 17** (Commutativity)**.** *Let $\mathbb{S} = (D, L, U)$ be a semiautomaton, and $Q : D \longrightarrow R$ be a query. We say that $U$ is* commutative *(resp. $Q$ is commutative) if for any state $s \in D$ and any sequences of labels $\mathbf{a}, \mathbf{b}$ from $L$, we have $s \cdot \mathbf{a} \cdot \mathbf{b} = s \cdot \mathbf{b} \cdot \mathbf{a}$ (resp. $Q(s \cdot \mathbf{a} \cdot \mathbf{b}) = Q(s \cdot \mathbf{b} \cdot \mathbf{a})$).*

Commutative update implies commutative query, but the converse does not hold. A query may be indifferent to the order of updates but the state itself may be order-sensitive. For instance, the query counting the number of "a"s in a string with string concatenation as the update operation.

Insertion of tuples into a database instance is a commutative update. An update that allows insertion or deletion of tuples over a set-semantics database is not commutative; however insertion or deletion of tuples over a $\mathbb{Z}$-set semantics database is commutative (and forms an abelian group) [24].

**Proposition 18.** *Let $\mathbb{S} = (D, L, U)$ be a semiautomaton with a bottom state $\bot$, and $Q$ be a commutative query. Then, all free termination states return the same value for $Q$.*

*Proof.* Take any two free termination states $s_1, s_2 \in D$. Let $\mathbf{a}_1$ be a sequence of updates such that $\bot \cdot \mathbf{a}_1 = s_1$ and let $\mathbf{a}_2$ be a sequence of updates such that $\bot \cdot \mathbf{a}_2 = s_2$. Because $Q$ is commutative, we have $Q(\bot \cdot \mathbf{a}_2 \cdot \mathbf{a}_1) = Q(\bot \cdot \mathbf{a}_1 \cdot \mathbf{a}_2) = r$. Since $s_1$ is a free termination state and $s_1 \longrightarrow \bot \cdot \mathbf{a}_1 \cdot \mathbf{a}_2$, we have $Q(s_1) = Q(\bot \cdot \mathbf{a}_1 \cdot \mathbf{a}_2) = r$. Similarly, $s_2$ is a free termination state and $s_2 \longrightarrow \bot \cdot \mathbf{a}_2 \cdot \mathbf{a}_1$, so $Q(s_2) = r$. Thus, $Q(s_1) = Q(s_2)$. □

**Proposition 19.** *Let $\mathbb{S} = (D, L, U)$ be a semiautomaton with a bottom state $\bot$, and $U$ be commutative. If a query is freely terminating, then any state can reach a free termination state.*

*Proof.* Let $s$ be the current state. Since $Q$ is freely terminating, there exists a free termination state $s_t$. Let $\mathbf{a}_t$ be an update sequence such that $\bot \cdot \mathbf{a}_t = s_t$, and $\mathbf{a}$ an update sequence such that $\bot \cdot \mathbf{a} = s$. From the commutativity of $U$, we have that $\bot \cdot \mathbf{a}_t \cdot \mathbf{a} = \bot \cdot \mathbf{a} \cdot \mathbf{a}_t = s'$. Since $s_t \longrightarrow s'$, $s'$ must be a free termination state. But then we also have that $s \longrightarrow s'$. □

### 3.4.3 Group-Like Structures

If $U$ is not acyclic, then we can construct examples where free termination is not possible. We will use the notions of inverses and identity values from groups, but to generalize beyond binary update operations $U : D \times D \longrightarrow D$ we must define these terms for the general case of transition graphs.

**Definition 20.** *Let $\mathbb{S} = (D, L, U)$ be a semiautomaton. A state* $\mathsf{id} \in D$ *is called an* identity state *if* $\mathsf{id} \twoheadrightarrow s$ *for every $s \in D$. A state $s \in D$ is called* invertible *if it can reach an identity state.*

We call the following theorem "the inverse curse theorem".

**Theorem 21.** *Let $\mathbb{S} = (D, L, U)$ be a semiautomaton, and $Q : D \longrightarrow R$ be a non-constant query. If every state of $D$ is invertible, then $Q$ has no free termination states in $\mathbb{S}$.*

*Proof.* Suppose that some state $s \in D$ freely terminates. Since $Q$ is not constant, there exists some state $s' \in D$ such that $Q(s) \neq Q(s')$. Since $s$ is invertible, we have that $s \twoheadrightarrow \mathsf{id}$, where $\mathsf{id}$ is an identity element. By the definition of an identity element, $\mathsf{id} \twoheadrightarrow s'$. Hence, $s \twoheadrightarrow s'$, which contradicts the fact that $s$ is a free termination state. □

**Example 22.** As an example, consider the case where $D = \mathbb{Z}$, and let $U(i, +) = i + 1$ and $U(i, -) = i - 1$, i.e., we have a counter that can be incremented or decremented. Then, any state in $D$ is an identity state and invertible, so any query $Q$ has no free termination states unless it is constant.

When $U : D \times D \to D$ and $(D, U)$ forms a group ($U$ has identity element and every element has an inverse), then we obtain the following corollary:

**Corollary 23.** *Let $\mathbb{S} = (D, L, U)$ be a semiautomaton such that $(D, U)$ forms a group. If $Q$ is not a constant query, then $Q$ has no free termination states.*

This corollary tells us that an update function that forms a group precludes the possibility of free termination. It also tells us that in view maintenance, which often studies rings rather than semirings, free termination is impossible. This is consistent with the monotonicity lens on threshold queries from CRDTs [69] where an inverse corresponds to moving backward in the partial order, which prevents the possibility of threshold queries with free termination states. The practical benefits of having inverses in CRDTs to allow "undo" operations have been discussed in [39] and [93]. This is an interesting dichotomy. Two parallel lines of work have shown the value of invertibility in data systems (DBSP [24], DBToaster [66]) and the value of coordination-free monotone queries (CALM theorem, CRDTs), but the benefits of these properties appear mutually exclusive. We will discuss these topics further in Chapter III.

## 3.5 Free Termination in Distributed Systems

In this section, we will study the problem of distributively computing a query in a coordination-free manner via the lens of free termination. Both transducer networks [10] and CRDTs offer coordination-free models of eventual convergence, i.e., the existence of a time (called the quiescence point) where the query result converges. However, without the ability of a machine to determine whether it has already reached such a quiescence point, the applicability of this notion in practical systems is limited [69]. If a user wants to read the (complete) output of a transducer network, eventual consistency cannot provide a certain answer.

By definition, free termination can determine when a complete answer can be given with certainty and hence aligns with the need for distributed systems to promptly respond to user requests. However, this stronger guarantee warrants a more fine-grained definition of coordination-freeness. In particular, we will need to talk about coordination-freeness as a property of a query and its input. This is because if a query were to freely terminate on every input, it would mean that the inputs do not affect the query result. This is in contrast to the CALM Theorem [10], which talks about coordination-freeness as a property of an entire query but offers a weaker guarantee than Free Termination.

### 3.5.1 Distributed Model

Our task is to compute a query $Q$ over a relational instance $I$ defined over a schema $\mathbf{R}$ over a network $\mathcal{N}$. A network $\mathcal{N}$ is defined as a finite, connected, undirected graph over a set of vertices $V$. Initially, the instance $I$ is horizontally partitioned across the nodes in the network.

The distributed computational model we consider here is a simplified version of the relational transducer network model used by Ameloot et al. In particular, we leverage the equivalence

between oblivious transducers and coordination-freeness proven in [10] to focus on oblivious networks and specifically a type of communication protocol used by many constructions in [10] called *network flooding* . In this construction, nodes attempt to achieve eventual consistency by broadcasting all their local information to their neighbors and sending no other messages. More precisely, the computation in each node will be captured by the semiautomaton $\mathbb{S}_{\cup}$, where each state is an instance over $\mathbf{R}$ and each transition adds a new tuple in the instance. This roughly corresponds to an oblivious, inflationary, and monotone relational transducer, with the critical difference that there is no output generated – only the state is modified. Nodes can communicate by sending a tuple from their local state to be added to the instance of any neighboring node.

For now, we will focus on Boolean queries. Formally, a *configuration* of the network is a triple of mappings $\gamma = (state, buf, ready)$, where $state$ maps each node in $V$ to a state in $\mathbb{S}_{\cup}$, $buf$ maps each node in $V$ to a finite multiset of facts from $\mathbf{R}$ called a buffer, and $ready$ maps each node to $\{\mathsf{F}, \mathsf{T}\}$. Initially, each state contains only the tuples in $I$ from the local partition, the buffers are empty, and $ready(v) = \mathsf{F}$. There are three types of transitions between two configurations:

**Produce Transition:** A node can move any tuple in its local state to the buffers of its neighboring nodes;

**Consume Transition:** A node can update its state by removing a tuple from its local buffer and adding it to its local instance.

**Ready Transition:** A node can set $ready(v) \leftarrow \mathsf{T}$.

Importantly, once $ready$ is set to true, it cannot be further modified. Setting $ready$ to true denotes that the query result will not change. An algorithm for $Q$ in this model determines two things: $(i)$ when to send each tuple to its neighbors, and $(ii)$ if and when to perform a ready transition.

A *run* $\rho$ is an infinite sequence of transitions starting from an initial configuration. A run is *fair* if every fact in every buffer is eventually taken out, and it is *complete* if every fact in a state is eventually sent to its neighbors. Finally, we say that an algorithm computes the correct output of $Q$ if for all inputs $I$ and all horizontal partitionings of $I$, whenever $ready(v) = \mathsf{T}$, $Q(state(v)) = Q(I)$. In every infinite run, there is a natural number $n > 0$ such that none of the states change after the $n$-th transition: we call this the *quiescence point*. In a fair and complete run, each local state eventually converges to $I$, and hence at the quiescence point $state(v) = I$ for every node $v \in V$. However, without coordination, it is not possible to know when a node has received the entire input.

**Definition 24** (Coordination-Free Correctness)**.** *We say that the pair* $(Q, I)$ *is* coordination-free correct *if there exists an algorithm such that in every fair and complete run the correct output of* $Q$ *is computed and* $ready$ *is set to true at all nodes.*

In other words, no matter how computation proceeds and how messages are exchanged, at some point, the algorithm will perform the ready transition and hence know (without coordination) that it has computed the correct result and thus can give it to the user. The next proposition relates the above notion of fine-grained coordination-freeness to free termination.

**Theorem 25.** *$(Q, I)$ is coordination-free correct if and only if $I$ is a free termination state for $Q$ in the semiautomaton $\mathbb{S}_\cup$.*

*Proof.* Suppose $I$ is a free termination state. Consider the following algorithm: it will set $ready(v)$ to true exactly when $state(v)$ is a free termination state for $Q$ in $\mathbb{S}_\cup$. This algorithm is correct, since at the instance $I'$ where $ready$ becomes true, all reachable states maintain the result of $Q$. Since $I$ is reachable from $I'$ (from the network flooding construction), $Q(I) = Q(I')$. Further, every fair and complete run reaches the quiescence point, when $state(v) = I$, and thus the algorithm will set $ready$ to true.

In the other direction, suppose $(Q, I)$ is coordination-free correct and consider an algorithm that computes the output correctly. At the quiescence point, when $state(v) = I$, this algorithm must set $ready$ to true (since the state will remain unchanged from that point on). But now consider a fair and complete run for another input $I' \supseteq I$ (which is a reachable state in $\mathbb{S}_\cup$) such that some node $v$ receives first all of $I$. At this point, the algorithm would need to do the ready transition. But because of correctness, it must be that $Q(I) = Q(I')$. Hence, $I$ is a free termination state. □

If the input is not a free termination state, the system will often need to perform some coordination to get the user a concrete answer. We thus avoid coordination for a given query *on some inputs, but not all inputs!* In practice, the distribution of inputs to a given system is what determines how helpful free termination is.

We next discuss how we could define a notion of coordination-free correctness for the entire query $Q$. As a first attempt, we could define $Q$ to be coordination-free correct if $(Q, I)$ is coordination-free correct for every input $I$. From Theorem 25, this is equivalent to saying that every input is free terminating, which happens only when $Q$ is a constant query. Hence, we need to slightly relax this notion, by requiring that $(Q, I)$ is coordination-free correct for some inputs.

**Definition 26.** *We say that a Boolean query $Q$ is* positively (resp. negatively) coordination-free *if $(Q, I)$ is coordination-free correct for every input $I$ such that $Q(I) = \mathsf{T}$ (resp. $Q(I) = \mathsf{F}$).*

The notion of positive coordination-freeness is exactly the notion of query coordination-freeness used for transducer networks in [10]. Indeed, a transducer can only write true in its output tape, so if nothing is written, we assume a false output.

**Theorem 27.** *A Boolean query $Q$ is positively (resp. negatively) coordination-free if and only if $Q$ is monotone (resp. antitone).*

*Proof.* From Theorem 25, $Q$ is positively coordination-free if and only if every state $I$ with $Q(I) = \mathsf{T}$ is a free termination state. Say $I \subseteq I'$ and suppose $Q(I)$ is true. Then, $I$ is a free termination state and thus $I'$ is as well, which implies $Q(I') = Q(I) = \mathsf{T}$. Hence, $Q$ is monotone. The antitone case is symmetric. □

For instance, consider the Boolean query $\forall x : R(x) > 0$. This query freely terminates on all false instances, and hence it can be considered negatively coordination-free. Observe that

Ameloot et al. [10] categorizes antitone Boolean queries as not being computable by oblivious transducer networks. This is because of the definition of the output of a transducer network being the union of outputs and the encoding of the boolean values True and False being the presence of an empty tuple and the absence of a tuple respectively. In addition to antitone queries, some queries are neither monotone nor antitone, but still have coordination-free correct inputs such as our example from Section 3.4.1, $Q() = R(c) \wedge \neg S(c)$.

***Non-Boolean Queries.*** Consider now a non-Boolean query $Q$ that outputs a set. In this case, we will introduce a ready variable $ready(v, t)$ for every node $v$ and every potential tuple $t$. Note that this introduces another layer of granularity since we can now compute some tuples in a coordination-free correct manner, while others we cannot. We can similarly lift coordination-free correctness to the entire query $Q$ by saying that $Q$ is positively (resp. negatively) coordination-free if every Boolean query $(t \in Q(I))$ is positively (resp. negatively) coordination-free. Positive coordination-freeness captures exactly how oblivious relational transducers work: they can write only correct facts to the output tape, which they can never retract. The following result is immediate.

**Theorem 28.** *A non-Boolean query $Q$ is positively (resp. negatively) coordination-free if and only if $Q$ is monotone (resp. antitone).*

We complete this section by an example that shows the benefits of a fine-grained coordination-free definition. Consider the following query $Q$, which is neither monotone nor antitone: $Q(x) = (R(x) \wedge x > 10) \vee (S(x) \wedge \neg T(x))$. In this case, any instance that contains a tuple $R(20)$ freely terminates (with true) for the output tuple $(20)$. Also, any instance that contains $(T(5))$ freely terminates (with false) for the output tuple $(5)$. Thus, we can correctly output the existence/non-existence of the two tuples in a coordination-free manner.

### 3.5.2 Distributed Computation with Metadata

Many distributed systems depend on some form of metadata to infer facts about what state transitions may be possible in the future. The essential such example from [10] is the all() relation which returns the IDs of all transducers in a network. all() is used to compute non-monotone queries and can also be modeled with free termination.

To encode the all() relation, we extend the schema $\mathbf{R}$ to include a nullary relation $All()$ that will be set to true if we know that all machines have sent all their local data. Formally, we will extend $\mathbb{S}_{\cup}$ such that the update transitions of any state with $All = \mathsf{F}$ will be as before, but all outgoing transitions of a state with $All = \mathsf{T}$ will be self-loops. This makes every state with $All = \mathsf{T}$ be a free termination state. Since in every fair and complete run a node will receive all input data, we are always guaranteed that we will reach a free termination state and hence can do the ready transition correctly. Using the all() relation, machines are able to determine that they have heard from each other machine in the network and make this transition.

Another common form of metadata in tranducer networks is partitioning policy metadata [9, 118]. We now show how this form of metadata can be modeled with free termination as well.

In the policy-aware setting, we are equipped with a *distribution policy* $P$ that maps each fact $t$ to a subset of the network nodes (which are the nodes that hold $t$). Each node can locally apply $P$ to any (possible) fact that uses constants from the current local active domain of the node.

To do this, we will define a new semiautomaton $\mathbb{S}^{\pm}_{\cup}$. Each state of the semiautomaton consists of a tuple of instances $(I^+, I^-)$ with the property that $I^+ \cap I^- = \emptyset$, i.e., the two instances have no tuples in common. Intuitively, $I^+$ tracks the presence of tuples, and $I^-$ tracks the absence of tuples. A transition in $\mathbb{S}^{\pm}_{\cup}$ simply adds a tuple in either $I^+$ or $I^-$. The query result is computed by running the query on the positive instance, $q(I^+)$.

The distributed computational model is similar to the one presented in Section 3.5 with the following modifications. First, in a configuration $\gamma$ the buffer $buf$ is a tuple $(buf^+, buf^-)$ which tracks tuples that are to be "added" and tuples that are to be "removed". Second, the transitions are modified as follows:

**Produce Transition:** A node can move any tuple in its local state to the buffer $buf^+$ of its neighboring node; it can also check whether for a potential tuple $t$ (from the current local active domain) $P(t)$ contains that neighboring node and if not it can add $t$ to the buffer $buf^-$ of its neighboring node.

**Consume Transition:** A node can update its state by removing a tuple from $buf^+$ and adding it to $I^+$, or removing a tuple from $buf^-$ and adding it to $I^-$.

**Ready Transition:** A node can set $ready(v) \leftarrow \mathsf{T}$.

A run is *fair* if every fact in every buffer is eventually taken out, and it is *complete* if every tuple in a state is eventually sent to its neighbors (and also every negative fact in a local policy is also sent to its neighbors). Correctness is defined in the same way as before.

For an instance $I$, we define $\bar{I}$ to be the set of tuples with values in the active domain, $\mathrm{adom}(I)$, that are not in $I$.

**Theorem 29.** *$(Q, I)$ is coordination-free correct if and only if $(I, \bar{I})$ is a free termination state for $Q$ in the semiautomaton $\mathbb{S}^{\pm}_{\cup}$.*

*Proof.* Consider the following algorithm: it will set $ready(v)$ to true exactly when $(I^+, I^-)$ is a free termination state for $Q$ in $\mathbb{S}^{\pm}_{\cup}$. This algorithm is correct, since at the instance $(J^+, J^-)$ where $ready$ becomes true, all reachable states maintain the result of $Q$. Note that from the network flooding construction, $J^+ \subseteq I$ and $J^- \subseteq \bar{I}$. Hence, $(I, \bar{I})$ is reachable from $(J^+, J^-)$ and $Q(J^+) = Q(I)$. Further, every fair and complete run reaches the quiescence point, when $state(v) = (I, \bar{I})$, and thus the algorithm will set $ready$ to true.

In the other direction, suppose $(Q, I)$ is coordination-free correct and consider an algorithm that computes the output correctly. At the quiescence point, when $state(v) = (I, \bar{I})$, this algorithm must set $ready$ to true (since the state will remain unchanged from that point on). But now consider a fair and complete run for another input $I' \supseteq I$ such that $I'$ does not contain any tuples from $\bar{I}$, and assume that some node $v$ receives first all of $I, \bar{I}$. Note that $(I', \bar{I}')$ is a reachable state in

$\mathbb{S}^{\pm}_{\cup}$. At this point, the algorithm would need to do the ready transition. But because of correctness, it must be that $Q(I) = Q(I')$. Hence, $I$ is a free termination state. □

We say that a query $Q$ is *domain-distinct-monotone* if $Q(I) \subseteq Q(I \cup J)$ for all instances $I, J$ for which $J$ is domain distinct from $I$ (meaning every fact in $J$ has some constant that does not appear in $I$). We can now show the following analogous theorem, which captures the characterization of policy-aware transducers in [9].

**Theorem 30.** *A Boolean query $Q$ is positively (resp. negatively) coordination-free in the policy-aware setting if and only if $Q$ is domain-distinct-monotone (resp. domain-distinct-monotone).*

*Proof.* From Theorem 29, $Q$ is positively coordination-free if and only if every state $(I, \bar{I})$ with $Q(I) = \mathsf{T}$ is a free termination state. Take $J$ domain distinct from $I$ and suppose $Q(I)$ is true. Then, $I$ is a free termination state. Since $(I \cup J, I \bar{\cup} J)$ is reachable from $(I, \bar{I})$, $Q(I \cup J) = Q(I) = \mathsf{T}$. Hence, $Q$ is domain-distinct-monotone. The antitone case is symmetric. □

## 3.6 Free Termination with Finite States

For practical applications, we are interested in computing free termination states both statically at compile time and dynamically at runtime. Thus far, we have identified algebraic properties of programs that allow us to detect free termination states. In this section, we study free termination when the semiautomaton has a finite state space.

### 3.6.1 Detecting Free Termination

At runtime, it is desirable to determine whether the current state is a free termination state. This can be done by computing the reachable states from the current state in the transition graph $G[\mathbb{S}]$ and verifying that they all return the same query result. Hence, this computation is linear to the size of the transition graph. However, it turns out that we can compute all free terminating states in linear time as well.

**Proposition 31.** *Let $\mathbb{S} = (D, L, U)$ be a semiautomaton with a finite state space, and $Q$ be a query. We can determine all free terminating states in time linear to the size of the transition graph (assuming computing $Q$ takes constant time).*

*Proof.* To achieve this result, we first need the following observation on the behavior of free terminating states: if $s$ is a free terminating state, then all states in the Strongly Connected Component (SCC) of $s$ in $G[\mathbb{S}]$ are also free terminating. Thus if two states $s, s'$ are in the same SCC and $Q(s) \neq Q(s')$, then none of the states in the SCC are free terminating.

The first step of the algorithm is to convert $G[\mathbb{S}]$ to a Directed Acyclic Graph (DAG) $G'$, where each node represents an SCC. This is standard and can be done in linear time. As a second step, we iterate over all SCCs and label them as a candidate if $Q$ is the same across all states in that

SCC; otherwise, we remove from $G'$ the SCC and all other nodes that can reach it. This step can also be implemented in linear time.

We are now left with a DAG $G''$ where each SCC has the same value for $Q$. In our final step, we perform a traversal of the SCCs in reverse topological order. Any SCC with no outgoing edge in $G''$ is free terminating (meaning all the states in the SCC are free terminating). At any step, if an SCC $C$ has outgoing edges only to free terminating SCCs and $Q$ is the same for $C$ and its outgoing SCCs, we mark $C$ as free terminating; otherwise, it is not free terminating. This final step also requires linear time. □

Recall that a finite semiautomaton $\mathbb{S}$ is a DFA without a start or accept states. If we denote a state of $\mathbb{S}$ to be the start state, and take $Q$ to be a Boolean query that returns true for accept states and false for the other states, we have exactly a DFA. In this case, free termination of a state means essentially that we can stop the computation of the DFA without the need to read any more symbols from the input. From Proposition 31, we obtain as a corollary:

**Corollary 32.** *The free termination states of a deterministic finite automata (DFA) can be computed in linear time in the size of the DFA.*

DFAs are an interesting example of converting an infinite state space (all strings over the alphabet) to a finite state space. We now explore connections between free termination and this notion of equivalent state representations.

### 3.6.2 State Minimization

In this section, we ask whether given a semiautomaton $\mathbb{S}$ and a query $Q$ we can construct another "simpler" semiautomaton $\mathbb{S}'$ that has the same behavior as $\mathbb{S}$ for the given query. Formally:

**Definition 33** (Equivalence)**.** *Let $\mathbb{S}, \mathbb{S}'$ be two semiautomata that both contain a start state $\perp$ that reaches all states and have the same label set $L$. Let $Q, Q'$ be two queries on $\mathbb{S}, \mathbb{S}'$ respectively. We say that $(\mathbb{S}, Q)$ is* equivalent *to $(\mathbb{S}', Q')$ if given the same sequence of transitions starting from $\perp$, the query result will be identical.*

If $(\mathbb{S}, Q)$ corresponds to a DFA, the above definition captures exactly DFA equivalence. In this case, one can simply perform state minimization in DFAs [104] to obtain a minimal DFA. The following property holds for free termination states in DFAs.

**Proposition 34.** *A free termination state cannot be part of a cycle in a minimal DFA.*

*Proof.* Assume for the sake of contradiction that we have a minimal DFA and a free termination state $s$ that participates in a cycle in that DFA. Each state in the cycle is reachable from $s$, so they must all return the same query result as $s$ (accept/reject) and also be free-termination states. The DFA in which each state in the cycle is collapsed into a single state is equivalent and has strictly fewer states, thus the original DFA cannot have been minimal. □

For a general pair $(\mathbb{S}, Q)$, we can follow the same idea as state minimization in DFAs. We define the *collapsing* of a set of states $S \subseteq D$ that take the same value for $Q$ as the modification of the semiautomaton that $(i)$ replaces $D$ with $(D \setminus S) \cup \{s_0\}$, where $s_0$ is a new state, and $(ii)$ any update that transitions to a state in $S$ goes to $s_0$, and any update that transitions from a state in $S$ starts in $s_0$. We also set $Q(s_0)$ to be the value of a state in $S$.

**Proposition 35.** *Let $\mathbb{S} = (D, L, U)$ be a semiautomaton and $Q$ be a query. Let $s \in D$ be a free terminating state. Then, the pair $(\mathbb{S}', Q')$ resulting from collapsing $U^\infty(s)$ is equivalent to $(\mathbb{S}, Q)$.*

*Proof.* Because $s$ is a free termination state, we know every state in $U^\infty(s)$ returns the same query result as $Q(s)$. We can collapse all states in $U^\infty(s)$ with all transitions being self-loops. The collapsed state $s_0$ is a free termination state and cannot be part of a cycle. Any sequence of updates that ends up in $U^\infty(s)$ in $\mathbb{S}$ will end up in $s_0$ in $\mathbb{S}$, and hence we have the desired equivalence. □

Given $(\mathbb{S}, Q)$ with finite state space, consider now the equivalent pair $(\mathbb{S}_\square, Q_\square)$ obtained by repeatedly applying the above proposition to free termination states until there is no more change; we call $(\mathbb{S}_\square, Q_\square)$ collapsed. This is analogous to a minimal state representation. Note that we need finiteness to guarantee that the process of collapsing states will terminate at some point. We can now show the following characterization of free termination in this case.

**Proposition 36.** *Let $\mathbb{S}$ be a semiautomaton with an initial state $\perp$ and finite state space, and $Q$ be a query. Then, for the equivalent collapsed pair $(\mathbb{S}_\square, Q_\square)$, a state is a free termination state if and only if all outgoing transitions are self-loops (i.e. the state is a fixpoint).*

*Proof.* From Proposition 35, we know that a free termination state that has any reachable states other than itself can be collapsed into a single state in which there are no outgoing edges except self-loops. The other direction is also clear as a state $s$ that has no outgoing transitions satisfies the definition of a free termination state as $U^\infty(s) = \{s\}$. □

State minimization offers an interesting perspective on distributed systems techniques like CRDTs. While CRDTs appear to only describe inflationary state mutations, it is common to convert non-inflationary updates into inflationary ones using metadata. We see this in both the two-phase set CRDT and the positive-negative counter CRDT (appendix 3.7). While the state transition graphs for these CRDTs are acyclic, they have equivalent representations that are cyclic. By looking at these structures from the perspective of user-visible state (query-layer equivalence), the separation between free termination of query results and eventual consistency of state becomes clear.

## 3.7 Free Termination in CRDTs

An append-only database is an example of a **grow-only set**. All results about free termination, threshold queries, and free termination of monotone and antitone queries apply to grow-only set CRDTs.

The **two-phase set** models something close to a database instance with insertions and deletions, but the semantics of deletion are slightly altered. The set of deletions is grow-only, so when an element is deleted once it is forever in the deletion set. This means that $m \in \mathsf{DELETES} \rightarrow m \notin \mathsf{INSERTS} - \mathsf{DELETES}$. In other words, we can negatively partially freely terminate on any value in DELETES. On the other hand, we can never positively partially free terminate as any element in the $\{INSERTS\}$ set could always be added to the DELETES set by some future update. In the paper Keep CALM and CRDT On [69] this example was pointed out as a CRDT that has eventually consistent state, but a non-monotone user-observable query over that state. The paper stated the intuition that this CRDT offers weaker guarantees than a threshold-query over a CRDT would offer. We now see that intuition of weaker guarantees formalized by our definition of free termination. The grow-only set CRDT offers freely terminating threshold and dual-threshold queries. While the state of the two-phase set grows monotonically over time, its non-monotone query can only offer negative partial free termination.

Observe that the state transition graph of the Two-Phase Set CRDT is inflationary with respect to the set subset partial ordering, but if we look at an equivalent state transition graph with respect to the query, the graph contains cycles. The graph however is not fully invertible, which is what allows for negative partial free termination.

The **grow-only counter** allows for eventually consistent counters for things like tallying votes or counting likes in a distributed setting. It is not sufficient to have the states be just a natural number that gets incremented because there is not an idempotent way to merge these states together, which may result in double-counting increments. To resolve this, each replica is assigned a unique ID and the only that replica will be able to apply updates that increment that unique ID's value. Effectively, the state of every node in the system is tracked in a map at each node and the merge operation overwrites this map with the more up-to-date values for any elements of the map your replica has out-of-date information on. $\mathsf{max} : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ is an associative, commutative, and idempotent operation so it forms a valid merge operation for a state-based CRDT. The update operation is monotone w.r.t. the ordering of max, which means that it can act as a proxy for determining which states are *newer* and which are outdated.

The **positive-negative-counter** extends the grow-only counter to support decrementing counters in addition to incrementing them. The challenge beyond the lack of idempotence from the grow-only counter is that the obvious update operation is no longer inflationary. To circumvent this, states are represented as monotonically growing pairs of natural numbers, with one representing the number of increments at that replica and the other representing the number of decrements at that replica. Observe that the range of the query is $\mathbb{Z}$ and all integer query results are reachable from all states in the semiautomaton. This means there is a fully invertible equivalent semiautomaton to the default CRDT representation, but the additional metadata kept around in the local states of the CRDT make the original semiautomaton fully acyclic and inflationary.

## 3.8 Conclusion

We have presented a general state transition framework for reasoning about coordination-free computation in distributed systems. Our central notion, *free termination*, allows the relational transducer approach for declarative networking and the algebraic lattices approach of conflict-free replicated data types to be modeled in a single framework.

# Part II

# Algebraic Upgrades

# Chapter 4

# Once Upon a Tree: Distributed Idempotence in O(1) Space

A critical challenge in modern geodistributed systems is to allow multiple readers and writers to share replicated state in a coordination-free fashion. Data structures like CRDTs offer solutions for types with naturally idempotent operations, such as sets. But many applications rely on non-idempotent types, including fundamental types like counters and multisets. For these types, additional coordination-free mechanisms are needed to enforce idempotence. Prior schemes require space linear in the number of replicas, which is impractical for widespread use cases like social networks and recommender systems.

In this chapter, we present OnceTree, a replication protocol that enables coordination-free idempotence enforcement in constant memory. OnceTree combines semi-lattice merges with distributed aggregation trees and controlled network flooding to enable the replication of any data type with an associative and commutative aggregation function. We evaluate OnceTree against existing replicated data types on cloud infrastructure and validate scalability trends for memory usage and propagation latency.

## 4.1 Introduction

In the past twenty years, the web has driven requirements for geo-distributed applications and always-on availability. We have seen developers bend over backwards to work around the speed-of-light limitations of geo-distributed latency and the fundamental impossibility of simultaneous availability and consistency a la the CAP Theorem [21]. Eventual consistency has emerged as an answer for a wide range of applications.

Many distributed systems techniques achieve replica consistency via monotonic or inflationary constructs, which guarantee coordination-free availability and consistency. This includes data type libraries (CRDTs [103]) and language features [7, 8, 31, 68, 81, 83]. Consistency in these contexts derives from the ability to merge updates from multiple nodes in a manner that is "ACI": Associative (insensitive to message batching), Commutative (insensitive to message ordering) and Idempotent (insensitive to repeated message arrival) ie a semi-lattice.

The requirement for idempotence is both a bit surprising and burdensome. Many operations we want to perform on our data are insensitive to batching and ordering (AC), but are **not idempotent** (I). A standard example is a "counter" or "tally". If we try to sum up tallies in a distributed fashion and we are not careful about duplicates, we can easily count things more than once and get incorrect results.

As we discuss in Section 4.2, there are various mechanisms in the literature to ensure exactly-once delivery that can be adapted to this context. Classical solutions require overly-strong consistency, while more recent solutions based on CRDTs still have space requirements that are $O(n)$ for a system of $n$ nodes. For many large systems, this $O(n)$ overhead results in storage and operational maintenance costs that get in the way of many lightweight features like "thumbs up" counters, which can enrich online experiences and drive statistical algorithms for content delivery.

In this Chapter, we introduce **OnceTree**, a protocol for replicating data types with non-idempotent operations that reduces this memory cost to **constant space**. Our key observation is that idempotence can be enforced hierarchically and therefore the metadata overhead can be reduced from tracking every node in the system to just tracking your neighbors in the network topology.

Given this, we turn to distributed data aggregation topologies. We observe that spanning trees—as used in distributed aggregation protocols like TAG [79]—provide a space-optimal topology for idempotence enforcement. Where aggregation techniques like TAG focus on returning results to a single root node, we are interested in replicating results to all nodes, such that every node provides strong eventual consistency. We address this challenge by combining a network flooding algorithm with the aggregation tree architecture and a hierarchical semi-lattice operation. The OnceTree protocol offers strong eventual consistency of state at every node in the system and robustness to duplicate or reordered messages over the network.

To evaluate the performance of the OnceTree protocol, we benchmark our implementation of a social media counter against the PN-counter from CRDT literature. To see the trends of our performance in a data center setting, we run OnceTree and the PN-Counter baseline on a public cloud cluster. Our experimental results focus on the scaling behavior of our protocol as

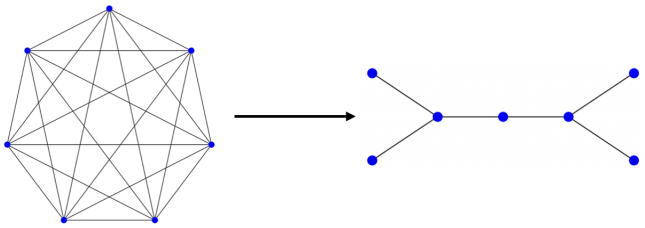


Figure 4.1: Instead of randomized broadcasts (left), we apply a tree topology (right), which enables constant memory usage.

we vary the cluster size. We demonstrate that OnceTree uses **constant** memory per-node as the cluster size increases by orders of magnitude, and that the propagation latency of updates scales according to the **log** of the number of replicas. These results validate our theoretical claims that OnceTree is effective as a constant-memory alternative to CRDTs.

In summary, we make the following contributions:

- We introduce the concept of hierarchical idempotence for constant space idempotence enforcement in distributed systems.
- We give a full protocol called OnceTree for combining hierarchical idempotence and strong eventual consistency and we prove the correctness of the protocol.
- We identify connections between three apparently disparate areas of data systems: coordination free update processing, aggregation queries, and network flooding.
- We give a localized tree reorganization protocol allowing nodes to join and leave the tree over time.
- We evaluate the memory usage, propagation latency, and operation throughput compared to existing CRDT solutions in cloud deployments atop the Hydro dataflow engine.

## 4.2 Background

The idempotence of requests has been a concern in databases and business processing for decades, though more typically discussed in industry than in academia. Both Gray and Reuter [45] and Bernstein and Newcomer [18] cover the issue in their transaction textbooks, often as part of real-world consequences outside of transaction commit—e.g., “drill hole” (idempotent) vs “dispense money from ATM” (non-idempotent) [45, p. 544]. In the 20th Century, at-most-once execution of non-idempotent actions was the purview of industrial products like Transaction Processing Monitors and Reliable Message Queues, which ran either as standalone servers or on top of transactional database backends.

The advent of e-commerce on the web brought these issues to the attention of a wider range of developers (e.g. [84]). As financial applications grew popular, even consumers became concerned about real-world consequences of non-idempotence. What happens if you reload the web page right after pressing the button to buy 100 shares of stock in Pets.com? In the 21st century, the majority of developers need to understand that code in general is neither idempotent nor irrelevant to real-world concerns. A general mechanism is required to enforce at-most-once execution—i.e. developers need reusable techniques to "upgrade" non-idempotent operations to idempotent ones.

The standard solution for at-most-once execution requires persistent state, and falls squarely in the tradition of database transactions mentioned above. Some agent (say, a server) must be able to identify and remember every request it receives, so that it can later identify identical requests that are to be ignored. Networking protocols like TCP will do this at a "network session" level, but they abdicate on end-to-end application guarantees: TCP sessions can fail for many reasons and erase accumulated state. Instead, developers typically store some "session state" in server infrastructure—either a transactional database, or a middleware service that acts as or uses a database. In a contemporary setting, that middleware might be a Kafka broker, or more typically a fault-tolerant cluster of such brokers managed by a Zookeeper consensus service.

### 4.2.1 Uncoordinated "Like" Gossip

Let's work with the example of implementing status variables for social media posts, with $n$ servers placed around the world. The traditional solutions above are unpleasantly heavyweight: all button-presses would need to round-trip through a strongly-consistent storage system. This has obvious overheads in resource consumption and cost. Worse, strongly consistent systems are subject to unavailability—victims of the CAP Theorem [21]—and are generally to be avoided in high-scale, eventually-consistent settings like social media.

To bypass CAP without sacrificing correctness, the Generalized CALM Agenda [52] suggests we look for a monotonic or inflationary implementation of idempotence that is eventually consistent in the face of failure and recovery. The CRDT literature provides solutions [94, 103] that we discuss next, but as we will see they still have substantial storage overheads.

To illustrate, let's focus on a generic status variable for one particular social media post. Each of our $n$ servers can respond to requests to update the variable or display it. Each can also receive "gossip" updates from the other servers, and merge them into the current status. Servers periodically gossip their status, either to randomly-chosen servers or exhaustively via broadcast.

Suppose the status variable is a Boolean, e.g., "flagged for inappropriate content", and the gossip merge function is logical OR: if anybody flags the post, it remains flagged. OR is an idempotent function, so the merge process is eventually consistent under any pattern of gossip including retries.

By contrast, let's return to our counter of "likes". It is tempting use integer addition (+) as our merge function, but of course then gossip can cause us to over-count: we do not know if the count coming in already incorporated our local count that we gossiped earlier! This is likely to cause us to mis-report the count, and lead to long-term replica divergence.

How can we avoid this? One trick that is used (in places ranging from TCP to CRDTs) is to include a *sequence number* with each message. Each node maintains a monotonically increasing sequence number or "clock" that it timestamps on its messages. Nodes also keep track of the clock values they've heard from each other node to form a *vector clock* of size $n$. To enforce at-most-once delivery, any duplicated messages from $i$ can be detected via their timestamp and dropped[1]. We call this the *Vector Clock Idempotence Trick*. But this is not a perfect solution either. Each vector clock requires $O(n)$ storage per node. The memory overhead is $O(n^2)$ in total across the $n$ nodes in the system.

There is another trick used for enabling idempotence; it is "state-based" rather than "operation-based", to use the language of CRDTs. This trick we will call the *Node ID Map Trick*. Instead of gossiping operation requests (like "increment"), we maintain and gossip a hash table of the status variable *values* for each server. The table maps from a key to a value pair: `serverId => (clock, value)`. Upon sending gossip, a server increments its `clock`, copies its entry `myID => (clock, value)` into the map, and gossips the entire map. Upon receiving gossip, a server merges the gossip into its local map on key-by-key basis, replacing the current entry for a key if the gossiped entry has a higher `clock`. It should be clear that this merge function is idempotent: if we receive the same map entry more than once, the pair of `serverID` and `clock` entry allow us to recognize and ignore duplicates. To display the overall count, we sum all the counter values in the map. Because each value in this map has only a single writer, this replication scheme is eventually consistent. The memory cost? Still $O(n)$ per node, resulting in $O(n^2)$ across the entire system.

### 4.2.2 Once Upon a Tree

In this paper, we present the OnceTree protocol, which aims for consistency, coordination-freeness *and* space efficiency. Like the CRDT approach, our solution is both strongly eventually consistent and coordination-free. Unlike the CRDT approach, our goal is *constant* storage overhead per node – $O(n)$ across the the network of $n$ servers. OnceTree solves this challenge by building on two key bodies of work: spanning tree flooding from the networking literature, and distributed aggregation queries from the database literature.

Although our counter example is not idempotent, it is **associative and commutative**. The OnceTree protocol takes advantage of these properties to enforce idempotence in constant space. Under the hood, OnceTree works by organizing replicas into a tree topology, such as the one on the right of Figure 4.1, rather than the all-to-all topologies used by replication protocols like CRDT gossip. Many core modern systems are designed with architectures that satisfy these constraints, and therefore can use OnceTree to replicate a wider range of data types while preserving low memory usage.

For example, much academic work uses the set rather than multiset ("bag") semantics of databases, but in practice SQL databases require duplicate semantics to be preserved. In the multiset setting, we cannot depend on the idempotence of tuple insertion. For a decentralized

[1] In addition, messages from a node $i$ that arrive out-of-order by $i$'s timestamp can be buffered by the receiver to emulate in-order delivery, as is done in TCP, and duplicates can be suppressed in the buffer as well.

eventually consistent database such as the Anna KVS [110, 111], supporting multiset semantics via existing CRDT or semi-lattice solutions would bottleneck the scalability of an "any scale" system on the linear memory usage. With OnceTree, it is possible to preserve the scalability of systems like Anna while supporting the bag semantics needed to implement a SQL database.

OnceTree also has many uses outside classic database systems. Distributed training for machine learning is a popular research topic [74, 89, 108] motivated by data privacy as well as horizontal scalability. Many solutions update models by combining their weights with other models or with gradients via summing or averaging. Both operations are associative and commutative, but not idempotent. OnceTree offers an idempotent and coordination-free solution for these applications like federated learning [57].

As a final example, consider weighted graphs. When representing distances between nodes such as a network graph or a graph of physical distances like world maps, updates to the weights of these graphs are not idempotent. But adjusting weights *is* associative and commutative. Furthermore, many applications on top of weighted graphs are naturally decentralized such as network routers updating their distance estimates. In such applications, OnceTree makes coordination-free replication **practical**. With OnceTree, the "any scale" promise of coordination-free systems can be extended to applications that only satisfy the A and C of ACI.

## 4.3 The OnceTree Protocol

In this section, we describe the OnceTree protocol we use to achieve constant space idempotence enforcement and strong eventual consistency. We will prove its correctness in Section 4.4.

### 4.3.1 The High Level

We assume the following system model: We have a distributed system made up of nodes that each have unique IDs. The nodes have mailboxes at which they receive messages and they are able to pull messages out of the mailboxes to process them. The nodes can send messages to each other's mailboxes over a network, which is best-effort with retries: messages can be delivered out of order or multiple times but all messages will eventually be delivered to the correct destination if it is live. Messages will not be corrupted or partially delivered. We assume the nodes have been organized into a logical tree overlay which can be accomplished with a standard network spanning tree algorithm [92].

Our system is *uniform* meaning each node runs the exact same code which is the code in Algorithm 1. The flow of communication follows a standard controlled flooding protocol for spanning trees in computer networking. The algorithm in simple English is "when I receive a message from my neighbor, I forward it to all of my neighbors except the sender." This guarantees messages eventually reach every node and that messages follow exactly one path between any (source, destination) pair.

On top of this controlled flooding, we combine two other tricks in our protocol. The first is the NodeID Map Trick in which we track the largest clock value we have ever heard from

each neighbor in a [neighborID => (clock, value)] map. This is where our robustness to out-of-order and duplicate deliveries comes from: it is guaranteed by the idempotent and commutative properties of the max() computation implemented in Algorithm 1. However, this introduces a new challenge: because we are only storing entries for each *neighbor* rather than every node in the cluster, an implementation with just this trick would not have sufficient information to compute queries representing operations from across the cluster.

The second trick is the use of distributed data aggregation. Instead of receiving a message from neighbor $i$ and forwarding it to every other neighbor as in a standard network, we aggregate the new message along with all our local values and forward the aggregation result to each of our neighbors. This keeps the size of state being passed around bounded by the size of the aggregate results, which are constant.

Observe that doing this naively would result in the last message from a neighbor $j$ being sent back to $j$ as part of this combined aggregation, breaking the exactly-one-path property we want from our tree with controlled flooding. To solve this cyclic problem, we must send each neighbor $j$ the aggregation of all our neighbors *except* the state we have heard from $j$. Achieving this requires each node to maintain aggregation state for each of its neighbors.

In summary, we receive a message from a neighbor $i$ in the form of an aggregation of their state. We "merge" it into our map of neighbor IDs to neighbor aggregates using the max() semi-lattice merge operation. If the merge changes *a value* in our neighborID map then for each neighbor $j$ in our neighborID map except the sender $i$, we will send $j$ the aggregation of all values in the map where neighborID $k \neq j$, i.e. the aggregation result with node $j$ excluded.

### 4.3.2 State and Gossip

Figure 4.2 depicts the local state of a OnceTree node that has three neighbors (N1, N2, and N3). Each node can be written to by clients, modifying its *local* counter value. Local updates are in the form of an "update" operation which in our like counter example could be +1 for like or -1 for dislike. Each node also tracks one aggregated counter for each of its neighbors. Together this local state constitutes a {NodeID:(clock,value)} map of a node's local update state as well as the state of each of its direct neighbors.

When any value in our local map is modified, we compute the aggregate of our local state to send along to each of our neighbors, excluding our map element for the node we are sending the aggregate to. In the figure, this is shown for a message arriving from neighbor 3. The red arrows show the aggregates which are computed to send to the other two neighbors upon receiving this message, but no aggregate is sent back to neighbor 3. Note that the aggregates sent to neighbors 1 and 2 do not reflect the local state from their own mailboxes.

The aggregate function in this example, for the purposes of simple explanation, is the same function used to compute updates. In our running example, this function is SUM as the update is an increment or decrement and the aggregate is the SUM of these counts. Some applications may use different functions for update and aggregate, or may want to materialize multiple aggregates. As long as the update and the aggregate operations are both associative and commutative, all of our correctness guarantees still hold.

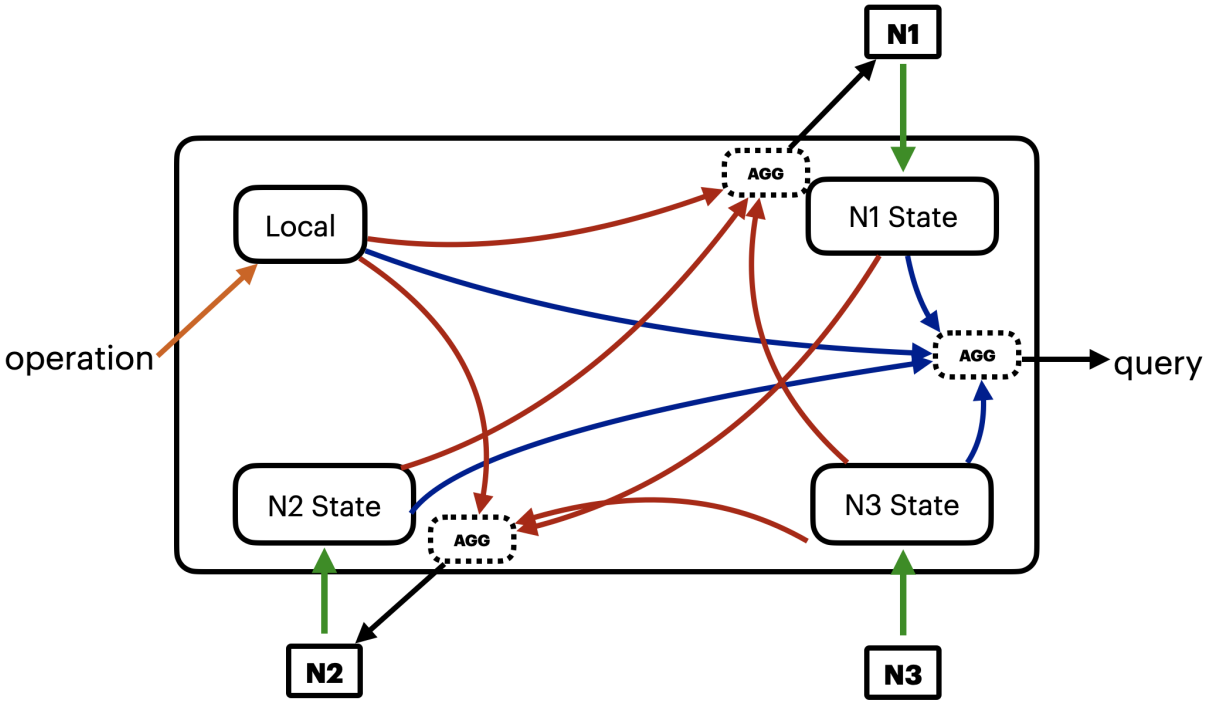



Figure 4.2: The dataflow of a OnceTree node: blue arrows contribute to query results, green arrows are messages from peers, and red arrows contribute to peer aggregates.

Figure 4.3 depicts the flow of data through our tree for the example of a single increment to a single node. We see that the update first propagates to the node's neighbors, then to the neighbors' neighbors, and so on. The values being sent are not "+1" in our protocol, but the full sum of local states. We use a local logical clock that increments whenever a local variable is changed in order to ensure our merge operation (max(incomingTime, currentTime)) ignores duplicate or out of order messages. One way to think of this clock is as a monotonicity enforcer for our potentially non-monotone update operation.

### 4.3.3 Aggregating Queries

The goal of the system is for each node to accept writes and to serve query results to clients without coordination. The strong eventual consistency guarantee (formalized in the next section) says that any update that a node receives locally from a client or that it hears about from another node in the tree should be immediately reflected in any query to that node. This implies that a client will read its own writes and that the query result from a single node will never "go backwards" in time.

We compute the query by simply aggregating each of the states we track locally for our NeighborID Map. Any local update is incorporated into the map immediately and any message deliv-

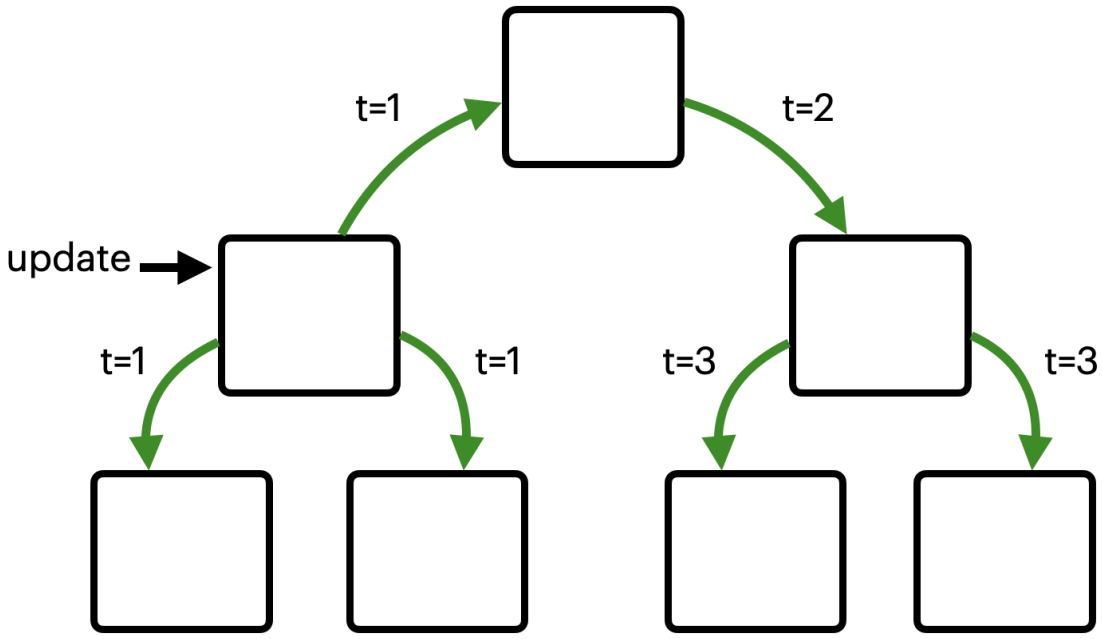



Figure 4.3: The flow of communication when an update occurs at a node; t=k denotes the kth propagation step.

ered to our mailboxes will be incorporated into the map immediately if it has not already been processed (because it is a duplicate or came out of order).

Altogether, the OnceTree protocol can be summarized in the relatively simple pseudocode of Algorithm 1. The simplicity of the protocol makes it easy to implement (indeed, our benchmark implementation is < 400 lines of idiomatic Rust), and perhaps more importantly, easy to *reason about* from the perspective of both proofs and end-user applications. We omit some optimizations from the full implementation, such as reactively gossiping aggregated values when they change rather than restricting ourselves to periodic gossip, but these can be layered on top of the existing protocol with minimal effort.

### 4.3.4 Constant Space

There are two core ideas at the heart of why we are able to achieve constant space. The first is the observation that to enforce idempotence without a giant log of past message IDs, you appear to need to track the state of every node in the system. Our hierarchical idempotence idea leans on the observation that if your updates only arrive at another node via a single network path then you only need to track the state of your neighbors to enforce idempotence. This reduces the

```
crdt OnceTree-Replica
    state_local, state_{i∈(0..|neighbors|)} ← (0, ⊥_aggregation)
    timestamp ← 0
    merge merge_i(v)
        if state_{i,clock} < v_clock then
            state_i ← v
    gossip
        timestamp ← timestamp + 1
        for i ← (0..|neighbors|) do
            agg ← state_{local,value}
            for j ← (0..|neighbors|) do
                if i ≠ j then
                    agg ← agg + state_{j,value}
            send i ← agg
    operation localUpdate(update)
        state_local ← state_local + update
    query
        agg ← state_{local,value}
        for i ← (0..|neighbors|) do
            agg ← agg + state_{i,value}
        return agg
```

**Algorithm 1:** Pseudocode for the OnceTree protocol.

number of states tracked from linear (the number of nodes) to constant (the number of neighbors).

However, there is still the question of how large the states you are tracking are. Are you going from storing O(n) integers to storing O(1) integers, or from O(n) integers to O(1) lists of integers that are O(n) long? This is where restricting our data to "aggregation" functions comes in. Any associative and commutative query would still achieve strong eventual consistency in our protocol, but the space usage will only be constant if the size of the query output is constant. Our requirement that the query be an "aggregate" function does not mean it needs to be a SQL aggregate, but simply match the formal definition of aggregate that the input domain is a set, multiset, or list and the output domain is a single element [55].

## 4.4 Convergence Guarantees

In this section, we prove the idempotence enforcement and eventual consistency guarantees that our design offers. In these proofs we assume no node failures. In section 4.6.2 we consider faults.

### 4.4.1 Assumptions

In our proofs, we making the following assumptions about the system and user-defined functions:

- We assume the nodes in our system are in a tree formed by a spanning tree protocol such as the canonical algorithm by Perlman [92]
- We assume our update/aggregate operation is associative and commutative.
- We assume the networking layer is best-effort with retries: messages can be delivered out of order or multiple times but all messages will eventually be delivered to the correct destination if it is live. Messages will not be corrupted or partially delivered.
- We assume all nodes do not exhibit byzantine behavior.

### 4.4.2 Preliminary Definitions and Proof Outline

We use the definitions from the original CRDT paper [103] for eventual consistency and strong eventual consistency. An object is (4) **eventually consistent** if

1. An update delivered at some correct replica is **eventually delivered** to all correct replicas
2. Correct replicas that have delivered the same updates eventually reach equivalent state
3. All method executions **terminate**

An object is **strongly eventually consistent** (SEC) if it is Eventually Consistent and "correct replicas that have delivered the same updates have equivalent state".

Consider the set of updates in the system. Our goal is to show that eventually all updates are delivered to all nodes and that the query result at each node returns exactly the aggregate of each update that was delivered to that node exactly once. We will prove this in six steps.

1. We will show that because of the tree topology, each update in the system is eventually delivered to each node. This satisfies condition (1) of SEC.
2. We will show that because of the tree topology each update delivered to a node will be delivered to exactly one of its mailboxes, and therefore the intersection of updates between any two mailboxes on a given node is empty. We show this by the property of the tree topology that there is at most one path from any update source to any destination node.
3. We will show that all updates eventually terminate by showing that there are no loops in our communication topology. This satisfies condition (3) of SEC.

   *With these guarantees we know that messages are not being duplicated by arriving along multiple paths. We still need to show that our local merge operation which may receive retried or out of order messages from the same sender is robustly handling these sources of nondeterminism.*

4. We will show that the local variable computed based on each respective mailbox represents exactly one occurrence of each update that has been delivered to that mailbox. We show this via the inflationary and idempotent properties of the MAX operation used to merge in messages received in the mailboxes.

5. We will show that no local variables contain duplicate updates by showing that none of the three data modifying operations in the design can introduce duplicates.

6. Lastly we will put the above results together to prove that the QueryResult returned to the user is exactly the aggregate of all updates delivered to all mailboxes of the node and each update delivered is represented exactly once in the query result. With this we will have proven conditions (2) and (4) of SEC as well as exactly once delivery in our protocol.

**Terminology:** All messages sent between nodes in this protocol are the result of computing an aggregation function. We say that a message "contains" an update if that update was one of the inputs to that aggregation function. The messages are often the result of many rounds of composition of aggregates, so this containment refers to all the leaf updates in the full aggregation tree that results in this message. Given this, for a message or a local variable to "contain duplicates" means that the aggregation tree it was computed by has more than one of the same update in its leaves.

We will refer to our aggregation function as **+** throughout this section. Recall that for simplicity of exposition we are also saying the update and aggregate functions are the same, so they are both referred to as **+**. Of course, they do not need to be the same and neither of them needs to be addition, they can be any associative and commutative aggregation functions.

### 4.4.3 Proofs

Proofs of Steps 1 through 3 all follow from the same key property of a tree that makes it useful in our protocol: trees have exactly one path between any pair of nodes.

**Step 1** is true by the fact that there exists a path between each pair of nodes.

**Step 2** is true by the fact that there is at most one path between any two nodes. For an update to arrive at a destination via two different neighbors would mean there are two paths from the update source to that destination, violating the exactly-one-path property of trees.

**Step 3** is true by the fact that trees contain no loops and that we never send an update back along the path it arrived on so it only travels "out" along the tree from where it originated.

**Step 4**: The local variables do not drop or duplicate any updates that are delivered via their respective mailboxes.

For intuition of why this is true, observe that our local state of a OnceTree node is a map of a fixed size computed via element-wise max operations. This is exactly the same semi-lattice used in the NodeID Map Trick except that our map has elements for each neighbor instead of n elements. Aside from the number of elements, the two semi-lattices are identical. Intuitively, we

use the max operation the same way in the OnceTree as it is used in standard counter semi-lattices like Grow-Only and PN-Counters. Max deduplicates and enforces monotonically growing values.

Without loss of generality we will show this for the var1 variable. Each message arriving at the mailbox of var1 grows monotonically with time (by the local inflationary logical clock). The result of our merge operation max() can only take on values of messages that arrive, so under the assumption that the arriving message does not contain duplicates, var1 will also not contain duplicates. The max() operation also guarantees that the value of var1 will be the latest value from the source that ever arrives at its destination (by the monotonicity of the logical clock once again). The source node simply accumulates updates over time, so the most recent in time includes every update that has ever been included in a message received by the mailbox of var1. QED.

**Step 5**: The local variables contain no duplicates at all.

There are three operations that modify the values of local variables: (1) Local update, (2) merging incoming messages via max(), and (3) aggregating local variables via **+**. We will show that none of these introduce duplicates and therefore local variables cannot contain duplicates.

(1) Local updates are unique and by definition are non-duplicate.

(2) max() does not introduce duplicates (in Step 4)

(3) **+** is composing some subset of the local variables. Assuming none of the composed values already contain a duplicate, **+** could only introduce a duplicate by an update being represented in more than one of the local variables being composed. Assume for the purpose of contradiction that there exists an update u that occurs in two of the local variables. There are two cases: (1) an update occurs in a local and a mailbox-based local variable or (2) an update occurs in two mailbox-based local variables.

Case (1): Let u be a local update at node D and let var1 be the mailbox-based variable at node D also containing u. Per the proof of Step 4, var1 can only take on values delivered to it via its mailbox. The origin of u is node D and the existence of u in var1 means there is a loop from D to D in our communication topology. This violates Step 3 that there are no loops.

Case (2): Let u be an update represented in var1 and var2 of a node D without loss of generality. By the proof of Step 4 we know that var1 and var2 can only take on values that are delivered to their respective mailboxes. This violates Step 2 that an update cannot be delivered to more than one mailbox of D because it again breaks the exactly-one-path guarantee of the tree topology with controlled flooding broadcast. QED.

**Step 6:** $query()$ = **+**(all delivered updates) and contains no duplicate updates. i.e. is "correct", strongly eventually consistent, and all updates are successfully deduplicated.

We already know from Step 4 and Step 5 that each of the local variables contains no duplicates and contains all updates delivered to its respective mailbox. Recall the definition of the OnceTree query from our pseudocode:

$query()$ = **+**(local, $neighbor_1$: value, ..., $neighbor_k$: value)

All updates represented in each of these local variables are represented in the query output which is all updates delivered to the node. It only remains to show that no duplicates are introduced by combining the respective local variables into the query result. This is true by part (3)

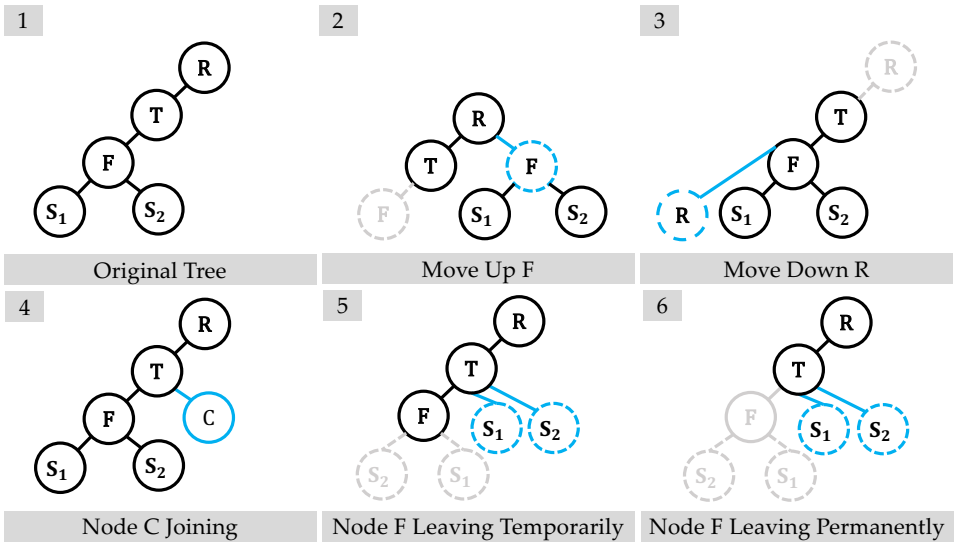



Figure 4.4: The five move operations described in Section 5.2. Each node post-move tracks its local state as well as the aggregates of each neighbor post-move. Aggregates of pre-move neighbors are forgotten.

of Step 5 that aggregation will not introduce duplicates if the local variables input contain no duplicates. QED.

## 4.5 Initialization and Reorganization

Thus far we have described a protocol that achieves constant space usage, idempotence, and strong eventual consistency by organizing nodes into a static tree topology. In this section, we show how OnceTree preserves these properties during reconfiguration of dynamic trees. A static topology implies a static membership, which is too restrictive for most deployments. We describe how to maintain our invariants in settings where nodes join and leave gracefully, deferring fault tolerance to Section 4.6.

We also consider how to safely modify the topology for a fixed membership. Tree topologies lack redundancy; this provides efficiency in a stable network, but makes the topology very sensitive to the performance and availiability of individual nodes. Particularly as nodes join and leave, a system may need to adjust the topology to ensure the tree remains connected and its performance objectives are met. We extend the protocol with a *move* operator that preserves all the invariants of OnceTree as nodes change their position in the tree.

### 4.5.1 Tree Initialization

There are a lot of scenarios that could use a OnceTree. These include a fixed set of machines in a datacenter, elastic clusters that grow and shrink with the workload, or even peer-to-peer settings that are common in CRDT literature like local-first software [63]. OnceTree is a general protocol, so we do not take an opinionated stance on the kinds of systems people may want to run on top of it. Instead, we outline different ways the tree should be initialized for different types of applications.

For a fixed-size backend cluster within one geographic region, we initialize a OnceTree by passing a list of all participating node addresses to a setup node (or nodes) that will organize them into a balanced tree of configurable fanout. For highly geo-distributed applications where nodes may have quite different distances between them, we can instead pass in the distance matrix and the setup node can compute a minimal spanning tree (MST) for the set of nodes using a classical algorithm [37, 47, 56]. After choosing a balanced tree or MST, the setup node will notify each actual OnceTree node of the addresses of its neighbors in the tree, and they can immediately start propagating any updates from their clients throughout the tree. Note that computation of the spanning tree can be scaled up over a larger set of setup nodes if the address list is prohibitively long [36].

For an auto-scaling backend cluster, if the workload spikes then individual nodes can add new nodes to the tree by spawning a child node to hand off part of its state or some of its clients to. This is a common technique in actor systems utilized, for example, in the coordination-free Anna key value store [111]. Similarly, a node can remove itself from the tree when the load decreases. For a peer-to-peer style setup, nodes joining may correspond to being invited to participate by a node already in the system, in which case the invited node can join as a child of the inviter in the tree. If a peer plans to leave the system cleanly, it notifies its neighbors who will run our node-leaving protocol described in the next section.

### 4.5.2 Planned Modifications to the Topology

We give five movement operations for OnceTree. They are depicted visually in Figure 4.4 and described in text below. Each preserves the strong eventual consistency guarantee of our protocol, which gives valuable guarantees to sticky clients as nodes move around in the tree without clients noticing. Each of our move operations preserves strong eventual consistency and exactly once delivery of updates as well as allowing clients to continue reading and writing throughout the move operation.

**Nodes Joining:** The simplest case of changing the tree topology is allowing a node to join the system. Nodes always join as new leaves in the tree. We assume the new node is in contact with some existing node in the tree [111]. Panel 4 of Figure 4 shows a node C joining a tree at parent T. To complete the join protocol, C initializes its clock to 0, adds the latest state of T to its neighbor list, and is up and running. T adds C to its neighbor list and immediately begins receiving updates from C and propagating them through the tree.

**Moving a Node One Level in the Tree** The protocol in Algorithm 2 shows the steps to move a node S (either S1 or S2) from being a neighbor of node F to becoming a neighbor of a new destination node T. This can be used to move a node one level up the tree or one level down the tree. The results of move-up and move-down are depicted in panels 2 and 3 of Figure 4. For simplicity of exposition, we will describe the scenario as a child node C moving up from its parent node P to become a child of its grandparent node G.

*At node C:* **atomic**
{
    ignore new messages from P
    send state to G
}
*At node G:* **atomic**
{
    insert C ← ($state_C$.time, +($state_C$.neighbors - $state_C$.P))
    merge P ← ($state_C$.P.time, $state_C$.P.val)
    send state to C, P (if alive)
}
*At node C:* **atomic**
{
    neighbors ← (neighbors \ {P}) ∪ {G}
    insert G ← ($state_G$.time, +($state_G$ \ {$state_G$.C}))
}
*At node P (if alive):* **atomic**
{
    neighbors ← neighbors \ {C}
    merge G ← ($state_G$.time, +($state_G$ \ {$state_G$.P}))
}

**Algorithm 2:** Pseudocode a node C moving to be a child of a node G through a node P.

In English, the protocol 1) has the moving node, C, add itself to the neighbor list of destination node G, 2) sets $state_G.P$ to $max_{time}(state_c.P, state_G.P)$ (to ensure neither go backward in time and violate SEC), and then has C add G to its neighbor list, replacing P in its list. The value of P is now included in G's aggregated state sent to C, so C should not track P anymore itself. If the protocol is run as the result of P failing then we're done. If P is still alive then the last step is to have G tell P that its newest aggregated state includes C, so P should forget about the state of C. Panels 2 and 3 of Figure 4.4 depict the tree state after a "move-up" operation and a "move-down" operation respectively. Throughout the move operation C will ignore messages from P. The essence of this protocol is the preservation of strong eventual consistency at each step by only updating values via a max() operation (our standard lattice merge). The nodes are fast-forwarding their views of P ($state_G.P$ and $state_C.P$) to whichever of them has the most recent view of P. C and G will both set their views of each others' values to the max of their two values, and G will become the owner

of the value of P. Note that G's view of G is always the most up to date, and the same is true for C's view of C, so $state_G$.C is set to $state_C$.local and $state_C$.G is set to $state_G$.local.

**Nodes Intentionally Leaving the Tree Temporarily:** This protocol is for when a node wishes to go offline but will come back online later (e.g. to keep using an app while on a flight without internet). The concern in a node going offline is that it could be on the path of communication between two other nodes in the tree. If the node is already a leaf of the tree, not an internal node, we don't need to do anything because no propagation path goes through it. If it is an internal node, we simply convert it into a leaf of the tree. We do this with the "move" operation, described above and shown in Algorithm 2. As illustrated in Panel 5 of Figure 4.4, we "move up" each child of the leaving node F, and now F is a leaf. We continue to track its clock and state at its parent just like any other leaf node, and it being offline just means its state at its parent will be stale. For the parent T to also gracefully leave the tree, an implementation must not only preserve F's state, but also sufficient routing information for F to rejoin the tree. This could be stored within the tree in peer-to-peer settings [99, 114] or in disaggregated storage in datacenter settings.

**Nodes Intentionally Leaving The Tree Permanently:** In the event that a node C plans to leave the tree permanently, we initiate the "Nodes Intentionally Leaving The Tree Temporarily" protocol for C, after which the parent P of (now-leaf) C will absorb the local state of C into its own local state and forget the state and clock of C. Specifically, P will set its local state to +($state_P$.local, $state_C$.local) and delete its aggregated value and clock for C. This movement is depicted in Panel 6 of Figure 4.4.

## 4.6 Fault Tolerance

If OnceTree nodes fail or are disconnected, the tree cannot propagate all updates. Note however that such faults do not break any node's ability to continue processing reads and writes from clients or offer SEC! Our concern is specifically with the failure of internal tree nodes, which partition the tree and prevent update propagation between live nodes.

Many "off-the-shelf" approaches to fault tolerance can be cleanly integrated with the OnceTree design. In this section, we provide three complementary points in this design space: a pessimistic approach using coordinated node replicas, an optimistic approach requiring full tree restarts, and a more nuanced coordination-free single-fault solution based on our move protocol from Section 4.5.2.

### 4.6.1 Simple Process Groups

A straightforward point in the fault tolerance design space is to "harden" each logical node in our tree so that it takes $f + 1$ physical machine failures for a tree node to fail. This can be done by implementing each OnceTree internal node using a process group of replicas: e.g. to tolerate one physical node failure we require two physical nodes per internal logical node of the tree. This design has the benefit that one can configure how many faults to tolerate independently at

each node, incurring a multiplicative factor of k * internalNodeCount space usage (still O(1) in the number of nodes). This design is relatively simple to debug and can be easily bolted on to the OnceTree protocol.

Replica consistency raises a key design tradeoff. One option is to propagate updates through replicas asynchronously. In this case we must settle for *eventual consistency* rather than *strong eventual consistency*. As a result, if a client switches from one node replica to a backup, the backup's data may be older than what the client previously saw. In the worst case a replica may fail permanently before all its updates are replicated, resulting in unbounded staleness.

Alternatively, to maintain SEC we can require updates across the process group to be synchronous: all nodes must incorporate the update before any makes it visible. Doing so, of course, departs from the fully coordination-free, asynchronous assumptions that motivated the design of OnceTree. On the other hand this may be a pragmatic hybrid in many settings: e.g. a geo-replicated system may efficiently support synchronous operations for fault tolerance within a small physical radius, while still favoring asynchronous propagation over long distances (across cloud regions).

### 4.6.2 Resetting The Tree

A second point in the design space is to reset the tree topology when faults cause the tree to become disconnected. Given the addresses of each node in a OnceTree, it is rather simple to initialize a reset of the tree topology that preserves strong eventual consistency. Simply put, each nodes maintains their local state but drops the aggregated views of its old neighbors; then a new spanning tree is constructed, and the OnceTree is allowed to run until a full round of flooding is complete. Once the flooding round has completed, the nodes can now begin reporting their new aggregated state as query results. This guarantees that the query result moves forward in time from where they were before the tree reset.

This reset approach leaves two challenges. The first is how to decide when to hit the reset button. This is application-specific and depends on what kind of tree-healing defenses are in place before resorting to this reset button.The second challenge is where to keep this list of each node in the tree. This again depends a lot on the setting. A simple approach is to maintain the set of live nodes in a document in cloud storage. The tree initialization code initiates this document. Nodes must add themselves to the list successfully before joining, and parents whose children time out must mark them dead in the list.

## 4.7 Evaluation

We have implemented OnceTree on top of the recently released open-source DFIR [50, 101] dataflow engine written in Rust, which makes it easy to define distributed dataflow through the composition of functional operators. The relative simplicity of the OnceTree protocol makes it easy to implement, totalling less than 300 lines of idiomatic Rust. PN-Counters, although ineffi-

cient memory-wise, benefit from the simplicity of the CRDT approach with an implementation in just 200 lines.

In our evaluation, we are primarily focused on evaluating the **memory utilization** and **propagation latency** when using the OnceTree protocol. Our benchmark is based on the motivating use case from the beginning of the paper: a like counter on a social media platform. This replicated state supports operations that increment or decrement the counter value, and a query that returns the current total like count. Because OnceTree is focused on scenarios involving many pieces of replicated state, we have each node in our cluster track a mapping from `uint64` post IDs to `int64` counters. By benchmarking the protocol with many counters, we can more effectively evaluate the memory efficiency of our approach.

We compare OnceTree to the PN-Counter from CRDT literature, the standard replicated data type for integers with increment and decrement operations. Our PN-Counter implementation, also written in DFIR, stores a pair of vectors which separately track the increments and decrements from each node in the cluster. When performing a query, these vectors are individually summed up and the difference is returned to the user. CRDTs use an all-to-all network topology, so we implement a broadcast approach where nodes send gossip to all their peers whenever their replica changes. Compared to approaches like randomized gossip, this gives the PN-Counter the best chance for low-latency propagation, which is its main difference over the OnceTree protocol.

In some more advanced implementations that involve additional book keeping, the bandwidth utilization of CRDTs can be reduced by passing around deltas rather than an entire copy of the state. Although this requires significantly more engineering effort, it can reduce propagation latency in scenarios with many replicas. To evaluate how OnceTree would compare to such protocols, we benchmark against a "PN-Delta" protocol that simulates the properties of an optimized CRDT by having nodes only gossiping the vector elements that correspond to their contribution to the state. Note that this mock protocol *requires* an all-to-all topology to satisfy eventual convergence, but for our latency-focused analysis this is sufficient.

We demonstrate that with OnceTree, the memory utilization at each node is constant regardless of the size of the cluster and the propagation latency scales according to $\log(n)$, where $n$ is the number of replicas in the system. This makes our protocol very practical for use in large-scale systems, where OnceTree enables the creation of many replicas with minimal impact to the staleness of queries.

### 4.7.1 Experimental Setup

We run all of our experiments on Google Cloud using n2-standard-4 VMs with 4 vCPUs, 16GB of memory, and 10Gbps egress bandwidth running Debian-11. All experiments are run in the us-west1 region. Although these machines have significantly more resources than necessary for the OnceTree, using larger machines reduces variability and allows us to compare against more expensive alternatives.

We run our experiments in a matrix style varying three properties: the replication protocol being used, the number of nodes in the cluster, and the number of concurrent increment requests. In our experiments with the OnceTree protocol, we launch a static tree topology that is a full

binary tree with a configurable depth. The number of nodes in the tree is $2^{depth} - 1$. When comparing against the PN-Counter, we instead create a cluster with the same number of nodes. We use the same set of virtual machines across each matrix configuration, to avoid variability due to node placement.

Both protocols are configured to immediately gossip their state whenever it changes to minimize latency. Nodes within our cluster use TCP channels to pass messages, which are serialized using the `bincode` library [90]. These channels have `TCP_NODELAY` set to further minimize communication latency.

DFIR comes with a built in deployment toolkit, which we use to programmatically configure network topologies and deploy our implementation to the cloud. Our benchmarking toolkit automatically provisions the appropriate machines, copies statically linked binaries to them, and performs a simple service discovery to establish the tree topololgy. In fact, a reader can reproduce our full set of results by installing the Hydro CLI and running a single Python script!

### 4.7.2 Memory Consumption

The key feature of the OnceTree protocol is its constant memory consumption with respect to the number of replicas. Memory consumption is measured using rss Unix counters at the process level of each process (each node). A separate process iteratively issues increment requests to a random element of a set of $2^{14}$ counters, which emphasizes the memory impact due to the number of counters on each node.

To pinpoint the effect of each *protocol* on the memory usage, we measure the *change* in memory utilization from when each node is initialized to the utilization after 60 seconds of randomized operations. We plot the median memory usage (as well as the 25th and 75th percentiles) across the cluster as we increase the number of nodes by powers of 2 in Figure 4.5.

Note the log x-axis, showing that the memory utilization of the classic CRDT counters is at least linear with respect to the number of replicas. On the other hand, OnceTree is unaffected by the number of replicas and holds at a constant, low memory usage (with any variation due to network buffers). Even with just $2^{14}$ counters, a relatively low number considering applications such as social media, the CRDT implementations use 10s or 100s of megabytes of memory at $2^8$ replicas, while OnceTree averages around 0.5 megabytes.

### 4.7.3 Propagation Latency

The key difference between OnceTree and classic CRDT replication protocols is how updates are propagated from a single replica to the rest of the cluster. Rather than using broadcasts to directly gossip updates to the rest of the cluster, OnceTree requires updates to follow a path through the tree to other nodes. These additional hops have the potential to impact the staleness of data, which is an important property for real-time applications where users expect to see live results.

To collect latency metrics, we deploy an additional node in the same zone as the rest of the cluster. This node iteratively sends a randomized increment request to a source node and measures the time until a pre-selected destination node sends a notification for the updated query

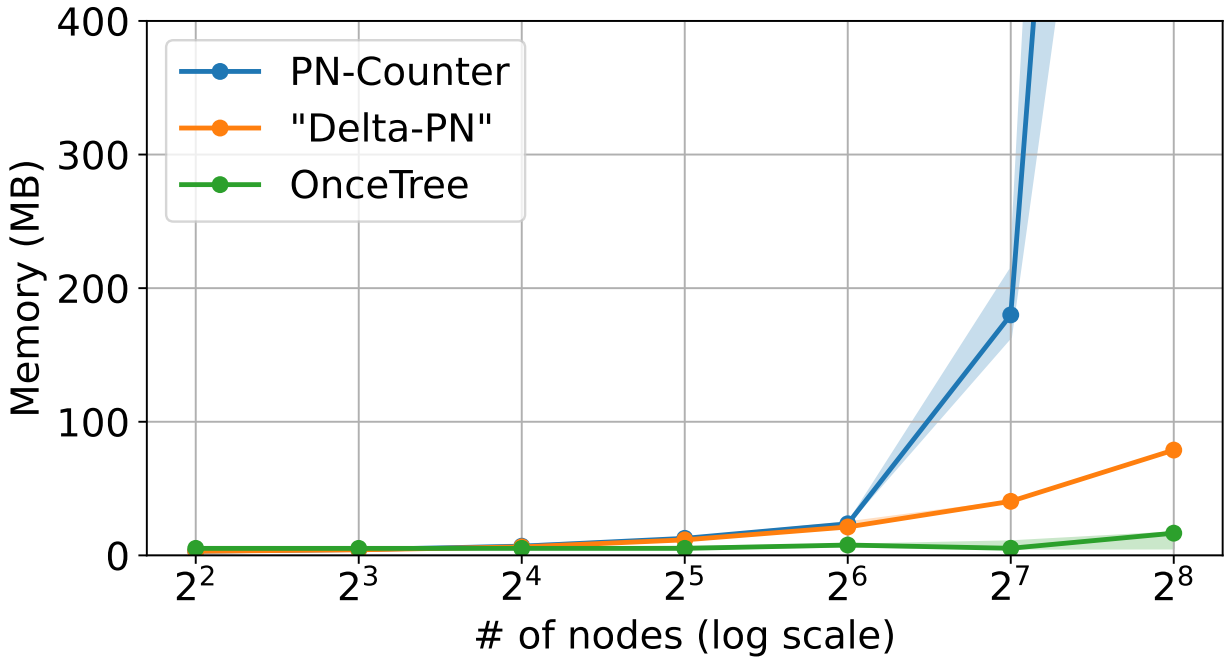



Figure 4.5: The median memory consumed by each node as the number of replicas is scaled.

result. Because we wait for a response before initiating a new increment, the latency metrics correspond to an *unloaded* system. We discuss the behavior of our protocol under the load of concurrent requests later in this section.

In our experiments, we compare the *worst-case* scenario for OnceTree against the *best-case* for PN-Counters. In our binary-tree topology, the longest path is between a pair of leaves on opposite sides of the root, so we pick the leftmost leaves as the sources (which receives increment requests) and the rightmost leaves as the destinations (which emit query updates). Because our CRDT implementation broadcasts gossip to the rest of the cluster, there is a direct link between any source and destination so we pick an arbitrary pair. A randomized gossip algorithm would have strictly worse latency, since the best case is still a direct link.

As in the memory utilization experiments, we average the latencies of increment operations executed across a 60-second period. We plot the results of our latency experiments in Figure 4.6. Note the logarithmic x-axis; each power of two corresponds to a tree topology that is one layer deeper or a cluster of CRDTs that is doubled in size.

Our empirical measurements line up exactly with the theoretical properties of our protocol. In particular, the propagation latency for OnceTree is proportional to the *logarithm* of the number of nodes. The propagation latency of the CRDT protocols is quite low for a small number of replicas, owing to the direct links between the source and destination nodes, but increases rapidly as the number of replicas grows due to the large amount of communication in the all-to-all topology.

We *do not* claim that OnceTree can achieve lower latency than CRDT replication protocols in

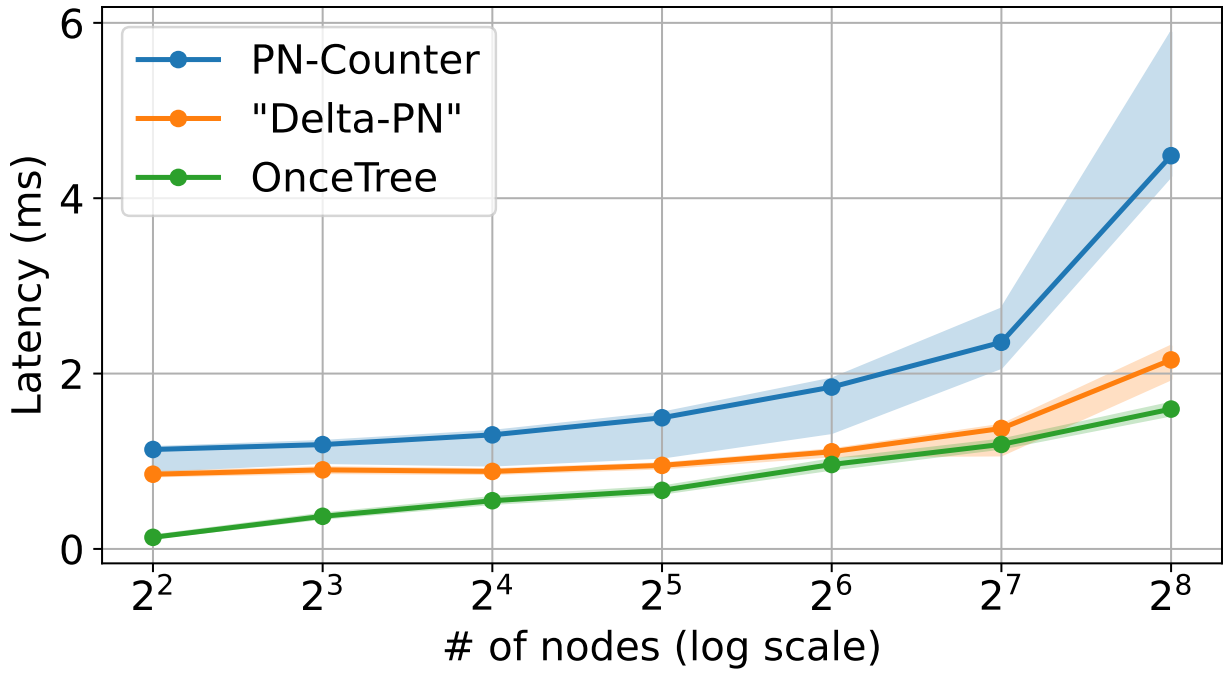



Figure 4.6: Propagation latency across a cluster for each replication protocol as the number of replicas is scaled.

general, since tuning can reduce the latency penalty for large clusters by gossiping to randomized subsets of nodes instead. But our experiments demonstrate that the additional latency from propagating through the tree remains low even as the number of replicas is scaled.

### 4.7.4 Operation Throughput

Finally, we explore the effects of concurrent request load on the OnceTree protocol and the CRDTs we compare it to. Rather than issuing increments sequentially, we allow a configurable number (powers of 2) of concurrent increments to be issued to the source replicas. This lets us analyze both the throughput of requests that can be achieved with each protocol and how the propagation latency changes as the load is increased. We plot the latency as the throughput of increments varies in Figure 4.7.

Once again, note the log x-axis for operation throughput. Because OnceTree can aggregate the query result at each node rather than having each node communicate its contribution directly to the rest of the cluster, it can scale very well as the number of concurrent operations increases with little latency penalty. On the other hand, the PN-Counter approach quickly saturates as more concurrent operations are issued. The mock "Delta-PN" protocol is able to match the throughput-latency curve of the OnceTree, because it sends far smaller packets when gossiping so avoids saturating.

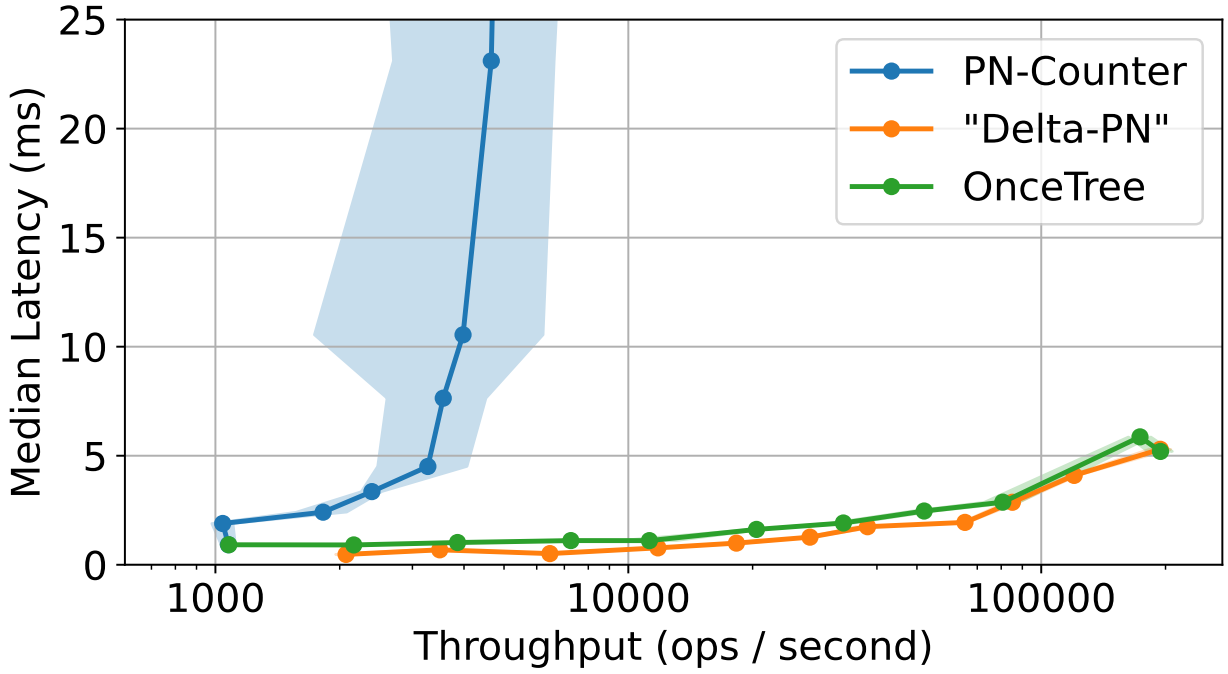



Figure 4.7: Latency versus throughput when varying the number of clients (# of nodes = 63).

These results demonstrate that the OnceTree protocol achieves its theoretical goals in a practical environment. Tasked with a common distributed systems task of maintaining a counter, OnceTree is able to dramatically reduce the amount of memory consumed by each node as the number of replicas increase. Furthermore, we demonstrate that the latency penalty of updates propagating across a tree rather than directly is low, and only increases proportional to the log of the number of replicas.

## 4.8 Future Work

There are many directions for future work. They fall broadly into three categories: (1) Supporting stronger consistency levels via tree toplogies (2) extensions to the algebraic connections between eventual consistency and query processing and (3) applying online optimization techniques to coordination-free tree reorganization.

### 4.8.1 Consistency and Recency Guarantees

One area of future work is pushing beyond strong eventual consistency to causal consistency, which CRDTs achieve via vector clocks [110]. It is not clear how to adapt our protocol to achieve causality in constant space. The constant space guarantee of the OnceTree protocol depends on

the ability to compose together all updates from a node into a single value which breaks the ability to interleave updates in causal order across nodes. We are currently exploring protocols that could achieve causal consistency in constant space per node under various assumptions. We hope to connect this exploration to the rich literature on vector clock optimizations for causal consistency in the distributed systems community. Of course, it is known that it is impossible to do better than O(n) space for causal consistency in the worst case [25].

### 4.8.2 Algebraic Formalizations

OnceTree offers what we call an "algebraic upgrade" from abelian semigroups to semilattices—that is, it automatically adds the "I" in "ACI" to arbitrary code that is already "AC". This raises two natural open questions: can one also develop "upgrades" to add associativity and/or commutativity to arbitrary code? If so, we suspect the solutions are unlikely to resemble the techniques in this paper.

The ability to upgrade structures with idempotence fits neatly with recent work in the database theory community. The seminal paper on provenance semi-rings by Green, et al. [46] demonstrated that many aspects of query processing and hence data provenance can be captured neatly in semi-rings—the familiar algebraic structure of elementary-school arithmetic with two operators $(+, \times)$ that are individually associative and commutative, and that follow the distributive law of $\times$ through $+$.

This work was recently extended in a best paper at PODS 2022 [3] to address fixed-point convergence, with a wide range of applicability to data provenance, multiset queries (SQL), recursive query processing, incremental maintenance of materialized views, and more. OnceTree can ensure that a semiring-based language continues to behave correctly in a distributed environment, enforcing idempotence "under the covers", and ensuring that the non-idempotent operators of the semi-ring are applied exactly once in a distributed setting, just as they would be in a centralized setting. We explore this question further in Chapter III.

### 4.8.3 Dynamics

Given our move operations in OnceTree, we are interested in dynamically modifying the tree online via these move operations to optimize a wide range of global objective functions. We can associate some cost with reconfiguring the tree and use online optimization algorithms to decide when to reconfigure. The biggest challenge is that we want to do this in a decentralized, coordination-free way using only local decisions of individual nodes and potentially stale information about the global state of the tree.

This is a challenging but fascinating topic. We have started to explore this for simple objective functions such as maintaining a balanced tree or a minimal spanning tree. We hope that in future work we are able to show coordination-free ways to achieve provable bounds on online approximations of such global objectives.

## 4.9 Related Work

OnceTree brings together several areas of distributed systems research: Idempotence enforcement, minimal CRDT representations, and computing distributed aggregates.

### 4.9.1 Idempotence Enforcement

We discussed in Section 4.2 the early database work on idempotence enforcement for business transactions and e-commerce. The early work on using queueing systems [17] is still the core way that many applications on the internet enforce idempotence. Applications use databases directly or they use services like Kafka [67].

Building on Quicksand [48] brought attention to the importance of ACI and the role of idempotence in that by introducing the ACID 2.0 terminology. In CRDTs, idempotence is often created by wrapping update operations into heavyweight structures like causal trees [30, 87, 109] or NodeIDMaps [102, 103].

### 4.9.2 CRDT Memory Efficiency

The challenge of memory overhead has been a long-time pain point for CRDTs not just for idempotence enforcement, but for many aspects of attempting to represent applications in this restricted programming model [80]. Bawens et al look at when it is safe to garbage collect state from CRDTs [16], which reasons about the history of operations which is automatically erased by our aggregation approach. A popular technique for smaller vector clocks in CRDT applications is Dotted Version Vectors [95] which limit the size of vector clocks for a given data item to just the nodes that modify the data item rather than every node in the system. We do not claim to solve the vector clock problem as the OnceTree protocol does not support full causal consistency. In fact, there is a strict lower bound for vector clock size that tells us that for the general case we actually can't do any better than O(n) sized vector clocks [25]!

Handoff Counter CRDTs [5] are another approach to reducing the memory overhead of CRDTs that also rely on restricting which nodes communicate with each other. They organize nodes into hierarchies without distinguishing between clients and servers. Instead of a fixed tree topology, nodes within a tier exchange data with each other (admitting multiple roots) and can send updates to multiple parents within lower-numbered tiers. To avoid duplicate updates, nodes request *tokens* that reserve an update *slot* in the parent tier. Similar to OnceTree, Handoff Counters maintain a monotone lower bound of lower tiers (parent) in a separate field *below* to calculate the safest, maximum value to report in queries.

### 4.9.3 Distributed Aggregates

Along with TAG [79], there has been a lot of work on robust aggregation in sensor networks [55]. For structured trees, one approach, offered by Chitnis et al [27], is to construct a "hybrid" topology with a mix of all-to-all gossip and structure.

In many ways the Synopsis Diffusion [86] line of work is the most directly related work to OnceTree. While their problem statement is to send data along multiple paths to reduce the error of sensor aggregation, the solution they end up at is all about ensuring idempotence so that the data arriving on multiple paths doesn't get counted more than once. Their solution is semi-lattices, all the way back in 2004! Their idea is to take aggregation functions that aren't intrinsically idempotent (like SUM) and to convert them to an idempotent representation. The catch that renders them distinct from our setting is that they accomplish this via duplicate-insensitive sketches that represent the true aggregate result approximately.

## 4.10 Conclusion

We have presented OnceTree, a solution to the memory consumption problem for coordination-free idempotence enforcement. Our work draws on connections between three disparate areas of data systems: distributed data aggregation, algebraic eventual consistency, and network communication protocols. We observe that idempotence can be enforced hierarchically, and therefore structured tree communication topologies are optimal for memory consumption. By combining static aggregation trees with single-path flooding we are able to guarantee deduplication and strong eventual consistency with constant space overhead and logarithmic latency.

# Part III

# Co-Habitation of Algebraic Structures in Distributed Data Systems

# Chapter 5

# Wrapping Rings in Lattices: An Algebraic Symbiosis of Incremental View Maintenance and Eventual Consistency

We reconcile the use of semi-lattices in CRDTs and the use of groups and rings in incremental view maintenance to construct systems with strong eventual consistency, incremental computation, and database query optimization.

## 5.1 Introduction

Algebraic models have been growing in popularity recently in both distributed systems and databases. In distributed systems, the semi-lattice model popularized as conflict-free replicated data types (CRDTs) offers an algebraic perspective on strong eventual consistency. In the database community, a generalization of relational algebra to semi-rings was promoted in the study of database provenance and a similar view in terms of full rings has been used to study database incremental view maintenance (IVM) [24, 66]. Both the distributed model of semi-lattices and the incremental view-maintenance model of rings have seen significant attention in building prototype systems such as the Anna distributed key-value store [110, 111] and the DBToaster Incremental view maintenance system [66].

In the interest of bringing declarative programming and automatic program optimization to distributed systems, we are concerned with incorporating the learnings from both of these lines of work into a single system. Initially, these two different approaches seemed mutually exclusive. Lattices grow monotonically while groups and rings require an inverse operation that is provably non-inflationary. This paper is about how to fit these two puzzle pieces together.

Our solution is not to try to get them to work together in one structure, but to separate them out into different layers that operate independently and only interact through a translation layer. This layered approach lines up with the separation of concerns that each structure is meant to deal with in a distributed system. The role of CRDTs and lattices is to ensure robustness to nondeterminism on our asynchronous network. The group and ring approaches were designed for single-node systems and play no role in the network layer of our system. By separating these, we get the benefits of the lattice at the network layer and the benefits of the groups and rings at the query processing layer.

The remainder of this paper is organized as follows: In Section 2, we give background on semi-lattices (CRDTs) for strong eventual consistency and on the use of abelian groups in incremental view maintenance. In Section 3, we show why the strawman of combining groups and lattices into one structure fails and then give two constructions for the co-habitation of semi-lattices and abelian groups. In Section 4, we give background on rings in IVM. In Section 5, we discuss the different deletion semantics used in IVM and CRDTs. In Section 6, we discuss future work on algebra-aware data systems. In Section 7, we discuss related work, particularly op-based CRDTs and $\delta$-CRDTs.

## 5.2 Background

In this section, we give necessary background on abelian groups for incremental view maintenance. Definitions of mathematical terms from abstract algebra are provided in the background section of this thesis. Feel free to skip this section if you are already familiar with this topic.

### 5.2.1 Groups and Incremental View Maintenance

Incremental view maintenance (IVM) is the study of how to efficiently maintain query answers over a database as the contents of the database gets updated over time. It has been studied in academia for decades and the techniques have been commercialized in a number of database systems including Materialize [72], Feldera [23], Azure Synapse [105], Amazon Redshift [13], and Databricks [34]. They continue to be a popular topic of academic research in the database community [40, 58, 66, 88] and were the topic of the 2023 VLDB best paper award [24]. For an excellent survey of incremental view maintenance see [26].

For simplicity of explanation, we start with the group-theoretic model of incremental view maintenance used in DBSP [24]. In Section 5.4, we extend our exploration to the ring-theoretic model which simply adds a second operator to the group to express more complex queries.

There are two common semantics for databases, the "set semantics" in which every tuple occurs at most once and the "bag semantics" in which tuples can occur multiple times [1]. The former gives us a data model in which the database is a set of tables and each table is a set of tuples. The latter gives us a data model in which the database is a set of tables but each table is a multi-set of tuples. Each tuple is "tagged" with a counter indicating its multiplicity. DBSP utilizes a generalization of these two models to enable positive *or negative* multiplicities of tuples which is called a "Z-set" [24, 46] because each tuple has a multiplicity from the integers ($\mathbb{Z}$). The benefit of allowing negative multiplicities is that we can now talk about insertions of tuples and deletions of tuples from the database in a unified way. A deletion of a tuple is simply its insertion with a multiplicity of negative one. A modification of a tuple is the deletion of that tuple followed by the insertion of the modified value.

Considering our state to be Z-sets and our incoming updates to be Z-sets as well (batches of tuples being inserted or deleted), we see that the update operation forms an abelian group. That is, our update is associative, commutative, and every incoming Z-set update, $u$, has a corresponding "inverse" Z-set, $u^{-1}$, such that $\text{update}(u, u^{-1}) = (T, 0)$.

With this model of state and updates to state, we can define operators over these Z-sets and queries composed out of operators. Given certain properties on the operators, we are able to guarantee the "incremental" computation of query results: that is, work done to compute the result after a new update can be proportional to the size of that update rather than to the size of the entire database.

The essential property of an operator that makes it efficiently incrementalizable is that it is *linear* which is defined as $f(a+b) = f(a)+f(b)$ where + is the group operator. This is equivalent to saying that operator $f$ is a homomorphism over our group. We can see that if an operator is linear then we can compute this operator incrementally (we already have $f(old\_db\_state)$ and when a new update comes in we just do $f(new\_update) + f(old\_db\_state)$).

It turns out many useful operators are linear with respect to this Z-set abelian group such as selection and projection in database queries. Linearity is also composable, so any query we can construct as a composition of linear operators will be efficiently incrementalizable.

Another key category of operators is *bilinear operators* which are binary operators $f(a, b)$ that satisfy distributivity over +. The classic example of a bilinear operator in database queries

is a join. We can think of bilinear operators as functions that would normally cost $N^2$ time in the database instance size to compute, but in the incremental setting we can compute them in time (update_size $\times$ N). Intuitively, when a new tuple arrives on one input to the join we need to check it against each existing tuple received on the other input once to compute the join.

It turns out that quite expressive query languages from databases are entirely incrementalizable in this model including relational algebra, Datalog, grouping, and aggregation– i.e. much of SQL and beyond. [24].

This DBSP model offers us considerable power, but is it compatible with eventual consistency? In the next section, we give a construction for combining group-theoretic incremental view maintenance and lattice-theoretic eventual consistency.

## 5.3 Co-habitation of Abelian Groups and Semi-lattices

When we first wanted to combine these two structures, we asked the seemingly obvious question "Can the CRDT update or CRDT merge operation be the abelian group update operation?" First we will show why these two strawman approaches cannot work and then dive into our multi-layer solution.

**Strawman 1: CRDT Update as Group Update**: The update operation of a CRDT must grow monotonically with respect to some partial order ie be *inflationary*. An abelian group update operation must have an inverse update operation. For both of these to be true, the group can only have one element (rendering it useless for expressing application semantics):

*Proof:* Assume + is inflationary ($\forall x, y : x + y \geq x$. Then $x + -x \geq x$ and $-x + x \geq x$. Adding x and -x to the respective sides we get $x \geq 0$ and $-x \geq 0$. We know $x + -x = 0$ so $0 \geq x$ and $0 \geq -x$. So $x = -x = 0$ for every x in the group.

**Strawman 2: CRDT Merge as Group Update**: Recall that a CRDT merge operation must be idempotent. We show that if an abelian group operation is idempotent, then the group must also be the one element group $(\{0\}, +)$.

*Proof:* For any idempotent element $x$ in the group $G$, we have that $x + x = x$. Due to invertibility of +, $x$ must also have an additive inverse, $(-x)$. Adding this to both sides yields $x = 0$, meaning all elements must be the additive identity. Thus, with an idempotent + operation, $G$ must be the one element group $(\{0\}, +)$.

Powering through the disappointment of these failed strawmen, we find that there is still a way for CRDTs and IVM groups to co-habitate! The trick is in separating the CRDT from the group and adding a translation layer that allows these two structures to co-exist. We first give a working construction for this using a simple set CRDT that is inefficient but demonstrates how the group and semi-lattice combine. In the next section, we provide a variant with a performance-optimized CRDT based on delta-CRDTs [75].

The key observation to see how group and semi-lattice structures can be combined is to think about how these two structures are really being used in our data system. We have some state of our data. We want to modify that state in an incremental way. For that, we need a + operation that

forms an abelian group. We can then express dataflow queries with linear and bilinear operators over that + operation.

Then what are lattices for in our data-intensive systems? **The role of the lattice is to allow us to replicate state while being protected against nondeterminism of computer networks.** The network plays no role in our data modification (+) or query evaluation. The network is a different layer of the system. Much like we write our application semantics without concern for how TCP is being used for delivery, we can write our group-based or ring-based application without concern for how our lattice is handling network nondeterminism. Much like with TCP, we leave our application alone with its + operation, and then at a lower layer we propagate updates around the system wrapped up in a nice robust semi-lattice. We call this construction a "lattice-wrapper" and depict this idea visually in Figure 5.1.

### 5.3.1 The Very Simple Construction

The pseudocode for this construction is given in Listing 5.1. At a high level, input updates arrive of type Z-set to apply to the local group structure at a replica. We pass the Z-set value into the group to modify the group state and update the materialized views at that replica. We also convert this incoming Z-set value into a semi-lattice value by pairing it with a randomly generated unique ID. The (updateID, Z-set) pair is propagated to other replicas which process this update by keeping track of the set of updateIDs they have seen and ignoring any repeated updateIDs. When the receiver hasn't seen the incoming updateID before, it adds it to its list of seen updates and passes the Z-set payload into its local group to be processed. Our lattice state is a set of updateIDs which we modify (merge) via set union - an associative, commutative, and idempotent operation.

Note that in the design that we have described, we do not enforce that updates arrive in FIFO order or causal order. It is only a couple extra lines of code in our merge function to do this (we wait until we see the updates in-order from the sender before passing them into the group), but because our update operation is an abelian group it is commutative, so we don't need to bother with any extra metadata or logic for enforcing update ordering. This commutativity of the group frees us from the worry of ordering; eventual consistency and linearizability give the same results at the application level for abelian group-based and ring-based applications.

### 5.3.2 The Performant Construction

The simple construction above requires each node to keep track of the list of every updateID they have ever seen. To improve on this metadata overhead, we give a construction in this section that reduces this metadata to be proportional to the number of replicas on average rather than the number of updates in the system.

The lattice wrapper we use is similar to the delta-CRDT lattice wrapper [75] and familiar in networking literature as a part of the TCP protocol. In English, each replica has a uniqueID and a local logical clock that it increments each time it receives a local update to the group. It uses this (uniqueID, localLogicalClock) pair as the updateID key to the Z-set update payload. It sends the

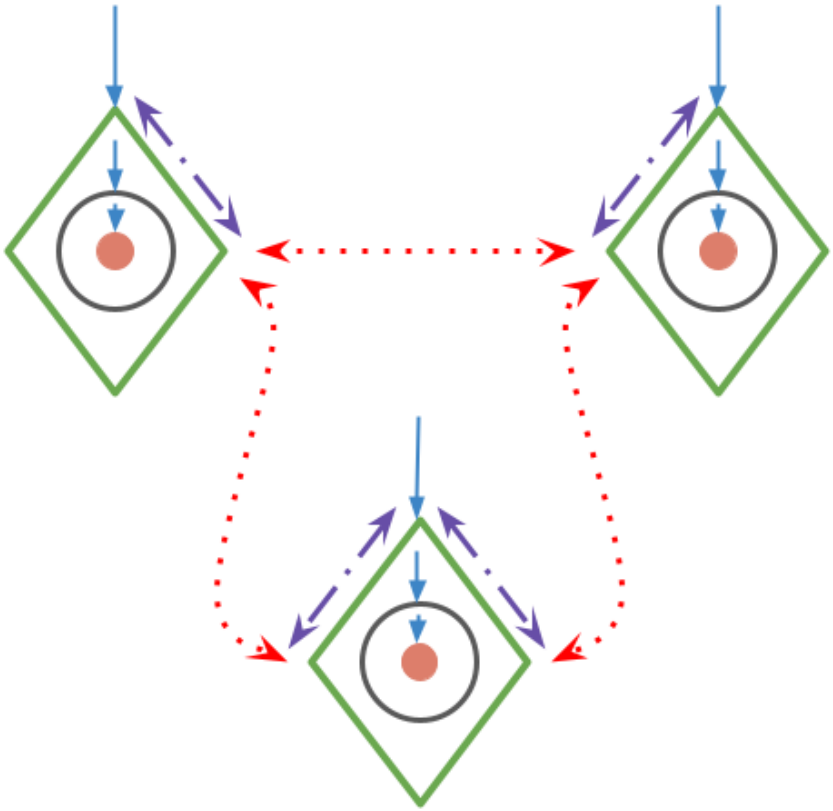


Figure 5.1: We depict three replicas of our lattice-wrapped view groups. Blue solid arrows are the incoming updates to the database instance of type Z-set. Red dotted arrows are CRDT merge operations being broadcast to each replica. The green diamonds represent the lattice wrappers and we see the solid blue updates are converted into dotted red merges via the lattice wrapper. The conversion is depicted by a purple semi-dotted arrow. We see that the updates are passed through to the black group (circle) inside the lattice wrapper. The red dot in the center is the materialized view that users can observe and where the dataflow pours into. We see that merge operations are received by the lattice wrappers and converted into update operations that are then passed through to the inner group structure.

element of the map resulting from each update along to the other nodes in the system. A replica receiving one of these map elements processes it using the merge operation, which like before simply checks whether that (uniqueID, localLogicalClock) is already in the set of updates the receiving node has processed. To reduce metadata overheads, the sequence of clock times per node can be stored in heavily compressed representations as they are long runs of contiguous integer values. The receiving nodes will also acknowledge received updates to the sender so the sender can garbage collect its map of update payloads once each other replica has acknowledged hearing about that update payload. The sender will re-send updates to nodes that haven't acknowledged

Listing 5.1: Pseudocode for Z-set lattice wrapper construction

```
let mut my_group = Group::Zset::new();
let mut my_inbox = Lattice::Set::new();

fn processUpdate(&mut self, update_payload: &Zset) {
        self.my_group.apply(update_payload);
        let update_id = Uuid::new_v4();
        let update_wrapper = (update_id, update_payload);

        my_outbox.insert(update_wrapper);
        neighbors.send(my_outbox);
}

fn receiveUpdate(&mut self, (incoming_uuid, incoming_zset): (Uuid, &Zset)) {
        if (!my_inbox.contains(incoming_uuid)) {
            self.my_group.apply(incoming_zset);
            my_inbox.insert(incoming_uuid);
        }
}
```

their receipt after a fixed time interval. For more details on delta-CRDT constructions, see [75].

## 5.4 Rings in Incremental View Maintenance

One of the major successes of databases is the power of query optimizers which take a logical query plan (similar to an abstract syntax tree) and search the space of equivalent dataflow graphs (physical query plans) for the one that will have the best performance. The space of equivalent dataflow graphs is defined in terms of different rewrite rules that can be applied to the logical query plan without changing the result of the query. These rewrite rules can be seen as algebraic axioms on the operators in the query language.

Common optimization techniques include choosing what order to execute joins in (associativity of join), choosing what order to apply filters in (commutativity of selection), and choosing whether to push filters before joins in the query plan (distributivity of filters over join). One observation is that the standard axioms that optimizers leverage for relational algebra form a semi-ring. A semi-ring at a high-level is a structure with two operators where one is associative and commutative, the other is just associative, and the second operator distributes over the first operator (see 2.1 for full definition). With this observation, database researchers have generalized the applicability of database techniques beyond relational algebra to any pair of operators that satisfy the algebraic axioms of a semi-ring. This semi-ring perspective has been popular recently in databases for studying data provenance [46], recursive queries [3], and incremental view maintenance [66].

The reason that we focus on rings instead of semi-rings in incremental view maintenance is because rings have inverses on +, and + corresponds to the update operation on the data. Thus, the + in a ring forms an abelian group and gives us the nice properties we discussed in Section 5.2. The $\times$ operator in the ring is analogous to *join* in relational algebra.

How does this picture of incremental view maintenance as a ring instead of as a group change our picture and lattice wrapper construction? The answer is that it doesn't need to change our lattice wrapper at all as the group and lattice structures already operate independently. The ring just means that the queries we express over the state of the group can be optimized with rewrite rules and get more performant dataflow graphs for the computation of views inside our group.

## 5.5 Inverses, Two-Phase Sets, and the Semantics of Deletion

In this section, we take a brief detour to discuss the interesting differences in deletion semantics found in Z-sets compared to CRDTs. We do not conclude that one semantics is better or worse than another; we simply highlight the differences and observe that some applications would prefer one and some would prefer the other.

Deletion semantics in **CRDTs** have been a long-standing challenge. In order to satisfy the *inflationary update* requirement, complex lattice structures have been introduced offering different tradeoffs for deletion semantics. The two-phase set design [94], for example, circumvents inflationary updates, but runs into two other problems with deletion; the "natural" single-node semantics of deletion are neither commutative nor idempotent. Commutativity means the order of operations in a sequence doesn't change the outcome, but if we think about "insert A; delete A" vs "delete A; insert A" the common meaning of this on a single node interface would be that the result of the first operation sequence is the empty set and the result of the second sequence is the set $\{A\}$. This non-commutativity leads the two-phase set design to treat all deletions as if they occur after all insertions - a decision that is quite unsatisfactory in the simulation of a single-node user experience.

Another problem with this two-phase set design is that in a common interface it should be possible to insert something, remove it, and then insert it again. Under the idempotent interpretation of deletion used in the two-phase set this is not possible; the item can be inserted at most once and deleted at most once. This deletion with *at most once* semantics is often referred to as "tombstoning".

To resolve these awkward semantics, the observe-remove set (OR-set) [96] additionally tracks the causality of updates to fix the idempotent deletions. The causality also fixes the single-node experience of "delete A; insert A", but for concurrent writes ambiguity in semantics remains, so a user must pick whether to default to deletes first or inserts first.

**Incremental view maintenance** literature has focused on the single-node setting and in that setting they are able to take a very simple and clean view of deletions. We treat the database instance as a Z-set under the hood. An insertion or deletion from the database increments or

decrements the multiplicity of the specified tuple. For user-facing multiset semantics we may display all negative multiplicities as 0. For a CRDT-view of this we can think of each tuple as having a pn-counter as one of its columns representing the multiplicity of the tuple.

The Z-set semantics itself is not a valid semantics for a state-based CRDT as the update operation is not inflationary and the state changes in non-idempotent ways, but we know that those concerns can be handled in our lattice wrapper layer, so what about the deletion semantics themselves? For Z-sets, insertions and deletions are fully commutative, so if a user issues a delete and expects the count to go from 0 to 0 then this will be broken. Fully commutative updates also lead to the (insert A; delete A) vs (delete A; insert A) anomaly from two-phase sets. However, prevention of both of these scenarios can be handled at the client by ignoring deletions when the observable count at the client is 0. The "observable count at the client" is exactly the updates that occur causally before the new update at that client.

One anomaly that can still occur is if two people see there is a count of one and simultaneously decide to delete it. In Z-set semantics this would result in a count of -1 if the deletions both occur within the gossip time window ("concurrently"). The best semantics for such a case is application specific, but the Z-set construction with client-side guards avoids tombstoning and achieves something that could reasonably be called "causal multiset semantics".

## 5.6 Discussion and Future Work

Exploring these two algebraic lenses allows us to get the best of both worlds in our systems. This is a step towards enabling algebra-aware systems in which the algebraic properties of application logic are known and can be utilized by the system as much as possible. In this paper, we began this journey by connecting just two of the many algebraic optimizations known in computer science. In future work, we plan to draw connections to the semi-ring approaches to provenance [46] and fixed point computation [3] from databases as well as to areas of computer science beyond databases. Algebra has played a central role in modern cryptography including the use of group and rings for public key cryptography [98] and the use of ideal lattices for fully homomorphic encryption [42].[1] Algebra also shows up in program analysis [22] as well as high-performance computing [60, 117]. Once we have a system that understands algebraic properties, we hope to apply the learnings from these other fields to it as well.

## 5.7 Related Work

Our study of the co-habitation of semi-lattices and rings is inspired by recent work in **incremental view maintenance** such as DBSP [24], DBToaster [66], and Differential Dataflow [85]. DBSP takes a group-theoretic view of updates and focuses on single-node updates without query optimizations. DBToaster was another single-node system based on rings and supporting traditional

[1] Different type of lattice than the semi-lattice used in CRDTs.

relational query workloads with optimized and incremental updates and queries. Differential Dataflow is an incremental computation framework that considers the distributed case, although not specifically in the context of eventual consistency.

To our knowledge, we are the first work to explicitly connect the algebraic view of incremental view maintenance from the databases literature and the algebraic view of eventual consistency from the distributed systems literature. Other works have explored distributed IVM in the context of transactions [54] and data warehousing [4, 115, 116].

We have focused our attention in this paper on state-based CRDTs rather than op-based CRDTs. The reason for this is that state-based CRDTs are modeled naturally as algebraic semi-lattices and their relationship to rings is interesting. Op-based CRDTs can be thought of as delegating the semi-lattice properties to a networking layer and dealing only with the application layer properties. In a sense, this makes the op-based CRDT perspective and the co-habitating rings and semi-lattices perspective similar - the mechanisms for ensuring exactly once delivery over the network can be treated separately from the mechanisms for managing changes to application-visible data.

**Op-based CRDTs** require that the update operation be associative and commutative, forming a commutative monoid. This is much like an abelian group except it drops the requirement that updates have inverses. Some existing op-based CRDTs like counters [96] already have an inverse operation, forming an abelian group and being amenable to DBSP-style incrementalization. Other op-based CRDTs like set CRDTs do not support inverses, but they raise the interesting question of what classes of operators and queries over commutative monoids are automatically incrementalizable. The definitions of linearity and bilinearity are the same for commutative monoids as for abelian groups. We leave a formal treatment of this connection to future work.

**$\delta$-CRDTs** [75] are a design that minimizes the network overheads of state-based CRDT communication. Much like IVM, they make the network utilization proportional to the size of an update batch rather than proportional to the size of the state.

## 5.8 Conclusion

We have presented a way to utilize two different algebraic views of data systems, CRDTs and incremental view maintenance, in the same holistic system. The seemingly incompatible semi-lattice and ring structures can be made to co-habitate by using ring structures at the "application layer" and semi-lattices at the "network layer". With this, we are able to perform incremental computation of complex query plans while guaranteeing coordination-free strong eventual consistency.

# Part IV

# Simplifying the Developer Experience

# Chapter 6

# Emmy: Peering Into the UDF Black Box Using Formal Verification of Algebraic Properties

We introduce Emmy, a library that helps developers reason about algebraic properties of Rust code.

## 6.1 Introduction

Throughout this thesis we have explored the question of how algebraic properties can be used to reason about correctness and optimization in distributed data systems. This leaves the obvious question: how does the system extract algebraic properties from the code? Our approach in Hydro follows the same pattern that SQL systems have followed for decades. We first describe this paradigm for SQL systems and then describe how we leverage features of the Rust programming language and formal verification tooling to improve on the SQL approach in the Hydro stack.

The SQL language is designed as a programming interface based on the domain-specific language of relational algebra. Relational algebra contains just five operators which can be combined together to express a wide range of complex programs. Those five operators are *select, project, join, union* and *set difference*. SQL programs are compiled into logical query plans - dataflow graphs made up of these five relational algebra operators.

This creates a tractable surface area for program analysis in database systems. By encoding the algebraic properties satisfied by each of the five operators, the optimizers is able to reason about what optimizations to the dataflow graph are safe.

This approach has been enormously successful for data systems. However, there is one essential pain point for users: expressing arbitrary programs in relational algebra is often burdensome and sometimes completely impossible. Relational algebra is not Turing-complete, so there are many programs that are simply impossible to express in this language. Additionally, many programs are possible to express in the language but it is prohibitively awkward to do so. Data systems approach this expressivity problem from two directions. The first is to extend relational algebra to support a wider range of programs while still having full optimizer knowledge of the properties that they satisfy. The other direction is to support the insertion of arbitrary code into the dataflow graph via *user-defined functions* (UDFs).

UDFs can be written in Turing-complete languages, solving the expressivity problem. This solution comes at great cost though as optimizers don't have information about the properties of UDFs, and thus struggle to optimize programs that include UDFs. In the Hydro stack, we have this same dichotomy of built-in operators we can reason about, and Rust user-defined functions that we cannot. The role of Emmy is to bridge this gap, allowing the optimizer to reason about these Rust UDFs.

One approach that many SQL systems support to help optimize UDFs is to allow developers to *tag* the UDFs with properties. For advanced users, this is a plausible approach because the developer has deep knowledge of the UDF they have implemented and can simply label it with the relevant properties that it satisfies. However, this is not the common case in SQL systems. Studies show that developers label their UDFs incorrectly 50% of the time [76]. Emmy approaches this problem from two angles. For the advanced users, Emmy provides tooling to ease the burden of labeling UDFs with algebraic properties. On the other hand, exposing these details of how the optimizer works to the developer breaks the promise of a declarative experience for the user. In the interest of allowing developers to truly express their goals declaratively and leave the rest of the work up to the system, Emmy also supports a *zero-touch* experience for determining algebraic properties of UDFs.

Recent advancements in formal verification allow Emmy to walk a fine line of leveraging powerful verification tools while hiding them from the developer experience. The reason this is difficult is same reason why we think it is necessary: verification tools are hard to use. Despite rapid improvements in the effectiveness of these tools, they still see little adoption in industrial systems. There are a few key challenges in the usability of verification tools. The first is that they typically require the developer to re-implement their code in a spec language of some kind. The second is that the programming paradigm of expressing the goals for the tool and guiding the tools towards those goals are unlike any coding paradigms found in mainstream software engineering. To address these challenges, Emmy leverages recent verification tools that can run directly against Rust code - no spec implementation necessary. Further, Emmy encodes the algebraic properties of interest directly in the verification languages, so no developer interaction is required.

In this Chapter, we introduce Emmy: A system for zero-touch proofs of algebraic properties of Rust programs. We validate the effectiveness of the zero-touch proofs approach on a dataset of Rust programs scraped from Github.

## 6.2 Emmy Verification Tools Background

**Fuzz Testing** is the process of programmatically testing huge numbers of random inputs to a piece of code. While fuzz testers cannot brute force all possible inputs for most data types, they can attempt thousands of inputs within seconds. This makes them effective for quickly finding examples where a test case fails. Because they are not exhaustive, a fuzz tester cannot prove that a property is true on all inputs. Fuzz testers can however prove that a property is false (by providing a counter-example). Fuzz testers typically have no limitations on the data types or kinds of code they can support.

**Bounded Model Checking** is a popular formal methods tool for verifying properties of hardware and software designs. Model checkers explore the space of possible states of the system for any states that violate the desired properties. For systems with a small number of states, this exploration can be exhaustive. When the goal is a complete proof, this typically constrains bounded model checkers to operating over bounded data types.

Most model checkers operate against a spec implementation in a spec language rather than real production code. This is the case for TLA+ for example, and rewriting code into a spec language is a major source of friction for developers adopting model checking in practice. The challenge is not just the duplicate work of implementing things once in a spec language and then again in a production language. The big problem is that any mismatch between the production implementation and the spec implementation renders the model checker results invalid. This means that as the code inevitably changes over time, if the spec is not accurately maintained then the benefits of model checkers are lost.

Recent tooling addresses this challenge in the Rust ecosystem by supporting bounded model checking directly on Rust code. The tool for this, Kani, is the tool we use in the Emmy library to ensure a zero-touch verification experience for developers building with Emmy.

## 6.3 Emmy Architecture

In this section, we describe the architecture of Emmy. First, we discuss our disiderata for what guarantees and experience we want to offer users of Emmy. The primary requirements that drive the design of Emmy are two simple goals.

1. Developers don't have to read or write verification code.

2. We must have 0% false positives on determining that an algebraic property is satisfied.

Each of these requirements has numerous implications for our design. The first design goal is motivated by the fact the custom languages and paradigms of thinking used by formal verification tools are what has been holding back their adoption in production software environments. Most software developers have no training in these languages like TLA+, Coq, or Lean and the programming paradigm is quite foreign. While these tools are highly valuable in what they can guarantee about a system, any design that relies on developers reading or writing these languages is unlikely to be adopted in practice.

Our second goal stems from the relative consequences of a false positive vs. a false negative in our setting. Falsely determining that an algebraic property is satisfied when it is not would mean the optimizer may transform the dataflow graph in ways that give incorrect behavior or outcomes. With the zero-touch experience of Emmy, this entire process would be hidden from the user and they would simply see incorrect results with no indication of why. This is, of course, an unacceptable user experience. On the other hand, a false negative simply means that we were given a UDF that satisfies some property, but we weren't able to prove the property and have to treat the UDF as if it did not satisfy it. The result is that certain optimization opportunities may be lost, but this is a performance concern not a correctness concern. Further, the UDF performs the same as if Emmy didn't exist in this situation. In a sense, the 0% false positives means Emmy is "all upside" relative to not using it.

In combination, these two requirements eliminate a large set of verification tools from our options. Most verification tools require expressing the code developers want to prove their property of in a custom verification language, and with our goal of 0% false positives, we cannot afford an error-prone code translation process from the developer language to the verification language. Even if a translation tool can have very high translation accuracy, we cannot assume the human in the loop is going to read the output verification code and determine it matches the input Rust code.

This leaves us with verification tools that can run directly against the user-written Rust code. The ecosystem of such tools is small, but stronger for Rust than for most other programming languages. In Emmy, we choose two tools in particular that can be run directly against Rust code, a fuzz tester called called *cargo fuzz* [100] and a bounded model checker called *Kani* [107]. These tools offer an interface where users can define invariants they want to prove about code and then pass in Rust code directly to test these invariants. In the case of Emmy, the invariants we are interested in are algebraic properties (e.g. associativity and commutativity) which we are able to hardcode into Emmy, so users of Emmy don't need to write invariant code at all. We are then

able to run the relevant tool with each algebraic property check we are interested in against the user's Rust code.

Figure 6.1 depicts the expression of the associativity property in the Kani verification language. Defining the properties for the fuzz tester follows a similar format. Today, Emmy supports the three algebraic properties of CRDTs (associativity, commutativity, and idempotence) for verification with Kani. Emmy supports every property described in the Algebra Background section of this thesis for fuzz testing. Defining each of these properties in Rust and supporting them in the fuzz tester adds up to roughly 2000 lines of Rust code.

Recall that bounded model checkers can only prove properties with 100% certainty for bounded data types, e.g. integers, booleans, or floating points. For a truly zero-touch experience, Kani is the tool Emmy uses to prove properties. For bounded data types, we support this zero-touch verification experience in Emmy today. Kani is able to prove truth or falseness for each algebraic property of interest a high percentage of the time as we will see in Section 6.4.

For unbounded data types, such as sets or vectors, we instead offer the user a co-pilot experience through our fuzz tester. Recall that fuzz testers cannot prove that a property is *true* for large input spaces, they can only find counter-examples that prove it is *false*. We set a time limit of one second for the fuzz tester to run. If it finds a counter example, we know that property is false. If it does not, we will not assume the property to be true in the system, as this would break our 0% false positives assumption. However, we instead can surface this information to a user that can use it as a form of co-pilot in reasoning about the properties of their UDFs. If they choose to, they may optimistically tag properties as true with the help of the fuzz testing results, but we can no longer guarantee correct code generation.

```
fn test_associativity<T: Clone + PartialEq + kani::Arbitrary +
   std::marker::Copy>(input_function: fn(T, T) -> T) {
    let a: T = kani::any(); // Generate any value for a
    let b: T = kani::any(); // Generate any value for b
    let c: T = kani::any(); // Generate any value for c

    let result1 = input_function(a, input_function(b, c));
    let result2 = input_function(input_function(a, b), c);

    assert!(result1 == result2, "{:#?} is not associative",
       input_function);
}
```

Figure 6.1: The code for encoding the three CRDT semi-lattice properties in the Kani verification language.

## 6.4 Evaluation

In this section, we explore the efficacy of the zero-touch verification experience in Emmy. Specifically, we evaluate the succes rate of the Kani bounded model checker in proving algebraic properties of UDFs. The goal of Emmy is to determine algebraic properties of UDFs for users of the Hydro language. There is no existing Rust UDF dataset, so to evaluate Emmy we scraped Github for examples of potential Rust UDFs. In particular, we scraped all Rust closures passed into *.reduce()* and *.fold()* operations. *.reduce()* and *.fold()* are the two standard ways we allow users to pass in UDFs in Hydro, so we determined that uses of Rust closures within these operations on Github form a decent proxy for what UDFs developers may write in Hydro.

In scraping Github, we downloaded roughly 131,000 uses of *.fold()* and 22,000 uses of *.reduce()* in Rust. In our evaluation of the Kani bounded model checker, we filtered this dataset down to functions over bounded data types that are entirely self-contained Rust closures. That is, the selected Rust closures do not call external libraries or functions that are defined elsewhere in the repository. We then deduplicated the syntactically identical closures such as the many occurences of *|a, b| a + b*.

For the case of *.folds*, we found 4,913 valid and syntactically unique such Rust closures. The result of Kani for each property is shown in Figure 6.2. We see that Kani performs quite well for all three properties with a successful proof rate of 86% for associativity, 86% for commutativity, and 99% for idempotence across the dataset of syntactically unique closures.

Our evaluation of Kani on the syntactically unique *.reduce()* dataset is depicted in Figure 6.3. We see a 83% success rate for proving associativity, 83% for commutativity, and 99% for idempotence across the dataset of syntactically unique closures.

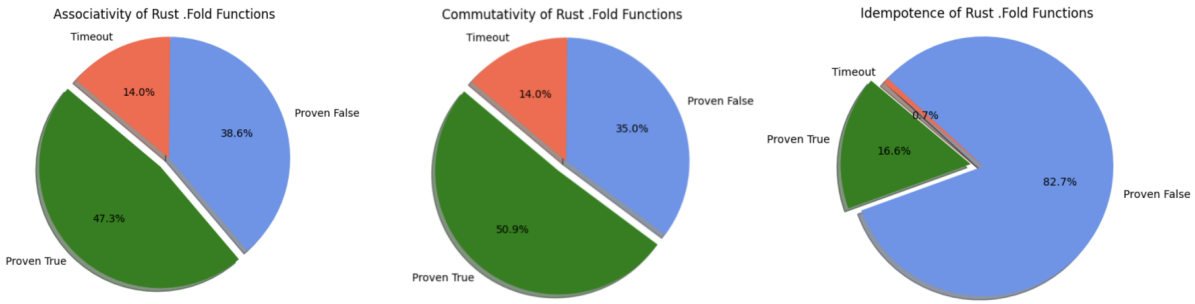



Figure 6.2: Results of running Kani against the Github *.fold()* dataset of Rust closures.

For the smaller *.reduce()* dataset, we were able to manually examine each UDF to determine the set of semantically unique functions out of the 1058 syntactically unique UDFs. We found 61 semantically unique UDFs satisfying our bounded data type and self-contained closure restrictions. As expected, simple arithmetic functions like addition and multiplication occur many times in the dataset.

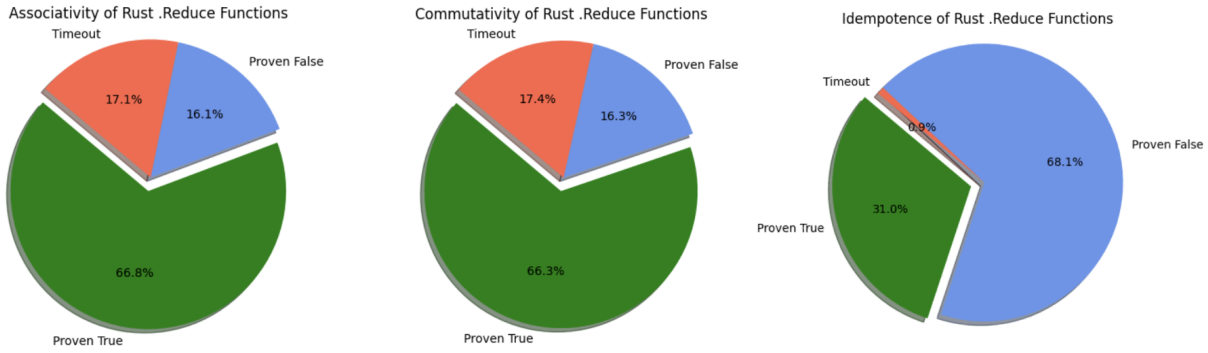



Figure 6.3: Results of running Kani against the Github *.reduce()* dataset of Rust closures.

## 6.5 Examples of Verification with Emmy

In this section, we look at the use of Emmy on examples relevant to the OnceTree protocol and algebraic idempotence upgrades as described in Chapter II. In order for an aggregation function to be amenable to the OnceTree idempotence upgrade, we require that it be associative and commutative. If the function is already idempotent, then the system doesn't need to bother introducing an idempotence upgrade. Some examples from our Github dataset that satisfy all three properties are logical OR, logical AND, integer minimum, and integer maximum. Kani was able to prove all three properties are true in each of these cases. In our dataset, we saw a variety of different implementations of these and Kani had no trouble with this variability. For example, Figure 6.4 depicts four different implementations of the minimum function in our dataset. One via the standard library, one via an *if statement*, and one via a *match expression*, and one via a min function. One idempotent UDF that Kani was not able to prove is *least common multiple*. This is a more complex function involving a loop and Kani timed out on this in our experiments running the tool for up to one minute. The code for this example from the dataset is available in Figure 6.5.

In our dataset, we also observed a variety of functions on bounded data types that are associative and commutative, but not idempotent. These include addition, multiplication, and XOR. Kani times out when attempting to prove these properties for multiplication which is a known weakness in bounded model checking. However, it is able to prove that the other two functions are associative and commutative and that they are not idempotent. We see that for the purposes of OnceTree, Emmy is an effective tool for exploring the relevant UDF properties. All properties of interest to OnceTree are supported and OnceTree's requirement that functions be aggregate functions (mapping collections to singletons) means that there will often be a pairwise operation on the elements of the input collection that operates only on bounded data types. While an array data type isn't supported by our bounded model checker, most aggregates (e.g. addition, multiplication, averaging, or standard deviation) can be defined as a function over two bounded data type inputs which are applied as a left-fold over the input collection.

```
// Conditional expression
|a, b| { if a < b { a } else { b } }

// Using standard library function
|a, b| std::cmp::min(a, b)

// Match expression
|a, b| match a < b { true => a, false => b, }

// Method call
|a, acc| a.min(acc)
```

Figure 6.4: Four different approaches to finding minimum values in Rust closures from the Github *.reduce()* dataset.

```
| a , b | {
    let mut gcd = a;
    let mut remainder = b;
    while remainder != 0 {
        (gcd , remainder) = (remainder , gcd % remainder);
    }
    (a * b) / gcd
}
```

Figure 6.5: A Rust closure implementing least common multiple from the Github *.reduce()* dataset.

## 6.6 Related Work

**User Defined Functions (UDFs)** and their performance are the topic of extensive research [11, 41, 97]. Many databases allow users to tag properties that are satisfied by their UDFs, but none support formal verification that those tags are correct. The role of algebraic properties in query optimization has been explored in [3, 29, 62, 65, 66, 93] and it has been demonstrated that reordering of operations, incrementalization, predicate pushdown, recursive execution, and eventual consistency can all be performed safely under different combinations of algebraic properties. The only research work to explore formally proving properties of database UDFs is by [77] which leverages dataflow analysis to determine whether UDFs satisfy simple properties like never returning a NULL value.

**Formal Verification of Algebraic Properties** has been studied primarily in relation to the verification of conflict-free replicated data types (CRDTs). This was the goal of [53] which utilized SMT solvers to verify commutativity of CRDTs. Propel [113] explores verification of algebraic properties for CRDTs using a custom type-system and [112] extends on Propel by developing a custom verification tool that can prove associativity, commutativity, and idempotence of Propel code written in a purely functional style.

## 6.7 Future Work

We are interested in generalizing Emmy in three dimensions: supporting positive proofs for unbounded data types, generalizing to other UDF programming languages, and generalizing to properties beyond algebraic ones.

**Positive Proofs for Unbounded Data Types**: The two tools supported by Emmy, Cargo Fuzz and Kani, limit the data types for which Emmy can prove properties. In particular, unbounded data types like sets, lists, or strings are not supported today. There are verification tools that can prove properties of unbounded data types, such as Dafny [73] or Verus [71]. The challenge with these tools is that the way they achieve proofs for unbounded data types is via extensive guidance (annotations) by the developer in how to prove the property for the specific input code. This takes the form of things like pre-conditions and post-conditions on parts of the code.

Given our requirements for Emmy that developers never read or write verification code, the ways in which these tools could be used in Emmy are highly restricted. We experimented with the Verus tool for proving the algebraic properties we are interested in with Emmy, but without significant annotation it was not able to prove these properties for our unbounded data type examples. As code generation techniques improve, we hope that these stronger tools can be integrated into Emmy while preserving our 0% false positives and zero-touch verification tooling requirements.

**Beyond Rust**: While the Hydro stack only supports Rust UDFs today, many database systems support UDFs in a wide range of languages. For example, Snowflake supports Python, Java, Javascript, and Scala UDFs [38]. The verification system we have built uses rust-specific libraries, but the methodology is in no way only applicable to Rust. In future work, we want to extend our verification pipeline to be as language-agnostic as possible and support other popular UDF languages such as Python and Java. With moderate engineering effort, our approach could be replicated for another language by applying the relevant fuzz testing and model checking tools for that specific language. In Java, for example, fuzz testing, model checking, and SMT solvers are supported by JQF [91], JBMC [32], and JavaSMT [59] respectively.

**Beyond Algebraic Properties**: From early query optimizers, properties have been used for optimization that theorists would not call "algebraic" properties. One such example is *determinism* of UDFs [51, 82] (the property that the function given the same input will always return the same output). We chose algebraic properties as the focus of Emmy because they are broadly relevant across domains and we found the goal of unifying these overlapping properties from

different domains compelling. Emmy's design does not however restrict its use to only verifying algebraic properties. If a property can be expressed as an *assert* statement in Rust, then it is a single line of code change to support its verification in Emmy. Different properties may be easier or harder for the tools to prove, but we think this generalization is very promising for covering other properties for optimizations of UDFs.

## 6.8 Conclusion

We have introduced Emmy, a library that helps developers reason about algebraic properties in declarative distributed programming. Emmy accomplishes this through the application of verification tools directly on Rust code. The use of verification tools is zero-touch from the perspective of the users, bringing the power of modern verification tools to developers without the usual baggage.

# Chapter 7

# Conclusion

In this dissertation, we have explored the role of algebraic properties in making distributed data systems correct and efficient. In Chapter I, we introduced a new formalism for reasoning about coordination-freeness that bridges the gap between the study of CRDTs in algebra and the study of the CALM Theorem in logic programming. Using this formalism, we are able to talk about the exact guarantees offered by CRDTs and the CALM Theorem. We then extend those guarantees to the natural requirement that query responses to users will remain unchanged regardless of whether a CRDT state has already converged with the property we dub *Free Termination*.

In Chapter II, we introduced an alternative perspective on CRDTs and the semi-lattice data model. In the interest of simplifying the developer experience of working with CRDTs, we observe that the algebraic properties required for CRDT correctness can often be added programatically on the system side, reducing the burden on the developer to design data structures satisfying each property. We explore this programmatic application of *idempotence* and introduce a new protocol called OnceTree that reduces the space overhead of *idempotence upgrades* from linear in the number of replicas to constant.

In Chapter III, we explored the relationship between two major algebraic data models, the semi-lattice model of CRDTs used for eventual consistency and the group and ring data models of incremental view maintenance. We show that despite the algebraic properties of these structures being mathematically incompatible, these data models can co-exist by utilizing semi-lattices at the network layer and groups or rings at the query processing layer, programatically converting the data payloads between the two kinds of structures. This work demonstrates that systems can benefit from both coordination-free consistency and incremental processing of updates all within the framing of algebraic properties of user code.

In Chapter IV, we introduced Emmy, a Rust library for helping developers work with algebraic properties. Emmy utilizes fuzz testing and bounded model checking to prove that algebraic properties are satisfied by the user's Rust code. Emmy is able to do this for bounded data types without any need for developers to translate their code to a spec language or interact with verification tooling in any way.

In each of these explorations, we see how algebra can improve the correctness and efficiency of distributed data systems. We then see how we can ease the burden on developers of building

systems that leverage these algebraic properties. In the spirit of decades of database research and development, we are able to push the knowledge of these properties into the system itself, rather than expose these details to the application programmer.

# Bibliography


[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases*. Vol. 8. Addison-Wesley Reading, 1995.

[2] Serge Abiteboul, Victor Vianu, Brad Fordham, and Yelena Yesha. "Relational transducers for electronic commerce." In: *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. 1998, pp. 179–187.

[3] Mahmoud Abo Khamis, Hung Q Ngo, Reinhard Pichler, Dan Suciu, and Yisu Remy Wang. "Convergence of datalog over (pre-) semirings." In: *Proceedings of the 41st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 2022, pp. 105–117.

[4] Divyakant Agrawal, Amr El Abbadi, Achour Mostéfaoui, Michel Raynal, and Matthieu Roy. "The lord of the rings: Efficient maintenance of views at data warehouses." In: *Distributed Computing: 16th International Conference, DISC 2002 Toulouse, France, October 28–30, 2002 Proceedings 16*. Springer. 2002, pp. 33–47.

[5] Paulo Sérgio Almeida and Carlos Baquero. "Scalable eventually consistent counters over unreliable networks." In: *Distributed Computing* 32.1 (2019), pp. 69–89.

[6] Peter Alvaro, Peter Bailis, Neil Conway, and Joseph M Hellerstein. "Consistency without borders." In: *Proceedings of the 4th annual Symposium on Cloud Computing*. 2013, pp. 1–10.

[7] Peter Alvaro, Neil Conway, Joseph M Hellerstein, and William R Marczak. "Consistency Analysis in Bloom: a CALM and Collected Approach." In: *CIDR*. Citeseer. 2011, pp. 249–260.

[8] Peter Alvaro, William R Marczak, Neil Conway, Joseph M Hellerstein, David Maier, and Russell Sears. "Dedalus: Datalog in time and space." In: *Datalog Reloaded: First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers*. Springer. 2011, pp. 262–281.

[9] Tom J Ameloot, Bas Ketsman, Frank Neven, and Daniel Zinn. "Weaker forms of monotonicity for declarative networking: A more fine-grained answer to the CALM-conjecture." In: *ACM Transactions on Database Systems (TODS)* 40.4 (2015), pp. 1–45.

[10] Tom J Ameloot, Frank Neven, and Jan Van den Bussche. "Relational transducers for declarative networking." In: *Journal of the ACM (JACM)* 60.2 (2013), pp. 1–38.

[11] Samuel Arch, Yuchen Liu, Todd C Mowry, Jignesh M Patel, and Andrew Pavlo. "The Key to Effective UDF Optimization: Before Inlining, First Perform Outlining." In: (2023).

[12] Michael Arntzenius and Neelakantan R Krishnaswami. "Datafun: a functional Datalog." In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. 2016, pp. 214–227.

[13] AWS. *Refreshing a Materialized View*. https://docs.aws.amazon.com/redshift/latest/dg/materialized-view-refresh.html. June 2021.

[14] Tim Baccaert and Bas Ketsman. "Distributed Consistency Beyond Queries." In: *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 2023, pp. 47–58.

[15] Carlos Baquero and Francisco Moura. "Using structural characteristics for autonomous operation." In: *ACM SIGOPS Operating Systems Review* 33.4 (1999), pp. 90–96.

[16] Jim Bauwens and Elisa Gonzalez Boix. "Memory efficient crdts in dynamic environments." In: *Proceedings of the 11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*. 2019, pp. 48–57.

[17] Philip A Bernstein, Meichun Hsu, and Bruce Mann. "Implementing recoverable requests using queues." In: *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*. 1990, pp. 112–122.

[18] Philip A Bernstein and Eric Newcomer. *Principles of transaction processing*. Morgan Kaufmann, 2009.

[19] Garrett Birkhoff. *Lattice Theory*. 2nd ed. Vol. 25. American Mathematical Society, 1948.

[20] *Bourbaki–Witt theorem*. Wikipedia. Available at https://en.wikipedia.org/wiki/Bourbaki%E2%80%93Witt_theorem. URL: https://en.wikipedia.org/wiki/Bourbaki%5C%E2%5C%80%5C%93Witt_theorem (visited on Apr. 27, 2025).

[21] Eric Brewer. "CAP twelve years later: How the" rules" have changed." In: *Computer* 45.2 (2012), pp. 23–29.

[22] Manfred Broy, Martin Wirsing, and Peter Pepper. "On the algebraic definition of programming languages." In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9.1 (1987), pp. 54–99.

[23] Mihai Budiu. *Incremental Database Computations*. https://www.feldera.com/blog/incremental-database-computations/. Jan. 2024.

[24] Mihai Budiu, Tej Chajed, Frank McSherry, Leonid Ryzhyk, and Val Tannen. "DBSP: Automatic Incremental View Maintenance for Rich Query Languages." In: *Proceedings of the VLDB Endowment* 16.7 (2023), pp. 1601–1614.

[25] Bernadette Charron-Bost. "Concerning the size of logical clocks in distributed systems." In: *Information Processing Letters* 39.1 (1991), pp. 11–16.

[26] Rada Chirkova, Jun Yang, et al. "Materialized views." In: *Foundations and Trends® in Databases* 4.4 (2012), pp. 295–405.

[27] Laukik Chitnis, Alin Dobra, and Sanjay Ranka. "Aggregation methods for large-scale sensor networks." In: *ACM Transactions on Sensor Networks (TOSN)* 4.2 (2008), pp. 1–36.

[28] Kevin Clancy and Heather Miller. "Monotonicity types for distributed dataflow." In: *Proceedings of the Programming Models and Languages for Distributed Computing*. 2017, pp. 1–10.

[29] Sara Cohen. "User-defined aggregate functions: bridging theory and practice." In: *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. 2006, pp. 49–60.

[30] Automerge contributors. *automerge/automerge*. https://github.com/automerge/automerge. Apr. 2023.

[31] Neil Conway, William R Marczak, Peter Alvaro, Joseph M Hellerstein, and David Maier. "Logic and lattices for distributed programming." In: *Proceedings of the Third ACM Symposium on Cloud Computing*. 2012, pp. 1–14.

[32] Lucas Cordeiro, Pascal Kesseli, Daniel Kroening, Peter Schrammel, and Marek Trtik. "JBMC: A bounded model checking tool for verifying Java bytecode." In: *International Conference on Computer Aided Verification*. Springer. 2018, pp. 183–190.

[33] Natacha Crooks, Youer Pu, Lorenzo Alvisi, and Allen Clement. "Seeing is believing: A client-centric specification of database isolation." In: *Proceedings of the ACM Symposium on Principles of Distributed Computing*. 2017, pp. 73–82.

[34] Databricks. *Introducing Materialized Views and Streaming Tables for Databricks SQL*. https://www.databricks.com/blog/introducing-materialized-views-and-streaming-tables-databricks-sql. June 2023.

[35] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[36] Frank Dehne and Silvia Gotz. "Practical parallel algorithms for minimum spanning trees." In: *Proceedings Seventeenth IEEE Symposium on Reliable Distributed Systems (Cat. No. 98CB36281)*. IEEE. 1998, pp. 366–371.

[37] Edsger W Dijkstra. "A note on two problems in connexion with graphs." In: *Numerische mathematik* 1.1 (1959), pp. 269–271.

[38] Snowflake Documentation. *User-defined functions overview*. https://docs.snowflake.com/en/developer-guide/udf/udf-overview. 2024.

[39] Stephen Dolan. "Brief announcement: The only undoable crdts are counters." In: *Proceedings of the 39th Symposium on Principles of Distributed Computing*. 2020, pp. 57–58.

[40] Iman Elghandour, Ahmet Kara, Dan Olteanu, and Stijn Vansummeren. "Incremental techniques for large-scale dynamic query processing." In: *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*. 2018, pp. 2297–2298.

[41] Kai Franz et al. "Dear User-Defined Functions, Inlining isn't working out so great for us. Let's try batching to make our relationship work. Sincerely, SQL." In: *CIDR*. Vol. 16. The 49th Annual International Symposium on Computer Architecture, New York … 2024, pp. 617–630.

[42] Craig Gentry. "Fully homomorphic encryption using ideal lattices." In: *Proceedings of the forty-first annual ACM symposium on Theory of computing*. 2009, pp. 169–178.

[43] Abraham Ginzburg. *Algebraic Theory of Automata*. Academic Press, 1968.

[44] Goetz Graefe. "The cascades framework for query optimization." In: *IEEE Data Eng. Bull.* 18.3 (1995), pp. 19–29.

[45] Jim Gray and Andreas Reuter. *Transaction processing: concepts and techniques*. Elsevier, 1992.

[46] 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*. 2007, pp. 31–40.

[47] Peter Hart, Nils Nilsson, and Bertram Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths." In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107. DOI: 10.1109/tssc.1968.300136.

[48] Pat Helland and Dave Campbell. "Building on Quicksand." In: (2009).

[49] Joseph M Hellerstein. "The declarative imperative: experiences and conjectures in distributed logic." In: *ACM SIGMOD Record* 39.1 (2010), pp. 5–19.

[50] Joseph M Hellerstein, Shadaj Laddad, Mae Milano, Conor Power, and Mingwei Samuel. "Initial Steps Toward a Compiler for Distributed Programs." In: *Proceedings of the 5th workshop on Advanced tools, programming languages, and PLatforms for Implementing and Evaluating algorithms for Distributed systems*. 2023, pp. 1–10.

[51] Joseph M Hellerstein and Michael Stonebraker. "Predicate migration: Optimizing queries with expensive predicates." In: *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*. 1993, pp. 267–276.

[52] Joseph M. Hellerstein and Peter Alvaro. "Keeping CALM." In: *Communications of the ACM* 63.9 (Aug. 2020), pp. 72–81. DOI: 10.1145/3369736.

[53] Farzin Houshmand and Mohsen Lesani. "Hamsaz: Replication Coordination Analysis and Synthesis." In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019). DOI: 10.1145/3290387.

[54] Richard Hull and Gang Zhou. "A framework for supporting data integration using the materialized and virtual approaches." In: *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*. 1996, pp. 481–492.

[55] Paulo Jesus, Carlos Baquero, and Paulo Sérgio Almeida. "A survey of distributed data aggregation algorithms." In: *IEEE Communications Surveys & Tutorials* 17.1 (2014), pp. 381–404.

[56] Jr Joseph B. Kruskal. "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem." In: *Proceedings of the American Mathematical Society* 7.1 (1956), pp. 48–50.

[57] Peter Kairouz et al. "Advances and open problems in federated learning." In: *Foundations and Trends® in Machine Learning* 14.1–2 (2021), pp. 1–210.

[58] Ahmet Kara, Hung Q Ngo, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. "Maintaining triangle queries under updates." In: *ACM Transactions on Database Systems (TODS)* 45.3 (2020), pp. 1–46.

[59] Egor George Karpenkov, Karlheinz Friedberger, and Dirk Beyer. "JavaSMT: A unified interface for SMT solvers in Java." In: *Verified Software. Theories, Tools, and Experiments: 8th International Conference, VSTTE 2016, Toronto, ON, Canada, July 17–18, 2016, Revised Selected Papers 8*. Springer. 2016, pp. 139–148.

[60] Jeremy Kepner et al. "Mathematical foundations of the GraphBLAS." In: *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE. 2016, pp. 1–9.

[61] Mahmoud Abo Khamis, Hung Q Ngo, Reinhard Pichler, Dan Suciu, and Yisu Remy Wang. "Datalog in wonderland." In: *ACM SIGMOD Record* 51.2 (2022), pp. 6–17.

[62] Mahmoud Abo Khamis, Hung Q Ngo, Reinhard Pichler, Dan Suciu, and Yisu Remy Wang. "Datalog in wonderland." In: *ACM SIGMOD Record* 51.2 (2022), pp. 6–17.

[63] Martin Kleppmann, Adam Wiggins, Peter Van Hardenberg, and Mark McGranaghan. "Local-first software: you own your data, in spite of the cloud." In: *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 2019, pp. 154–178.

[64] Christoph Koch. "Incremental query evaluation in a ring of databases." In: *Proceedings of the twenty-ninth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 2010, pp. 87–98.

[65] Christoph Koch. "Incremental query evaluation in a ring of databases." In: *Proceedings of the twenty-ninth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 2010, pp. 87–98.

[66] Christoph Koch, Yanif Ahmad, Oliver Kennedy, Milos Nikolic, Andres Nötzli, Daniel Lupei, and Amir Shaikhha. "DBToaster: higher-order delta processing for dynamic, frequently fresh views." In: *The VLDB Journal* 23 (2014), pp. 253–278.

[67] Jay Kreps, Neha Narkhede, Jun Rao, et al. "Kafka: A distributed messaging system for log processing." In: *Proceedings of the NetDB*. Vol. 11. 2011. Athens, Greece. 2011, pp. 1–7.

[68] Lindsey Kuper and Ryan R Newton. "LVars: lattice-based data structures for deterministic parallelism." In: *Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing*. 2013, pp. 71–84.

[69] Shadaj Laddad, Conor Power, Mae Milano, Alvin Cheung, Natacha Crooks, and Joseph M Hellerstein. "Keep CALM and CRDT On." In: *Proceedings of the VLDB Endowment* 16.4 (2022), pp. 856–863.

[70] Shadaj Laddad, Conor Power, Mae Milano, Alvin Cheung, and Joseph M. Hellerstein. *Katara: Synthesizing CRDTs with Verified Lifting*. 2022. DOI: 10.48550/ARXIV.2205.12425.

[71] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. "Verus: Verifying rust programs using linear ghost types." In: *Proceedings of the ACM on Programming Languages* 7.OOPSLA1 (2023), pp. 286–315.

[72] Jessica Laughlin. *Why use a Materialized View?* https://materialize.com/blog/why-use-a-materialized-view/. Aug. 2020.

[73] K Rustan M Leino. "Dafny: An automatic program verifier for functional correctness." In: *International conference on logic for programming artificial intelligence and reasoning*. Springer. 2010, pp. 348–370.

[74] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. "Scaling distributed machine learning with the parameter server." In: *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 2014, pp. 583–598.

[75] Albert van der Linde, João Leitão, and Nuno Preguiça. "$\delta$-crdts: Making $\delta$-crdts delta-based." In: *Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data*. 2016, pp. 1–4.

[76] Xinyu Liu, Joy Arulraj, and Alessandro Orso. "A Framework for Inferring Properties of User-Defined Functions." In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 2024, pp. 1–11.

[77] Xinyu Liu, Joy Arulraj, and Alessandro Orso. "A Framework for Inferring Properties of User-Defined Functions." In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 2024, pp. 1–11.

[78] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E Gay, Joseph M Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. "Declarative networking." In: *Communications of the ACM* 52.11 (2009), pp. 87–95.

[79] Samuel Madden, Michael J Franklin, Joseph M Hellerstein, and Wei Hong. "{TAG}: A Tiny {AGgregation} Service for {Ad-Hoc} Sensor Networks." In: *5th Symposium on Operating Systems Design and Implementation (OSDI 02)*. 2002.

[80] Christopher Meiklejohn. *Applied Monotonicity: A Brief History of CRDTs in Riak*. https://christophermeiklejohn.com/erlang/lasp/2019/03/08/monotonicity.html. Mar. 2019.

[81] Christopher Meiklejohn and Peter Van Roy. "Lasp: A language for distributed, coordination-free programming." In: *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*. 2015, pp. 184–195.

[82] Microsoft. *Deterministic and Nondeterministic Functions*. SQL Server 2022. Microsoft. 2023. URL: https://learn.microsoft.com/en-us/sql/relational-databases/user-defined-functions/deterministic-and-nondeterministic-functions?view=sql-server-ver16 (visited on May 6, 2025).

[83] Mae Milano, Rolph Recto, Tom Magrino, and Andrew C Myers. "A tour of gallifrey, a language for geodistributed programming." In: *3rd Summit on Advances in Programming Languages (SNAPL 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2019.

[84] Jeffery C Mogul. "Clarifying the fundamentals of HTTP." In: *Proceedings of the 11th international conference on World Wide Web*. 2002, pp. 25–36.

[85] Derek G Murray, Frank McSherry, Michael Isard, Rebecca Isaacs, Paul Barham, and Martin Abadi. "Incremental, iterative data processing with timely dataflow." In: *Communications of the ACM* 59.10 (2016), pp. 75–83.

[86] Suman Nath, Phillip B Gibbons, Srinivasan Seshan, and Zachary Anderson. "Synopsis diffusion for robust aggregation in sensor networks." In: *ACM Transactions on Sensor Networks (TOSN)* 4.2 (2008), pp. 1–40.

[87] Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klamma. "Yjs: A Framework for Near Real-Time P2P Shared Editing on Arbitrary Data Types." In: *Proceedings of the 15th International Conference on Engineering the Web in the Big Data Era - Volume 9114*. ICWE 2015. Rotterdam, The Netherlands: Springer-Verlag, 2015, pp. 675–678. DOI: 10.1007/978-3-319-19890-3_55.

[88] Milos Nikolic, Haozhe Zhang, Ahmet Kara, and Dan Olteanu. "F-IVM: learning over fast-evolving relational data." In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2020, pp. 2773–2776.

[89] Feng Niu, Benjamin Recht, Christopher Re, and Stephen J. Wright. "HOGWILD! A Lock-Free Approach to Parallelizing Stochastic Gradient Descent." In: *Proceedings of the 24th International Conference on Neural Information Processing Systems*. NIPS'11. Granada, Spain: Curran Associates Inc., 2011, pp. 693–701.

[90] Ty Overby. *bincode-org/bincode*. https://github.com/bincode-org/bincode. Apr. 2023.

[91] Rohan Padhye, Caroline Lemieux, and Koushik Sen. "Jqf: Coverage-guided property-based testing in java." In: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2019, pp. 398–401.

[92] Radia Perlman. "An algorithm for distributed computation of a spanningtree in an extended lan." In: *ACM SIGCOMM computer communication review* 15.4 (1985), pp. 44–53.

[93] Conor Power, Saikrishna Achalla, Ryan Cottone, Nathaniel Macasaet, and Joseph M Hellerstein. "Wrapping Rings in Lattices: An Algebraic Symbiosis of Incremental View Maintenance and Eventual Consistency." In: *Proceedings of the 11th Workshop on Principles and Practice of Consistency for Distributed Data*. 2024, pp. 15–22.

[94] Nuno Preguiça. "Conflict-free Replicated Data Types: An Overview." In: *arXiv e-prints* (2018), arXiv–1806.

[95] Nuno Preguiça, Carlos Baquero, Paulo Sérgio Almeida, Victor Fonte, and Ricardo Gonçalves. "Dotted version vectors: Logical clocks for optimistic replication." In: *arXiv preprint arXiv:1011.5808* (2010).

[96] Nuno Preguiça, Carlos Baquero, and Marc Shapiro. "Conflict-free replicated data types (CRDTs)." In: *arXiv preprint arXiv:1805.06358* (2018).

[97] Karthik Ramachandra, Kwanghyun Park, K Venkatesh Emani, Alan Halverson, César Galindo-Legaria, and Conor Cunningham. "Froid: Optimization of imperative programs in a relational database." In: *Proceedings of the VLDB Endowment* 11.4 (2017), pp. 432–444.

[98] Ronald L Rivest, Adi Shamir, and Leonard Adleman. "A method for obtaining digital signatures and public-key cryptosystems." In: *Communications of the ACM* 21.2 (1978), pp. 120–126.

[99] Antony Rowstron and Peter Druschel. "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems." In: *Middleware 2001: IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, Germany, November 12–16, 2001 Proceedings 2*. Springer. 2001, pp. 329–350.

[100] Rust Fuzz Team. *Fuzzing with cargo-fuzz*. https://rust-fuzz.github.io/book/cargo-fuzz.html. Accessed: 2025-05-13. 2025. URL: https://rust-fuzz.github.io/book/cargo-fuzz.html.

[101] Mingwei Samuel, Joseph M Hellerstein, and Alvin Cheung. "Hydroflow: A Model and Runtime for Distributed Systems Programming." In: (2021).

[102] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. "A comprehensive study of convergent and commutative replicated data types." In: *Technical Report, Inria–Centre Paris-Rocquencourt; INRIA*. 2011.

[103] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. "Conflict-free replicated data types." In: *Symposium on Self-Stabilizing Systems*. Springer. 2011, pp. 386–400.

[104] Michael Sipser. "Introduction to the Theory of Computation." In: *ACM Sigact News* 27.1 (1996), pp. 27–29.

[105] Sindhu Subhas. *Hive - Materialized Views*. https://techcommunity.microsoft.com/t5/analytics-on-azure-blog/hive-materialized-views/ba-p/2502785. June 2021.

[106] Douglas B Terry, Marvin M Theimer, Karin Petersen, Alan J Demers, Mike J Spreitzer, and Carl H Hauser. "Managing update conflicts in Bayou, a weakly connected replicated storage system." In: *ACM SIGOPS Operating Systems Review* 29.5 (1995), pp. 172–182.

[107] Alexa VanHattum, Daniel Schwartz-Narbonne, Nathan Chong, and Adrian Sampson. "Verifying dynamic trait objects in Rust." In: *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*. 2022, pp. 321–330.

[108] Joost Verbraeken, Matthijs Wolting, Jonathan Katzy, Jeroen Kloppenburg, Tim Verbelen, and Jan S Rellermeyer. "A survey on distributed machine learning." In: *Acm computing surveys (csur)* 53.2 (2020), pp. 1–33.

[109] Stephane Weiss, Pascal Urso, and Pascal Molli. "Logoot: A Scalable Optimistic Replication Algorithm for Collaborative Editing on P2P Networks." In: *2009 29th IEEE International Conference on Distributed Computing Systems*. 2009, pp. 404–412.

[110] Chenggang Wu, Jose M Faleiro, Yihan Lin, and Joseph M Hellerstein. "Anna: A kvs for any scale." In: *IEEE Transactions on Knowledge and Data Engineering* 33.2 (2019), pp. 344–358.

[111] Chenggang Wu, Vikram Sreekanti, and Joseph M Hellerstein. "Autoscaling tiered cloud storage in Anna." In: *Proceedings of the VLDB Endowment* 12.6 (2019), pp. 624–638.

[112] George Zakhour, Pascal Weisenburger, and Guido Salvaneschi. "Automated Verification of Fundamental Algebraic Laws." In: *Proceedings of the ACM on Programming Languages* 8.PLDI (2024), pp. 766–789.

[113] George Zakhour, Pascal Weisenburger, and Guido Salvaneschi. "Automated verification of fundamental algebraic laws." In: *Proceedings of the ACM on Programming Languages* 8.PLDI (2024), pp. 766–789.

[114] Ben Y Zhao, Ling Huang, Jeremy Stribling, Sean C Rhea, Anthony D Joseph, and John D Kubiatowicz. "Tapestry: A resilient global-scale overlay for service deployment." In: *IEEE Journal on selected areas in communications* 22.1 (2004), pp. 41–53.

[115] Yue Zhuge, Hector Garcia-Molina, Joachim Hammer, and Jennifer Widom. "View maintenance in a warehousing environment." In: *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*. 1995, pp. 316–327.

[116] Yue Zhuge, Hector Garcia-Molina, and Janet L Wiener. "The strobe algorithms for multi-source warehouse consistency." In: *Fourth International Conference on Parallel and Distributed Information Systems*. IEEE. 1996, pp. 146–157.

[117] Ling Zhuo and Viktor K Prasanna. "High performance linear algebra operations on reconfigurable systems." In: *SC'05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. IEEE. 2005, pp. 2–2.

[118] Daniel Zinn, Todd J Green, and Bertram Ludäscher. "Win-move is coordination-free (sometimes)." In: *Proceedings of the 15th International Conference on Database Theory*. 2012, pp. 99–113.