Language Support for Loosely Consistent Distributed Programming

by

Neil Robert George Conway

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
Professor Tapan S. Parikh
Professor Rastislav Bodík

Fall 2014

Language Support for Loosely Consistent Distributed Programming

Abstract

Language Support for Loosely Consistent Distributed Programming

by

Neil Robert George Conway

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Joseph M. Hellerstein, Chair

Driven by the widespread adoption of both cloud computing and mobile devices, distributed computing is increasingly commonplace. As a result, a growing proportion of developers must tackle the complexity of distributed programming—that is, they must ensure correct application behavior in the face of asynchrony, concurrency, and partial failure.

To help address these difficulties, developers have traditionally relied upon system infrastructure that provides *strong consistency* guarantees (e.g., consensus protocols and distributed transactions). These mechanisms hide much of the complexity of distributed computing—for example, by allowing programmers to assume that all nodes observe the same set of events in the same order. Unfortunately, providing such strong guarantees becomes increasingly expensive as the scale of the system grows, resulting in availability and latency costs that are unacceptable for many modern applications.

Hence, many developers have explored building applications that only require *loose consistency* guarantees—for example, storage systems that only guarantee that all replicas eventually converge to the same state, meaning that a replica might exhibit an arbitrary state at any particular time. Adopting loose consistency involves making a well-known tradeoff: developers can avoid paying the latency and availability costs incurred by mechanisms for achieving strong consistency, but in exchange they must deal with the full complexity of distributed computing. As a result, achieving correct application behavior in this environment is very difficult.

This thesis explores how to aid developers of loosely consistent applications by providing *programming language support* for the difficulties they face. The language level is a natural place to tackle this problem: because developers that use loose consistency have fewer system facilities that they can depend on, consistency concerns are naturally pushed into application logic. In part, our goal has been to recognize, formalize, and automate application-level consistency patterns.

We describe three language variants that each tackle a different challenge in distributed programming. Each variant is a modification of *Bloom*, a declarative language for distributed programming we have developed at UC Berkeley. The first variant of Bloom, *Bloom*^{L}, enables deterministic distributed programming without the need for distributed coordination. Second, *Edelweiss* allows distributed storage reclamation protocols to be generated in a safe and automatic fashion. Finally,

Bloom^{PO} adds sophisticated ordering constraints that we use to develop a declarative, high-level implementation of concurrent editing, a particularly difficult class of loosely consistent programs.

Contents

List of Figures

List of Source Code Listings

List of Tables

Acknowledgments

First and foremost, I would like to thank my advisor, Joe Hellerstein, for his tireless support and thoughtful council throughout my time in graduate school. The ideas in this thesis were developed in close collaboration with Joe—but more importantly, Joe taught me how to be a researcher. If your experience in graduate school is largely defined by your relationship with your advisor, then I have been very fortunate indeed.

I would also like to thank my thesis committee for their helpful feedback: Ras and Tap provided outside perspectives on this work that made the final result much stronger. Michael Franklin provided very useful suggestions as the chair of my qualifying exam committee. I am particularly indebted to Mike for giving me the opportunity to join his startup company as an intern in 2006. My experience working at Truviso with Mike and Sailesh Krishnamurthy was a major factor in my decision to apply to graduate school—I wanted to be like them.

I had the privilege of working with David Maier of Portland State University. Professor Maier made major contributions to Bloom^L, particularly in establishing how Bloom^L relates to other efforts to extend logic programming to lattice structures. Moreover, his careful and insightful feedback on my work has been very valuable.

I have been fortunate to work with a group of brilliant and generous colleagues at UC Berkeley. Much of this thesis is the result of joint work with the other members of the BOOM group: Peter Alvaro, Peter Bailis, Bill Marczak, and Josh Rosen. In particular, Bill Marczak first observed the similarity between monotonic logic programming and join semilattices, which we subsequently developed into Bloom^L. Peter Alvaro implemented atomic read/write registers in Edelweiss (Section 4.6)—but more importantly, he was deeply involved in discussions on many topics related to this thesis. Emily Andrews performed the Edelweiss experiments (Section 4.7) and also contributed to DiCE.

When I began graduate school, the senior students in the Berkeley Database Group helped me to immediately feel welcome. I learned to do systems research in part by following the high standard set by Tyson Condie and Rusty Sears on the BOOM Analytics project. I owe Kuang Chen and Beth Trushkowsky a debt of gratitude for passing the database prelim exam—the countless hours we spent doing practice questions together was invaluable.

Amanda Kattan has been a great source of support, a boundless fount of enthusiasm, and a dedicated copy editor. I am incredibly happy to have met her.

Finally, many individuals have offered feedback and advice on the ideas in this thesis. My thanks to Martin Abadi, Daniel Bruckner, Mosharaf Chowdhury, Tyson Condie, Sean Cribbs, Alan Downing, Alan Fekete, Ali Ghodsi, Todd Green, Andrew R. Gross, Haryadi Gunawi, Coda Hale, Pat Helland, Alan Kay, Andy Konwinski, Tim Kraska, Lindsey Kuper, Jonathan Harper, Erik Meijer, Leo Meyerovich, Christopher Meiklejohn, Pascal Molli, Rusty Sears, Marc Shapiro, Mehul Shah, Evan R. Sparks, Matei Zaharia, and Daniel Zinn.

My research was supported in part by the Natural Sciences and Engineering Research Council of Canada, and gifts from EMC, Microsoft Research, NTT Multimedia Laboratories, and Yahoo!.

Chapter 1

Introduction

In recent years, distributed programming has moved from the sidelines of computing into the mainstream. Driven by the ubiquity of both cloud computing and mobile devices, the vast majority of new, non-trivial programs in the future will involve communication and distributed computation. Hence, the vast majority of programmers will need to face the challenges of distributed programming.

Building reliable distributed programs is not fundamentally different from building reliable centralized programs: in both cases, the programmer must ensure that the application behaves correctly (e.g., satisfies its correctness invariants) for all possible inputs [7]. However, programming in a distributed environment raises additional challenges: due to the presence of asynchrony, concurrency, and partial failures, the programmer must ensure that their program behaves correctly for *all possible schedules* of network messages and node failures. This significantly raises the level of difficulty, particularly for programmers who are not experts in distributed computing.

To simplify the development of distributed systems, programmers have traditionally relied upon storage and messaging infrastructure that provides *strong consistency* guarantees. A variety of mechanisms have been proposed for achieving different flavors of strong consistency, including atomic broadcast [51], consensus [97], distributed ACID transactions [68], group communication systems [29], and distributed locking services [36, 140]; many of these proposals have been widely used in practice. These mechanisms simplify the development of distributed systems because they present useful, high-level abstractions that hide much of the inherent complexity of distributed programming. For example, *serializability* [121] and *linearizability* [83] are two strong consistency guarantees that have seen widespread adoption: by ensuring that all replicas in a system observe the same events in the same order, systems that provide one of these properties make it easier for programmers to build reliable applications.

The Costs of Strong Consistency. Despite the historical success and rich theory underpinning systems infrastructure for strong consistency, an increasing number of practitioners have chosen to reduce or avoid relying on strong consistent in many situations [28, 50, 78, 79]. The core problem is that providing strong consistency guarantees becomes increasingly difficult as the scale of the system grows. Applications that rely upon strong consistency guarantees must pay three different costs: *availability* in the presence of network partitions, *latency*, and *operational complexity*.

The difficulty of providing strong consistency guarantees in the presence of network failures (e.g., partitions) has been known since at least the early 1980s [48, 49], but the relationship between consistency and availability only received widespread attention with the publication of Brewer's well-known CAP Theorem [30, 31, 65]. The CAP Theorem states that a distributed system cannot simultaneously provide consistency (i.e., linearizability), availability, and partition tolerance. For most applications, sacrificing partition tolerance is not an option [1, 31, 73], thereby forcing developers to choose between availability and strong consistency. For many recent systems, even brief periods of downtime cause severe financial repercussions [39, 50, 149]. As a result, many modern system designers are reluctant to accept decreased availability in exchange for the programming conveniences of strong consistency.

Mechanisms for strong consistency also increase response latency: to prevent conflicts between concurrent updates to different replicas, each update must contact at least one other site [1]. Although the latency of communication within a single data center is typically small, many modern distributed systems are *geo-replicated*: they consist of nodes at multiple geographically dispersed data centers [21, 43, 45, 91, 103, 141]. Communication between data centers is usually expensive (e.g., hundreds of milliseconds) [91]. For many applications, waiting for even a single round-trip between data centers before applying an update incurs an unacceptable latency cost: indeed, one of the primary reasons for adopting geo-replication in the first place is to allow client operations to be handled entirely by a single data center that is geographically close to the client's location [104].[1]

Some practitioners have also observed that traditional mechanisms for providing strong consistency make data center operations more fragile and error-prone. Because strong consistency techniques require *blocking* (e.g., to wait for a commit/abort decision or for a distributed lock manager to grant a lock request) and increase the degree of coupling between operations, they increase system fragility and can lead to "convoy effects", "self-synchronization", and "chaotic load fluctuations" [28]. As a result, James Hamilton has argued [28]:

> The first principle of successful scalability is to batter the consistency mechanisms down to a minimum, move them off the critical path, hide them in a rarely visited corner of the system, and then make it as hard as possible for application developers to get permission to use them.

The Challenges of Loose Consistency. In response to these concerns, many system designers have chosen to reduce or entirely avoid the use of strongly consistent systems infrastructure. Instead, many modern designs rely on only weak guarantees from distributed storage—for example, application logic might only assume that all the replicas of an object *eventually converge* to the same state (often called "eventual consistency") [19, 147, 152]. Amazon's Dynamo is among the most influential of these designs [50], although many similar systems have been built (e.g., [23, 94, 126]).

These systems avoid the availability and latency costs of strong consistency by allowing conflicting updates to be accepted by different replicas at the same time. Since the conflicting updates

[1] Note that mechanisms for providing strong consistency do not necessarily decrease *throughput*. If the application can be structured such that operations are batched together or many independent operations can be issued in parallel, high rates of throughput can still be achieved [21, 45, 74].

might occur on different sides of a network partition, avoiding the conflict (as in strong consistency) would require waiting for the network partition to heal—sacrificing availability in the mean time. Instead, eventually consistent systems accept both updates; changes are replicated between sites asynchronously, and a conflict resolution procedure is used to allow replicas that have accepted conflicting updates to converge to the same state. Some systems use a fixed rule to decide which update should be preferred (e.g., last-writer wins [88]), whereas others allow a user-defined merge procedure to be employed [50, 147]. Such merge procedures can exploit application semantics to allow most conflicts to be resolved harmoniously; for example, if the updates represent additions to a user's shopping cart, application semantics might allow the set union of all concurrent updates to be accepted. When conflicts cannot be automatically resolved (e.g., two edits to a document that make conflicting changes to the same text), systems often save aside both versions of the document for manual resolution by a user or administrator [89].

Write-write conflicts are a special case of a more general class of problems that must be handled by applications that rely on eventually consistent storage. Recall that eventual consistency only guarantees that all replicas of an object will "eventually" converge to the same state; no guarantee is made about when that convergence will occur, or what particular state the object will converge to.[2] In the mean time, applications might read arbitrarily old versions of an object. For example, suppose an application issues two read operations against the same distributed object, one after another. There is no guarantee that the second read will return a "newer" object version than the first read; in fact, the two reads could return versions that reflect completely disjoint sets of write operations. The problem is exacerbated if the application then issues its own write operations that use values derived from the reads: while the storage will guarantee that all values eventually converge, ensuring that the system converges to a set of values that respect the application's correctness properties is beyond the scope of the guarantees provided by the storage system [7, 42].

Design Patterns for Application-Level Consistency. Eschewing strong consistency can potentially improve latency and availability, but it makes the task of the application developer much more difficult. To ensure that an application meets its correctness requirements, the developer must reason about many possible combinations of message reorderings and partial failures. For each piece of distributed state, the developer must consider which correctness properties need to be maintained, how those properties might be violated by asynchrony, and how application logic can be written to ensure the violation is either tolerated, compensated for, or prevented from occurring. As a result, ensuring consistency is no longer primarily the province of systems infrastructure; rather, it becomes an *application-level* responsibility [7].

Moving the locus of responsibility for ensuring consistency from systems infrastructure to application developers risks makes it significantly harder to write correct applications; indeed, strong consistency mechanisms have seen widespread adoption in large part because they present a simple abstraction that makes reliable distributed programming simpler. To aid developers tasked with achieving application-level consistency on top of loosely consistent infrastructure, several authors have identified design patterns and "rules of thumb" that have proven useful in practice [31, 46,

[2] For example, using "last-writer wins" as a merge procedure results in the system dropping conflicting writes with earlier timestamps, in many cases without user notification.

57, 59, 77, 78, 79, 80, 111, 112, 114, 118, 125]. For example, the "ACID 2.0" pattern suggests that application-level operations should be made Associative, Commutative, and Idempotent whenever possible [125]. These properties help to ensure that application behavior will be robust in the face of batching, reordering, and duplication of network messages. Another notable design pattern is eschewing mutable shared state whenever possible [46, 59, 77, 111]: if each node only accumulates and exchanges immutable messages, many traditional concerns about distributed consistency can be sidestepped.

However, informal patterns such as these are far from a complete solution to the difficulties raised by loose consistency. Because they are described informally, design patterns cannot be verified by a compiler, integrated with debugging tools, or encoded into systems infrastructure that can be reused by multiple applications. Without integration into programming languages or frameworks, developers need to understand the patterns and decide how best to apply them, which requires experience and expertise. Underlying these practical concerns are some fundamental questions that design patterns leave unanswered. For example, are there inherent limitations on the kinds of application logic that can be expressed using commutative, associative, and idempotent operations? That is, are there certain kinds of application logic that *require* inter-site coordination and strong consistency, or are both families of techniques equally expressive?

In summary, design patterns are a useful tool but further work is needed to simplify the development of loosely consistent distributed programs.

A Language-Centric Approach. The difficulty of programming against eventually consistent systems infrastructure has been widely recognized. The research community has responded by proposing a variety of consistency protocols that preserve availability in the face of network partitions but offer stronger guarantees than eventual consistency, such as causal consistency [4], highly available transactions [17], and red-blue consistency [98]. Researchers have also built prototype systems that implement different variations of these guarantees, such as COPS [103], Eiger [104], and Gemini [98].

While defining new consistency guarantees and realizing those guarantees in systems infrastructure is certainly useful, it leaves important questions unanswered: for example, how should application semantics be matched to the semantics offered by the distributed storage layer? Can application logic achieve its correctness requirements with eventual consistency, or is a stronger level of consistency required? If so, which of the many guarantees should be adopted? If one operation requires a particular consistency level, what happens if the operation interacts with data produced by another operation, which might be content with a lower level of consistency?

In short, programming with loose consistency requires understanding how application semantics is influenced by asynchronous network communication and partial failure. To answer these questions, new systems infrastructure is not sufficient by itself—as argued above, programming with loose consistency implies that consistency concerns are no longer primarily the responsibility of systems infrastructure [7]. Rather, mechanisms for dealing with loose consistency are found throughout the software stack, and often intimately depend on application semantics. These semantics are typically encoded in the application logic itself, which is written in an application-level programming language. Thus, we choose to focus on *language support*: that is, we show how novel programming

language features can be introduced to address widely recognized challenges in loosely consistent programming.

Language Support for Loose Consistency. In this thesis, we explore a variety of techniques that simplify the development of loosely consistent distributed programs. In particular, this thesis draws upon earlier work at UC Berkeley on *declarative networking*, which used deductive database languages such as Datalog to build network protocols and distributed algorithms. As part of this thesis, we designed a new declarative language for distributed computing called *Bloom*; Chapter 2 summarizes this language and the history of its development.

In the subsequent chapters of this thesis, we develop three variants of Bloom that address different challenges raised by loosely consistent distributed programming. In many cases, these extensions draw upon widely observed design patterns, which we have then formalized and built into programming languages. In particular, Chapter 3 introduces Bloom^L, an extension to Bloom that supports deterministic distributed programming through the use of join semilattices and monotone functions. This formalizes the "ACID 2.0" design pattern of employing associative, commutative, and idempotent operations. In Chapter 4, we introduce Edelweiss, a sublanguage of Bloom designed for programs that accumulate sets of immutable messages. We show how Edelweiss allows distributed garbage collection schemes for such programs to be generated automatically. Finally, Chapter 5 turns to a particularly challenging case study of loosely consistent programming: *concurrent editing*, in which users at different sites edit a shared document, and their edits are later reconciled. We introduce a Bloom extension called Bloom^{PO}, which allows partial order constraints (such as causal order) to be specified in the program text and respected by the language runtime. We then use Bloom^{PO} to build DiCE, a simple and declarative solution to the concurrent editing problem.

Chapter 2

Background: Bloom

Bloom is a declarative language for distributed computing that we have designed at UC Berkeley. In this chapter, we summarize our motivations and goals when designing Bloom, and then review the basic elements of the language. In the remainder of this thesis, we introduce several extensions to Bloom.

2.1 History

Our work on Bloom began in the context of the *Declarative Networking* project, which ran from 2003–2009 at UC Berkeley [105]. The goal of that project was to investigate how declarative languages could be used to implement a variety of network protocols and applications, such as distributed crawlers [44, 107], routing protocols [108], overlay networks such as the Chord distributed hash table [106], and sensor networks [38]. This work also resulted in the design of several Datalog variants for expressing distributed programs, including NDLog [105], Overlog [40], and SNLog [38].

Our work on Bloom was initially motivated by two goals. First, we wanted to shift our focus from network protocols in the narrow to a broader class of distributed systems. Second, we wanted to design a new language that could make declarative distributed programming an attractive option for the general practitioner audience. We felt that the time was ripe for a new programming language focused on distributed computing: the widespread adoption of cloud computing meant that many developers were writing distributed systems, and yet traditional languages were a poor fit for the challenges encountered building such designs.

Although our eventual goal was to design a new programming language, we decided to begin by using an existing declarative language to build several relatively complicated distributed programs, and then to use that experience to guide our subsequent language design efforts. Hence, we used Overlog to build a Hadoop-compatible distributed file system and MapReduce job scheduler [8]. As part of this work, we also implemented two-phase commit and the Paxos consensus protocol in Overlog [9].

Our experience using Overlog to build these systems was generally positive: for example, the

distributed file system we built achieved performance within 20% of HDFS but was implemented with an order of magnitude fewer lines of code. We were also able to rapidly evolve our initial design, adding complex features such as high availability and horizontal partitioning with relatively minor revisions to the code. Nevertheless, we also ran into several challenges that informed our later work on Bloom:

Semantics of mutation: Many of our bugs were caused by confusion about exactly when state update rules would "fire," and when the consequences of the state update would be made visible to other rules. Indeed, the semantics of updates in Overlog was never formalized, which meant that the "correct" program semantics was essentially defined by the behavior of the Overlog runtime. Naturally, this was problematic: understanding the semantics of state update in Overlog required the programmer to have a mental model of how the Overlog runtime was implemented. Moreover, improvements to the implementation of the Overlog runtime caused the behavior of our programs to change in unpredictable ways. On reflection, it is not surprising that state update was a common source of problems: traditionally, Datalog is defined over a static database and hence cannot model an evolving database.

Resolution: To ensure that state update in Bloom had a clear semantics, we began by designing Dedalus [6, 13], a "precursor" language to Bloom in which all facts are given explicit timestamps, and each rule specifies the temporal relationship between body atoms and the rule head. In this way, the precise time at which a data value is introduced, replaced, or removed is made explicit. This change resolved the semantic ambiguities that plagued earlier languages like Overlog. Dedalus provides three temporal operators: a rule can describe what is true at the current timestamp, what is true or no-longer-true at the immediately following timestamp, or what will become true at some non-deterministic future timestamp. These alternatives model local deduction, state update, and asynchronous communication, respectively. In Section 2.2, we will review these ideas in the context of Bloom, which borrows the same set of temporal operators used in Dedalus.

Avoidance of implicit communication: In Overlog, the body of a rule can reference data values located at multiple nodes. The Overlog runtime is responsible for doing the communication necessary to send values between nodes to perform the distributed join. When building systems infrastructure in Overlog, we carefully avoided using such rules because we found it difficult to reason about their behavior in the presence of system failures and network partitions—a crucial concern when designing distributed file systems and consensus protocols. For example, if a rule implies that data should be sent from host Y to host X but X has not received the expected data, X has a relatively straightforward set of error handling strategies (e.g., after a timeout, X can either ask Y to resend the message or declare Y failed and then take the appropriate recovery actions. Whereas if X receives the output of a join between n nodes, the space of possible error handling strategies is much more complicated, and indeed may depend on the particular join evaluation strategy chosen by the Overlog runtime.

This Overlog feature grew out of the conceit that an Overlog program conceptually manages a single "global" database, which happens to be partitioned among a set of nodes. Reasoning about global database state is appropriate (and convenient!) in certain domains: for example, users of

large-scale analytic database systems write SQL queries against a global schema, and the DBMS automatically moves data and computation around as needed to evaluate the query; node and network failures are handled transparently and automatically [16]. In contrast, when writing distributed infrastructure, the programmer typically wants to carefully control where a particular data value is stored and to write custom logic for detecting and responding to failure events. For example, we envisioned using Bloom to write programs such as a distributed DBMS with parallel query execution and fault tolerance; hence, managing communication and dealing with failure should be the responsibility of the application programmer, not the language runtime.[1] In some ways, this decision echoes the well-known advice of Waldo et al., who argued that attempting to present the same interface for both local and remote operations is often unwise [153].

Resolution: We restricted both Dedalus and Bloom to require that each rule body only accesses data that is located at a single node. Note that the location of the rule body and the rule head might differ; if that is the case, the rule describes asynchronous communication between the body location and the head location. Such "communication rules" are explicitly identified in the syntax of both Dedalus and Bloom, which allows programmers to easily spot the sources of asynchrony in their programs. Because of this restriction, each Bloom node only does local query evaluation: it evaluates its rule set over its local database content, optionally sending some of the resulting derived values to other nodes. As suggested above, in some sense this means that Bloom is "lower level" than Overlog or a parallel dialect of SQL, in which communication and data movement is implicit. Because of our focus on building systems infrastructure, we felt that programmers needed precise control over this aspect of their programs.

Cryptic syntax: Overlog uses the traditional Datalog rule syntax, in which joins are expressed via unification. We found that this was a frequent obstacle when presenting Overlog programs to practitioners and most academics. Moreover, we observed that relatively few of our programs used recursion or many-way joins (for which the Datalog syntax is convenient). Most of the distributed programs we studied involve relatively simple operations over data sets. For example, a typical server program might join a set of inbound messages with local state and use projection to produce a set of outbound response messages. Another problematic aspect of Overlog was the fact that it defined its own type system. This required additional work by the programmer to convert values to and from the host language's type system.

Resolution: Bloom eschews the traditional Datalog rule syntax in favor of a syntax designed to be more familiar to imperative and functional programmers. We also decided to implement Bloom as a *domain-specific language* (DSL) embedded inside a *host language*; this had several benefits. First, it allowed us to reuse the host language's type system—avoiding the need to build a type system ourselves, and easing integration with other host language code. Second, most

[1] Our initial goal with Bloom was to move from targeting network protocols to a broader class of distributed systems. On reflection, our focus remained somewhat narrow: most of our example programs involved distributed storage systems, coordination protocols, and other kinds of systems infrastructure. Our decision to eschew the global database abstraction reflects this focus: attempting to mask node failures and introduce location transparency would be ill-advised when building systems infrastructure, whereas a language focused on application-level distributed programming might choose to present a higher-level abstraction that hides these concerns.

modern languages include good support for manipulating collections (e.g., `map`, `filter`, and `fold` operations). Reusing this syntax for Bloom rules made the resulting programs more familiar to programmers with prior knowledge of the host language. We built Bloom prototypes that used several different host languages, including Erlang, Javascript, and Scala, but most of our engineering effort was devoted to the Ruby-based variant of Bloom, which we call *Bud*.

2.2 Language Constructs

Bloom programs are bundles of declarative *statements* (rules) about collections of *facts* (tuples). An *instance* of a Bloom program performs computation by evaluating its statements over the contents of its local collections. Instances communicate via asynchronous message passing.

An instance of a Bloom program proceeds through a series of *timesteps*, each containing three phases.[2] In the first phase, inbound events (e.g., network messages) are received and represented as facts in collections. In the second phase, the program's statements are evaluated to compute all the additional facts that can be derived from the instance's local collections. In some cases (described below), a derived fact is intended to achieve a "side effect," such as modifying local state or sending a network message. These effects are deferred during the second phase of the timestep; the third phase is devoted to carrying them out.

In the remainder of this thesis, we show Bloom code written for Bud, the Ruby-based Bloom implementation. Listing 2.1 shows a Bloom program represented as an annotated Ruby class. A small amount of Ruby code is needed to instantiate the Bloom program and begin executing it, for example:

```
# Create a new instance of the shortest paths Bloom program (standard Ruby instanciation syntax).
sp = ShortestPaths.new
# Cause that instance to begin executing asynchronously (sending and receiving network messages).
sp.run_bg
```

Data Model

In Bloom, a collection is an unordered set of *facts*, akin to a relation in Datalog. The Bud prototype adopts the Ruby type system rather than inventing its own; hence, a fact in Bud is just an array of immutable Ruby objects. Each collection has a *schema*, which declares the structure (column names) of the facts in the collection. A subset of the columns in a collection form its *key*: as in the relational model, the key columns functionally determine the remaining columns. The collections used by a Bloom program are declared in a `state` block. For example, line 5 of Listing 2.1 declares a collection named `link` with three columns, two of which form the collection's key. Ruby is a dynamically typed language, so keys and values in Bud can hold arbitrary Ruby objects.

[2] There is a declarative semantics for Bloom [6, 13], but for the sake of exposition we describe the language operationally here.

```
1  class ShortestPaths
2    include Bud

4    state do
5      table :link, [:from, :to] => [:cost]
6      scratch :path, [:from, :to, :next_hop, :cost]
7      scratch :min_cost, [:from, :to] => [:cost]
8    end

10   bloom do
11     path <= link {|l| [l.from, l.to, l.to, l.cost]}
12     path <= (link * path).pairs(:to => :from) {|l,p| [l.from, p.to, l.to, l.cost + p.cost]}
13     min_cost <= path.group([:from, :to], min(:cost))
14   end
15 end
```

Listing 2.1: All-pairs shortest paths in Bloom.

Bloom provides several collection types to represent different kinds of state (Table 2.1). A `table` stores persistent data: if a fact appears in a table, it remains in the table in future timesteps (unless it is explicitly removed). A `scratch` contains transient data—conceptually, the content of each scratch collection is emptied at the start of each timestep.[3] Scratches are similar to views in SQL: they are often useful as a way to name intermediate results or as a "macro" construct to enable code reuse. A `channel` allows communication between Bloom instances. The schema of a channel has a distinguished *location specifier* column (prefixed with "@"). When a fact is derived for a channel collection, it is sent to the remote Bloom instance at the address given by the location specifier.

Name	Behavior
`table`	Persistent storage.
`scratch`	Transient storage.
`channel`	Asynchronous communication. A fact derived into a `channel` appears in the database of a remote Bloom instance at a nondeterministic future time.
`periodic`	Interface to the system clock.
`interface`	Defines the inputs and outputs of a Bloom module.

Table 2.1: Bloom collection types.

The `periodic` collection type allows a Bloom program to access the system clock. A `periodic` collection is declared with time interval (e.g., "1 second"); the Bloom runtime arranges (in a best-effort manner) for a new tuple to be inserted into the collection whenever the specified amount of wall-clock time has elapsed. In distributed programming, `periodic` collections are often used to implement timeouts. Finally, Bloom provides a simple module system which allows data encapsulation and thin interfaces between components. An abstract interface consists of a set of "input" and "output" collections, which are denoted by using the `interface` collection type. An implementation of an interface is a set of statements that read data from the module's input

[3] Note that the language runtime may choose to avoid emptying and recomputing scratch collections for every timestep, an optimization known as *view materialization* [70, 71]. The current version of Bud implements a simple version of this technique, which we found to be important for good performance.

collections and derive data into the module's output collections.

Statements

Each Bloom statement has one or more input collections and a single output collection. A statement takes the form:

<*collection-identifier*> <*op*> <*collection-expression*>

The left-hand side (lhs) is the name of the output collection and the right-hand side (rhs) is an expression that produces a collection. A statement defines how the input collections are transformed before being included (via set union) in the output collection. Bloom allows the usual relational operators to be used on the rhs (selection, projection, join, grouping, aggregation, and negation), although it adopts a syntax intended to be more familiar to imperative programmers. In Listing 2.1, line 11 demonstrates projection, line 12 performs a join between `link` and `path` using the join predicate `link.to = path.from` followed by a projection to four attributes, and line 13 shows grouping and aggregation. Bloom statements appear in one or more `bloom` blocks, which group together semantically related statements to aid readability.

Bloom provides several operators that determine *when* the rhs will be merged into the lhs (Table 2.2). The <= operator performs standard logical deduction: that is, the lhs and rhs are true at the same timestep. The <+ and <- operators indicate that facts will be added to or removed from the lhs collection at the beginning of the *next* timestep. The <~ operator specifies that the rhs will be merged into the lhs collection at some non-deterministic future time. The lhs of a statement that uses <~ must be a channel; the <~ operator captures asynchronous messaging.

Op	Name	Meaning
<=	*merge*	lhs includes the content of rhs in the current timestep.
<+	*deferred merge*	lhs will include the content of rhs in the next timestep.
<-	*deferred delete*	lhs will not include the content of rhs in the next timestep.
<~	*async merge*	(Remote) lhs will include the content of the rhs at some non-deterministic future timestep.

Table 2.2: Bloom operators.

Bloom allows statements to be recursive—that is, the rhs of a statement can reference the lhs collection, either directly or indirectly. As in Datalog, care must be taken when combining negation and recursion: if the language allows negation and recursion to be used freely, it is possible to write programs that do not have a reasonable interpretation—for example, because they imply a contradiction such as "p iff $\neg p$" [27]. To avoid this problem, many different restrictions on the use of negation have been proposed in the literature (e.g., [63, 132, 133, 134, 150]). Bloom adopts a simple but conservative rule: we require that deductive statements (<= operator) must be *syntactically stratified* [15]: cycles through negation or aggregation are not allowed [13]. Interestingly, two of the Bloom variants we discuss in this thesis relax this restriction: Bloom^L (Chapter 3) expands the space of monotonic programs, while Bloom^{PO} (Chapter 5) allows programs with cycles through negation if the program also contains a

constraint that ensures such cycles will never be satisfied.

Conclusion

Bloom was designed as a general-purpose distributed programming language, albeit with a focus on building systems and data processing infrastructure. We believe Bloom is an attractive option for building a wide range of distributed systems, but in the remainder of this thesis we focus on a narrower problem: exploring the ways in which Bloom can be used to deal with the challenges encountered building applications on top of loose consistency guarantees.

Chapter 3

Bloom^L: Distributed Programming with Join Semilattices

As discussed in Chapter 1, a signature problem in loosely consistent distributed programming is the fact that different replicas might observe updates in different orders, and hence might reach different conclusions. The replicas may be unable to agree on a consistent global order for all updates because the replicas may not be able to communicate (e.g., due to a network partition) [65]. In a strongly consistent system, this problem is avoided by preventing some nodes from accepting updates until communication has been restored (e.g., by using a consensus protocol to ensure that a "quorum" of nodes agree to accept an update before it is applied [97]). However, this amounts to giving up system availability during network partitions, which users of loosely consistent systems are typically unwilling to do [31].

Hence, loosely consistent systems must behave correctly even when different replicas observe updates in different orders. A common approach to handling this situation is to ensure that concurrent updates are *commutative*—this ensures that different replicas will reach the same outcome if they apply the same set of operations, even if the order in which those updates are applied is not the same. However, it is very difficult for application developers to verify that their operations commute for all possible update sequences, and so recent research has attempted to build tools to aid programmers in this task. In particular, two different approaches have received significant attention: *Convergent Modules* and *Monotonic Logic*.

Convergent Modules: In this approach, a programmer writes encapsulated modules whose public methods provide certain guarantees regarding message reordering and retry. For example, Statebox is an open-source library that merges conflicting updates to data items in a key-value store; the user of the library need only register "merge functions" that are commutative, associative, and idempotent [85]. This approach has roots in database and systems research [55, 62, 79, 118, 147] as well as groupware [54, 145]. Shapiro et al. proposed a formalism for these approaches called *Convergent Replicated Data Types* (CvRDTs), which casts these ideas into the algebraic framework of *semilattices* [35, 138, 139].

CvRDTs present two main problems: (a) the programmer bears responsibility for ensuring

lattice properties for their methods (commutativity, associativity, idempotence), and (b) CvRDTs only provide guarantees for individual values, not for application logic in general. As an example of this second point, consider the following:

Example 1 *A replicated, fault-tolerant courseware application assigns students into study teams. It uses two* set *CvRDTs: one for* Students *and another for* Teams*. The application reads a version of* Students *and inserts the derived element* <Alice,Bob> *into* Teams*. Concurrently,* Bob *is removed from* Students *by another application replica. The use of CvRDTs ensures that all replicas will eventually agree that* Bob *is absent from* Students*, but this is not enough: application-level state is inconsistent unless the derived values in* Teams *are updated consistently to reflect* Bob*'s removal. This is outside the scope of CvRDT guarantees.*

Taken together, the problems with Convergent Modules present a **scope dilemma**: a small module (e.g., a set) makes lattice properties easy to inspect and test, but provides only simple semantic guarantees. Large CvRDTs (e.g., an eventually consistent shopping cart) provide higher-level application guarantees but require the programmer to ensure lattice properties hold for a complex module, resulting in software that is difficult to test, maintain, and trust.

Monotonic Logic: In recent work, we observed that the database theory literature on monotonic logic provides a powerful lens for reasoning about distributed consistency. Intuitively, a monotonic program makes forward progress over time: it never "retracts" an earlier conclusion in the face of new information. We proposed the CALM theorem, which established that all monotonic programs are *confluent* (invariant to message reordering and retry) and hence eventually consistent [14, 81, 110]. Monotonicity of a Datalog program is straightforward to determine conservatively from syntax, so the CALM theorem provides the basis for a simple static analysis of the consistency of distributed programs: if a Bloom statement applies a non-monotonic operator to a collection that is derived from an asynchronously computed collection, the output of the operator may be non-deterministic [11]. If different replicas produce different non-deterministic outcomes for the same set of events, the system is not eventually consistent.

The original formulation of CALM and Bloom only verified the consistency of programs that compute sets of facts that grow over time ("set monotonicity"); that is, "forward progress" was defined according to set containment. As a practical matter, this is overly conservative: it precludes the use of common monotonically increasing constructs such as timestamps and sequence numbers.

Example 2 *In a quorum voting service, a coordinator counts the number of votes received from participant nodes; quorum is reached once the number of votes exceeds a threshold. This is clearly monotonic: the vote counter increases monotonically, as does the threshold test (*`count(votes)` $> k$*) which "grows" from False to True. But both of these constructs (upward-moving mutable variables and aggregates) are labeled non-monotonic by the original CALM analysis.*

The CALM theorem obviates any scoping concerns for convergent monotonic logic, but it presents a **type dilemma**. Sets are the only data type amenable to CALM analysis, but the programmer may have a more natural representation of a monotonically growing phenomenon. For example, a monotonic counter is more naturally represented as a growing integer than a growing set.

This dilemma leads either to false negatives in CALM analysis and over-use of coordination, or to idiosyncratic set-based implementations that can be hard to read and maintain.

Bloom^L: Logic And Lattices

We address the two dilemmas above with *Bloom^L,* an extension to Bloom that incorporates a semilattice construct similar to CvRDTs. We present this construct in detail below, but the intuition is that Bloom^L programs can be defined over arbitrary types—not just sets—as long as they have commutative, associative, and idempotent *merge functions* ("least upper bound") for pairs of items. Such a merge function defines a partial order for its type. This generalizes Bloom (and traditional Datalog), which assumes a fixed merge function (set union) and partial order (set containment).

Bloom^L provides three main improvements in the state of the art of both Bloom and CvRDTs:

1. Bloom^L solves the type dilemma of logic programming: CALM analysis in Bloom^L can assess monotonicity for arbitrary lattices, making it significantly more liberal in its ability to test for confluence. Bloom^L can validate the coordination-free use of common constructs like timestamps and sequence numbers.

2. Bloom^L solves the scope dilemma of CvRDTs by providing monotonicity-preserving mappings between lattices via *morphisms* and *monotone functions*. Using these mappings, the per-component monotonicity guarantees offered by CvRDTs can be extended across multiple items of lattice type. This capability is key to the CALM analysis described above. It is also useful for proving the monotonicity of sub-programs even when the whole program is not designed to be monotonic.

3. For efficient incremental execution, we extend the standard Datalog semi-naive evaluation scheme [22] to support lattices. We also describe how to extend an existing Datalog runtime to support lattices with relatively minor changes.

Outline

The remainder of the chapter proceeds as follows. Section 3.1 provides background on confluence, monotonicity, and the CALM Theorem. In Section 3.2 we introduce Bloom^L and show how Datalog-style logic programming can be extended to support join semilattices, cross-lattice morphisms, and monotone functions. We detail Bloom^L's built-in lattice types and show how developers can define new lattices. We also describe how the CALM analysis extends to Bloom^L. In Section 3.3, we describe how we modified the Bloom runtime to support Bloom^L.

We then present two case studies demonstrating how Bloom^L can be used to build distributed programs. In Section 3.4, we use Bloom^L to implement a distributed key-value store that supports eventual consistency, object versioning using vector clocks, and quorum replication. In Section 3.5, we revisit the simple e-commerce scenario presented by Alvaro et al. in which clients interact with a replicated shopping cart service [11]. We show how Bloom^L can be used to make the "checkout"

operation monotonic and confluent, despite the fact that it requires aggregation over a distributed data set.[1]

3.1 Background: CALM Analysis

Work on deductive databases has long drawn a distinction between *monotonic* and *non-monotonic* logic programs. Intuitively, a monotonic program only computes more information over time—it never "retracts" a previous conclusion in the face of new information. In Bloom (and Datalog), a simple conservative test for monotonicity is based on program syntax: selection, projection, and join are monotonic, while aggregation, negation, and deletion are not.

The CALM theorem connects the theory of monotonic logic with the practical problem of distributed consistency [11, 81]. Monotonic programs are *confluent*: for any given input, all program executions result in the same final state regardless of network non-determinism. Confluence is a strictly stronger property than eventual consistency: that is, all confluent programs are eventually consistent [14, 110].[2] Hence, monotonic logic is a useful building block for loosely consistent distributed programming.

According to the CALM theorem, distributed inconsistency may only occur at a *point of order*: a program location where an asynchronously derived value is consumed by a non-monotonic operator [11]. This is because asynchronous messaging results in non-deterministic arrival order, and non-monotonic operators may produce different conclusions when evaluated over different subsets of their inputs. For example, consider a Bloom program consisting of a pair of collections A and B (both fed by asynchronous channels) and a statement that sends a message whenever an element of A arrives that is not in B. This program is non-monotonic and exhibits non-confluent behavior: the messages sent by the program will depend on the order in which the elements of A and B arrive.

We have implemented a conservative static program analysis in Bloom that follows directly from the CALM theorem. Programs that are free from non-monotonic constructs are "blessed" as confluent: producing the same output on different runs or converging to the same state on multiple distributed replicas. Otherwise, programs are flagged as potentially inconsistent. To achieve consistency, the programmer either needs to rewrite their program to avoid the use of non-monotonicity or introduce a coordination protocol to ensure that a consistent ordering is agreed upon at each of the program's points of order. As discussed in Chapter 1, coordination protocols increase latency and reduce availability in the event of network partitions, so in this chapter we focus on deterministic coordination-free designs.

[1] An abbreviated version of the material in this chapter appeared in the Proceedings of the ACM Symposium on Cloud Computing [42].

[2] Note that the converse is not true: some eventually consistent programs are not confluent. For example, if a replica uses a "last writer wins" policy to resolve conflicting updates, the "last writer" is determined by network arrival order. If the system ensures that all replicas agree on the "last write", this system would be eventually consistent but non-deterministic, because the "winning" write is not determined strictly from the input. We have explored non-deterministic but eventually consistent programs in recent work [10]; in this chapter we focus on deterministic (confluent) programs.

```
1  QUORUM_SIZE = 5
2  RESULT_ADDR = "example.org"

4  class QuorumVote
5    include Bud

7    state do
8      channel :vote_chn, [:@addr, :voter_id]
9      channel :result_chn, [:@addr]
10     table   :votes, [:voter_id]
11     scratch :cnt, [] => [:cnt]
12   end

14   bloom do
15     votes      <= vote_chn {|v| [v.voter_id]}
16     cnt        <= votes.group(nil, count(:voter_id))
17     result_chn <~ cnt {|c| [RESULT_ADDR] if c >= QUORUM_SIZE}
18   end
19 end
```

Listing 3.1: A non-monotonic Bloom program that waits for a quorum of votes to be received.

Limitations Of Set Monotonicity

The original formulation of the CALM theorem considered only programs that compute more facts over time—that is, programs whose *sets* grow monotonically. Many distributed protocols make progress over time but their notion of "progress" is often difficult to represent as a growing set of facts. For example, consider the Bloom program in Listing 3.1. This program receives votes from one or more clients (not shown) via the `vote_chn` channel. Once at least `QUORUM_SIZE` votes have been received, a message is sent to a remote node to indicate that quorum has been reached (line 17). This program resembles a "quorum vote" subroutine that might be used by an implementation of Paxos [97] or quorum replication [64].

Intuitively, this program makes progress in a semantically monotonic fashion: the set of received votes grows and the size of the `votes` collection can only increase, so once a quorum has been reached it will never be retracted. Unfortunately, the current CALM analysis would regard this program as non-monotonic because it contains a point of order: the grouping operation on line 16.

To solve this problem, we need to introduce a notion of program values that "grow" according to a partial order other than set containment. We do this by extending Bloom to operate over arbitrary lattices rather than just the set lattice.

3.2 Language Constructs

This section introduces Bloom^L, an extension to Bloom that allows monotonic programs to be written using arbitrary lattices. We begin by reviewing the algebraic properties of lattices, monotone functions, and morphisms. We then introduce the basic concepts of Bloom^L and detail the built-in lattices provided by the language. We also show how users can define their own lattice types.

When designing Bloom^L, we decided to extend Bloom to include support for lattices rather than building a new language from scratch. Hence, Bloom^L is backward compatible with Bloom and was implemented with relatively minor changes to the Bud runtime. We describe how code written using lattices can interoperate with traditional Bloom collections in Section 3.2.

Definitions

A *bounded join semilattice* is a triple $\langle S, \sqcup, \bot \rangle$, where S is a set, $\sqcup$ is a binary operator (called "join" or "least upper bound"), and $\bot$ is an element of S (called "bottom"). The $\sqcup$ operator is associative, commutative, and idempotent. The $\sqcup$ operator induces a partial order $\leq_S$ on the elements of S: $x \leq_S y$ if $x \sqcup y = y$. Note that although $\leq_S$ is only a partial order, the least upper bound is defined for all elements $x, y \in S$. The distinguished element $\bot$ is the smallest element in S: $x \sqcup \bot = x$ for every $x \in S$. For brevity, we use the term "lattice" to mean "bounded join semilattice" in the rest of this chapter. We use the informal term "merge function" to mean "least upper bound."

A *monotone function* is a function $f : S \to T$ such that S, T are partially ordered sets (posets) and $\forall a, b \in S : a \leq_S b \Rightarrow f(a) \leq_T f(b)$. That is, f maps elements of S to elements of T in a manner that respects the partial orders of both posets.[3]

A *morphism* from lattice $\langle X, \sqcup_X, \bot_X \rangle$ to lattice $\langle Y, \sqcup_Y, \bot_Y \rangle$ is a function $g : X \to Y$ such that $g(\bot_X) = \bot_Y$ and $\forall a, b \in X : g(a \sqcup_X b) = g(a) \sqcup_Y g(b)$. Intuitively, g maps elements of X to elements of Y in a way that preserves the lattice properties. Note that morphisms are monotone functions but the converse is not true in general.

Language Constructs

Bloom^L allows both lattices and collections to represent state. A lattice is analogous to a collection type in Bloom, while a *lattice element* corresponds to a particular collection. For example, the `lset` lattice is similar to the `table` collection type provided by Bloom; an element of the `lset` lattice is a particular set. In the terminology of object-oriented programming, a lattice is a class that obeys a certain interface and an element of a lattice is an instance of that class. Listing 3.2 contains an example Bloom^L program.

As with collections, the lattices used by a Bloom^L program are declared in a `state` block. More precisely, a state block declaration introduces an identifier that is associated with a lattice element; over time, the binding between identifiers and lattice elements is updated to reflect state changes in the program. For example, line 10 of Listing 3.2 declares an identifier `votes` that is mapped to an element of the `lset` lattice. As more votes are received, the lattice element associated with the `votes` identifier changes (it moves "upward" in the `lset` lattice). When a lattice identifier is declared, it is initially bound to $\bot$, the smallest element in the lattice. For example, an `lset` lattice initially contains the empty set.

[3] To simplify the presentation, these definitions assume that monotone functions and morphisms are unary. Bloom^L supports monotone functions and morphisms with any number of arguments.

```
1  QUORUM_SIZE = 5
2  RESULT_ADDR = "example.org"

4  class QuorumVoteL
5    include Bud

7    state do
8      channel :vote_chn, [:@addr, :voter_id]
9      channel :result_chn, [:@addr]
10     lset    :votes
11     lmax    :cnt
12     lbool   :quorum_done
13   end

15   bloom do
16     votes       <= vote_chn {|v| v.voter_id}
17     cnt         <= votes.size
18     quorum_done <= cnt.gt_eq(QUORUM_SIZE)
19     result_chn  <~ quorum_done.when_true { [RESULT_ADDR] }
20   end
21 end
```

Listing 3.2: A monotonic Bloom^L program that waits for a quorum of votes to be received.

Statements

Statements take the same form in both Bloom and Bloom^L:

<identifier> <op> <expression>

The identifier on the lhs can refer to either a set-oriented collection or a lattice element. The expression on the rhs can contain both traditional relational operators (applied to Bloom collections) and methods invoked on lattices. Lattice methods are similar to methods in an object-oriented language and are invoked using the standard Ruby method invocation syntax. For example, line 17 of Listing 3.2 invokes the `size` method on an element of the `lset` lattice.

If the lhs is a lattice, the statement's operator must be either <= or <+ (instantaneous or deferred deduction, respectively). The meaning of these operators is that, at either the current or the following timestep, the lhs identifier will take on the result of applying the lattice's least upper bound to the lhs and rhs lattice elements. The intuition remains the same as in Bloom: the rhs value is "merged into" the lhs lattice, except that the semantics of the merge operation are defined by the lattice's least upper bound operator. We require that the lhs and rhs refer to a lattice of the same type.

Bloom^L does not support deletion (<- operator) for lattices. Lattices do not directly support asynchronous communication (via the <~ operator) but lattice elements can be embedded into facts that appear in channels (Section 3.2).

Lattice Methods

Bloom^L statements compute values over lattices by invoking methods on lattice elements. Just as a subset of the relational algebra is monotonic, some lattice methods are monotone functions (as defined in Section 3.2). A monotone lattice method guarantees that, if the lattice on which the

Name	Description	Least Element	Merge(a, b)	Morphisms	Monotone Functions
`lbool`	Boolean (false → true)	`false`	$a \vee b$	`when_true(&blk)` → `v`	

`lmax`	Max over an ordered domain	$-\infty$	$max(a, b)$	`gt(n)` → `lbool`
				`gt_eq(n)` → `lbool`
				`+(n)` → `lmax`
				`-(n)` → `lmax`	

`lmin`	Min over an ordered domain	∞	$min(a, b)$	`lt(n)` → `lbool`
				`lt_eq(n)` → `lbool`
				`+(n)` → `lmin`
				`-(n)` → `lmin`	

`lset`	Set of values	empty set	$a \cup b$	`intersect(lset)` → `lset`
				`project(&blk)` → `lset`
				`product(lset)` → `lset`
				`contains?(v)` → `lbool`
					`size()` → `lmax`

`lpset`	Set of non-negative numbers	empty set	$a \cup b$	`intersect(lpset)` → `lpset`
				`project(&blk)` → `lpset`
				`product(lpset)` → `lpset`
				`contains?(v)` → `lbool`
					`size()` → `lmax`
					`sum()` → `lmax`

`lbag`	Multiset of values	empty multiset	$a \cup b$	`intersect(lbag)` → `lbag`
				`project(&blk)` → `lbag`
				`multiplicity(v)` → `lmax`
				`contains?(v)` → `lbool`
				`+(lbag)` → `lbag`
					`size()` → `lmax`

`lmap`	Map from keys to lattice values	empty map	see text	`intersect(lmap)` → `lmap`
				`project(&blk)` → `lmap`
				`key_set()` → `lset`
				`at(v)` → `any-lattice`
				`key?(v)` → `lbool`
					`size()` → `lmax`

Table 3.1: Built-in lattices provided by Bloom^L. Note that `v` denotes a Ruby value, `n` denotes a number, and `blk` indicates a Ruby code block (anonymous function).

method is invoked grows (according to the lattice's partial order), the value returned by the method will grow (according to the return value's lattice type). For example, the `size` method provided by the `lset` lattice is monotone because as more elements are added to the set, the size of the set increases. Intuitively, a lattice's monotone methods constitute a "safe" interface of operations that can be invoked in a distributed setting without risk of inconsistency.

A lattice method's signature indicates its monotonicity properties. Bloom^L distinguishes between methods that are monotone and a subset of monotone methods that are *morphisms*. Section 3.2

defines the properties that a morphism must satisfy, but the intuition is that a morphism on lattice T can be distributed over T's least upper bound. For example, the `size` method of the `lset` lattice is not a morphism. To see why, consider two elements of the `lset` lattice, $\{1, 2\}$ and $\{2, 3\}$. The `size` method is not a morphism because $size(\{1, 2\} \sqcup_{\texttt{lset}} \{2, 3\}) \neq size(\{1, 2\}) \sqcup_{\texttt{lmax}} size(\{2, 3\})$. Morphisms can be evaluated more efficiently than monotone methods, as we discuss in Section 3.3.

Lattices can also define non-monotonic methods. Using a non-monotonic lattice method is analogous to using a non-monotonic relational operator in Bloom: the Bud interpreter stratifies the program to ensure that the input value is computed to completion before allowing the non-monotonic method to be invoked. Bloom^L encourages developers to minimize the use of non-monotonic constructs: as the CALM analysis suggests, non-monotonic reasoning may need to be augmented with coordination to ensure consistent results.

Every lattice defines a non-monotonic `reveal` method that returns a representation of the lattice element as a plain Ruby value. For example, the `reveal` method on an `lmax` lattice returns a Ruby integer. This method is non-monotonic because once the underlying Ruby value has been extracted from the lattice, Bloom^L cannot ensure that subsequent code uses the value in a monotonic fashion.

Built-in Lattices

Table 3.1 lists the lattices included with Bloom^L. The built-in lattices provide support for several common notions of "progress": a predicate that moves from false to true (`lbool`), a numeric value that strictly increases or strictly decreases (`lmax` and `lmin`, respectively), and various kinds of collections that grow over time (`lset`, `lpset`, `lbag`, and `lmap`). The behavior of most of the lattice methods should be unsurprising, so we do not describe every method in this section.

The `lbool` lattice represents conditions that, once satisfied, remain satisfied. For example, the `gt` morphism on the `lmax` lattice takes a numeric argument n and returns an `lbool`; once the `lmax` exceeds n, it will remain $> n$. The `when_true` morphism takes a Ruby block; if the `lbool` element has the value `true`, `when_true` returns the result of evaluating the block. For example, see line 19 in Listing 3.2. `when_true` is similar to an "if" statement.[4]

The collection-like lattices support familiar operations such as union, intersection, and testing for the presence of an element in the collection. The `project` morphism takes a code block and forms a new collection by applying the code block to each element of the input collection. Elements for which the code block returns `nil` are omitted from the output collection, which allows `project` to be used as a filter.

The `lbag` lattice demonstrates how Bloom^L can support multisets. Note that the `lbag` merge function takes the maximum of the multiplicities of an element that appears in both input lattices. Although this is the standard definition of multiset union, some applications might find summing the element multiplicities to be more useful. However, this behavior would not be a valid least upper bound (because it is not idempotent). Instead, applications that need to compute the multiset sum can use the + morphism.

[4] An "else" clause would test for an upper bound on the final lattice value, which is a non-monotonic property.

```
1  class Bud::SetLattice < Bud::Lattice
2    wrapper_name :lset

4    def initialize(x=[])
5      @v = x.uniq # Remove duplicates from input
6    end

8    def merge(i)
9      self.class.new(@v | i.reveal)
10   end

12   morph :intersect do |i|
13     self.class.new(@v & i.reveal)
14   end

16   morph :contains? do |i|
17     Bud::BoolLattice.new(@v.member? i)
18   end

20   monotone :size do
21     Bud::MaxLattice.new(@v.size)
22   end
23 end
```

Listing 3.3: Example set lattice implementation in Ruby.

The `lpset` lattice is an example of how Bloom^L can be used to encode domain-specific knowledge about an application. If the developer knows that a set will only contain non-negative numbers, the sum of those numbers increases monotonically as the set grows. Hence, `sum` is a monotone function of `lpset`; in contrast, taking the sum of the elements of an `lset` is non-monotonic in general.

The `lmap` lattice associates keys with values. Keys are immutable Ruby objects and values are lattice elements. For example, a web application could use an `lmap` to associate session IDs with an `lset` containing the pages visited by that session. The `lmap` merge function takes the union of the key sets of its input maps. If a key occurs in both inputs, the two corresponding values are merged using the appropriate lattice merge function. Note that the `at(v)` morphism returns the lattice element associated with key `v` (or $\bot$ if the `lmap` does not contain `v`).

User-defined Lattices

The built-in lattices are sufficient to express many programs. However, Bloom^L also allows developers to create custom lattices to capture domain-specific behavior. To define a new lattice, a developer creates a Ruby class that meets a certain API contract. Listing 3.3 shows a simple implementation of the `lset` lattice using a Ruby array for storage.[5]

[5]In Listing 3.3, we present a slightly simplified `lset` implementation: we omit a few methods for the sake of exposition and use an array to store the contents of the set. The actual `lset` implementation in Bloom^L uses a hash-based set data type, which yields better performance (e.g., duplicate elimination is more efficient).

A lattice class must inherit from the built-in `Bud::Lattice` class and must also define two methods:

- `initialize(i)`: given a Ruby object *i*, this method constructs a new lattice element that "wraps" *i* (this is the standard Ruby syntax for defining a constructor). If *i* is `nil` (the null reference), this method returns $\perp$.
- `merge(e)`: given a lattice element *e*, this method returns the least upper bound of *e* and *self*. The programmer must ensure that this method satisfies the algebraic properties of least upper bound—in particular, it must be commutative, associative, and idempotent (Section 3.2). Note that *e* and *self* must be instances of the same class.

Lattices can also define any number of monotone functions, morphisms, and non-monotonic methods. The syntax for declaring morphisms and monotone functions can be seen in lines 12–14 and 20–22 of Listing 3.3, respectively. Note that lattice elements are *immutable*—that is, lattice methods (including `merge` methods) must return new values, rather than modifying any of their inputs.

Because lattice methods contain arbitrary Ruby code, the author of a lattice collection should be careful to ensure that lattice methods satisfy the appropriate algebraic properties. For example, implementing a lattice method might require examining the underlying Ruby value contained in another lattice element. This can be done using the `reveal` method (e.g., line 13 in Listing 3.3). Since `reveal` is not monotonic, developers should use it carefully when implementing monotonic lattice methods. We discuss some ideas for how to assist developers in writing correct lattice implementations in Section 3.7.

A lattice definition must specify a keyword that can be used in Bloom^L state blocks. This is done using the `wrapper_name` class method. For example, line 2 of Listing 3.3 means that "`lset :foo`" in a state block will introduce an identifier `foo` that is associated with an instance of `Bud::SetLattice`.

Integration With Set-Oriented Logic

Bloom^L provides two features to ease integration of lattice-based code with Bloom rules that use set-oriented collections.

Converting Collections Into Lattices

This feature enables an intuitive syntax for merging the contents of a set-oriented collection into a lattice. If a statement has a Bloom collection on the rhs and a lattice on the lhs, the collection is converted into a lattice element by "folding" the lattice's merge function over the collection. That is, each element of the collection is converted to a lattice element (by invoking the lattice constructor) and then the resulting lattice elements are merged together via repeated application of the lattice's `merge` method. In our experience, this is usually the behavior intended by the user.

For example, line 16 of Listing 3.2 contains a Bloom collection on the rhs and an `lset` lattice on the lhs. This statement is evaluated by constructing a singleton `lset` for each fact in the rhs

collection and then merging the sets together. The resulting `lset` is then merged with the `votes` lattice referenced by the lhs.

Embedding Lattice Values In Collections

Bloom^L allows lattice elements to be used as column values of facts in Bloom collections. This feature allows Bloom^L programs to use a mixture of Bloom-style relational operators and lattice method invocations, depending on which is more convenient. Bloom also provides several collection types with special semantics (e.g., network communication, durable storage); allowing lattice elements to be embedded into collections avoids the need to create a redundant set of facilities for lattices.

Consider a Bloom^L rule that derives facts with an embedded lattice element as a column:

```
t1 <= t2 {|t| [t.x, cnt]}
```

where `t1` and `t2` are collections, `cnt` is a lattice, and the key of `t1` is its first column. Note that `cnt` might change over the course of a single timestep (specifically, `cnt` can increase according to the lattice's partial order). This might result in deriving multiple `t1` facts that differ only in the second column, which would violate `t1`'s key.

To resolve this situation, Bloom^L allows multiple facts to be derived that differ only in their embedded lattice values; those facts are merged into a single fact using the lattice's merge function. This is similar to specifying a procedure for how to resolve key constraint violations, a feature supported by some databases [119, 142]. For similar reasons, lattice elements cannot be used as keys in Bloom collections.

CALM Analysis For Lattices

As discussed in Section 3.1, CALM-based program analysis determines whether a program is order-sensitive by looking for "points of order"—locations where an asynchronously computed value is consumed by a non-monotonic operator. Bloom^L simply expands the set of monotonic operations that a program can contain; hence, we did not need to make any fundamental changes to our program analysis. Instead, we simply replaced the hard-coded list of monotonic operations with a list of the monotonic methods defined by the lattices used by the current program. In Sections 3.4 and 3.5, we show how CALM analysis can be applied to several Bloom^L programs.

CALM analysis is grounded in a model-theoretic characterization of confluence [110]. This work was done in the context of Dedalus, the formal language on which Bloom is based [13]. These results apply directly to Bloom, whose semantics are grounded in those of Dedalus. Monotonicity analysis in Bloom^L is a natural extension of the work on Bloom, but the development of a formal model-theoretic semantics for Bloom^L remains a topic for future work.

3.3 Program Evaluation

In this section, we describe how to evaluate Bloom^L programs. First, we generalize semi-naive evaluation to support lattices. We validate that our implementation of semi-naive evaluation results in significant performance gains and is competitive with the traditional set-only semi-naive evaluation scheme in Bud. We also describe how we extended Bud to support Bloom^L with relatively few changes.

Evaluation Strategy

Naive evaluation is a simple but inefficient approach to evaluating recursive Datalog programs. Evaluation proceeds in rounds. In each round, every rule in the program is evaluated over the entire database, including all derivations made in previous rounds. This process stops when a round makes no new derivations. Naive evaluation is inefficient because it makes many redundant derivations: once a fact has been derived in round i, it is rederived in every round $> i$.

Semi-naive evaluation improves upon naive evaluation by making fewer redundant derivations [22]. Let Δ_0 represent the initial database state. In the first round, all the rules are evaluated over Δ_0; let Δ_1 represent the new facts derived in this round. In the second round, we only need to compute derivations that depend on Δ_1 because everything that can be derived purely from Δ_0 has already been computed.

A similar evaluation strategy works for Bloom^L statements that use lattice morphisms. For a given lattice identifier l, let Δ_l^0 represent the lattice element associated with l at the start of the current timestep. Let Δ_l^r represent the new derivations for l that have been made in evaluation round r. During round one, the program's statements are evaluated and l is mapped to Δ_l^0; this computes Δ_l^1. In round two, l is now mapped to Δ_l^1 and evaluating the program's statements yields Δ_l^2. This process continues until $\Delta_l^i = \Delta_l^{i+1}$ for all identifiers l. The final value for l is given by $\bigsqcup_{l:\, j=0}^{i} \Delta_l^j$.

This optimization cannot be used for monotone functions that are not morphisms. This is because semi-naive evaluation requires that we apply functions to the partial results derived in each round k into Δ_l^k, and later combine them using the lattice's merge operation—effectively distributing the function across the merge. For example, consider computing the `lset` lattice's `size` method, which returns an `lmax` lattice. The semi-naive strategy would compute $\bigsqcup_{\texttt{lmax}:\, j=0}^{i} \mathit{size}(\Delta_{\texttt{lset}}^j)$—the maximum of the sizes of the incremental results produced in each round. Thus it produces a different result than naive evaluation, which evaluates the `size` function against the complete database state in each round.

Implementing semi-naive style evaluation for lattices was straightforward. For each lattice identifier l, we record two values: a "total" value (the least upper bound of the derivations made for l in all previous rounds) and a "delta" value (the least upper bound of the derivations made for l in the last round). We implemented a program rewrite that examines each Bloom^L statement. If a statement only applies morphisms to lattice elements, the rewrite adjusts the statement to use the lattice's delta value rather than its total value.

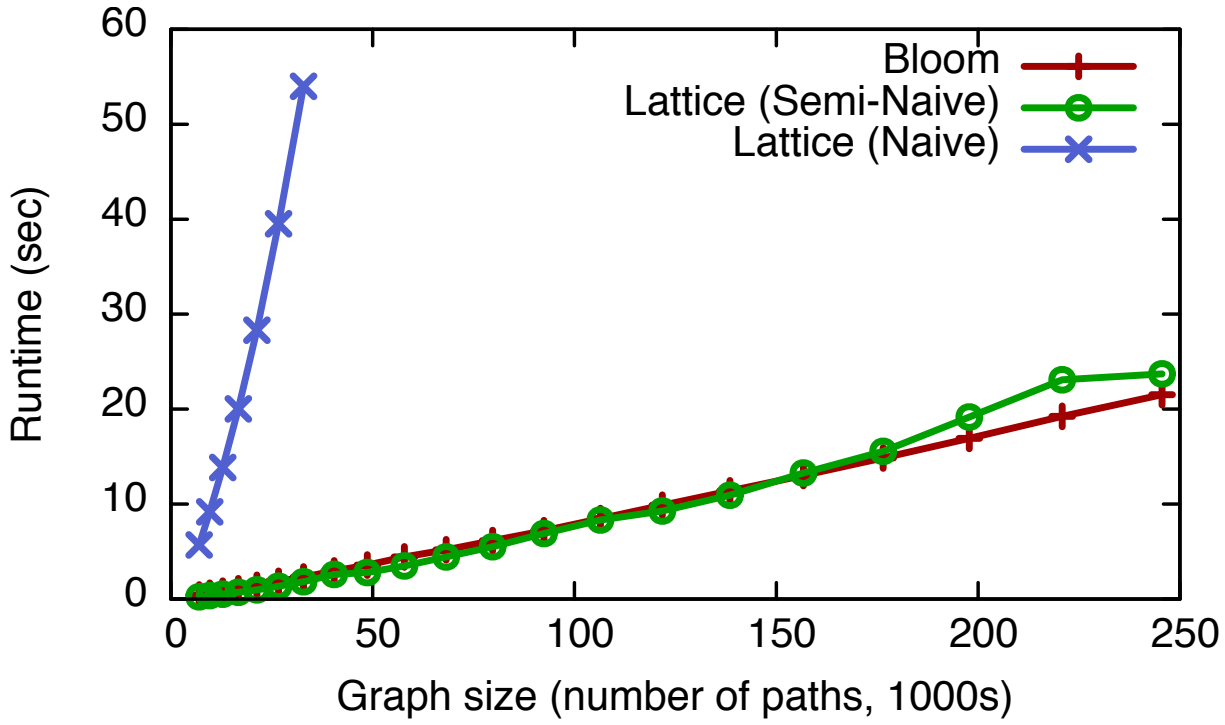


Figure 3.1: Performance of three different methods for computing the transitive closure of a graph.

Performance Validation

To validate the effectiveness of semi-naive evaluation for Bloom^L programs, we wrote two versions of a program to compute the transitive closure of a directed acyclic graph. One version was written in Bloom and used Bloom collections. The other version was written in Bloom^L using morphisms over the `lset` lattice. For the Bloom^L version, we ran the program both with and without semi-naive evaluation enabled. As input, we generated synthetic graphs of various sizes—in a graph with n nodes, each node had roughly $\log_2 n$ outgoing edges. We ran the experiment using a 2.13 GHz Intel Core 2 Duo processor and 4GB of RAM, running Mac OS X 10.7.4 and Ruby 1.9.3-p194. We ran each program variant five times on each graph and report the mean elapsed wall-clock time.

Figure 3.1 shows how the runtime of each program varied with the size of the graph. Note that we only report results for the naive Bloom^L strategy on small input sizes because this variant ran very slowly as the graph size increased. The poor performance of naive evaluation is not surprising: after deriving all paths of length n, naive evaluation will then rederive all those paths at every subsequent round of the fixpoint computation. In contrast, after computing length n paths, a semi-naive strategy will only generate length $n + 1$ paths in the next round. Bloom and semi-naive Bloom^L achieve similar results. We instrumented Bud to count the number of derivations made by the Bloom and semi-naive lattice variants—as expected, both programs made a similar number of derivations. These results suggest that our implementation of semi-naive evaluation for Bloom^L is effective and

performs comparably with a traditional implementation of semi-naive evaluation for sets.

For large inputs, Bloom began to outperform the semi-naive lattice variant. We suspect this is because the lattice implementation copies more data than Bloom does for this benchmark. Lattice elements are immutable, so the `lset` merge function allocates a new object to hold the result of the merge. In contrast, Bloom collections are modified in-place. We plan to improve the lattice runtime to avoid copies when it can determine that in-place updates are safe.

Discussion

When designing Bloom^L, we chose to extend the original Bloom language rather than inventing a new language from scratch. This design philosophy also applied to our language implementation: we found we were able to extend Bud to support Bloom^L with relatively minor changes.

Bud initially had about 7300 lines of Ruby source code (LOC). Adding support for Bloom^L required less than 1000 lines of new or modified code; moreover, most of these changes were cleanly separated from the core Bud code. For example, the built-in lattice types constituted 300 LOC and support for lattice-based query plan elements required 250 LOC. In contrast, we had to make no changes to Bud's core fixpoint loop and extending CALM analysis to support lattices required modifying only 10 LOC. This experience suggests that support for lattices can be added to an existing Datalog engine in a relatively straightforward manner.

3.4 Case Study: Key-Value Store

The next two sections contain case studies that show how Bloom^L can be used to build practical distributed programs.[6] In the first case study, we show that a distributed, eventually consistent key-value store can be *composed* via a series of monotonic mappings between simple lattices. This example shows how Bloom^L overcomes the "scope dilemma" of CvRDTs: by composing a complex program from simple lattices (mostly Bloom^L built-ins), we can feel confident that individual lattices are correct, while CALM analysis finishes the job of verifying whole-program consistency.

System Architecture

A key-value store (KVS) provides a lookup service that allows client applications to store and retrieve the *value* associated with a given *key*. Key-value pairs are typically replicated on multiple storage nodes for redundancy and the key space is partitioned to improve aggregate storage and throughput. As discussed in Chapter 1, *eventual consistency* is a common correctness criterion: after all client updates have reached all storage nodes, all the replicas of a key-value pair will eventually converge to the same final state [147, 152].

Listing 3.4 shows a simple KVS interface in Bloom^L. Client applications initiate *get(key)* and *put(key, val)* operations by inserting facts into the `kvget` and `kvput` channels, respectively; server replicas return responses via the `kvget_resp` and `kvput_resp` channels.

[6] Complete code listings for these case studies can be found in Appendix A.

```
1  module KvsProtocol
2    state do
3      channel :kvput, [:reqid, :@addr] => [:key, :val, :client_addr]
4      channel :kvput_resp, [:reqid] => [:@addr, :replica_addr]
5      channel :kvget, [:reqid, :@addr] => [:key, :client_addr]
6      channel :kvget_resp, [:reqid] => [:@addr, :val, :replica_addr]
7    end
8  end
```

Listing 3.4: KVS interface in Bloom^L.

```
1  class KvsReplica
2    include Bud
3    include KvsProtocol

5    state do
6      lmap :kv_store
7    end

9    bloom do
10     kv_store   <= kvput {|c| {c.key => c.val}}
11     kvput_resp <~ kvput {|c| [c.reqid, c.client_addr, ip_port]}
12     kvget_resp <~ kvget {|c| [c.reqid, c.client_addr, kv_store.at(c.key), ip_port]}
13   end
14 end
```

Listing 3.5: KVS implementation in Bloom^L.

Listing 3.5 contains the Bloom^L code for a KVS server replica. An `lmap` lattice is used to maintain the mapping between keys and values (line 6). Since the values in an `lmap` lattice must themselves be lattice elements, for now we assume that clients only want to store and retrieve monotonically increasing lattice values; we discuss how to lift this restriction in Section 3.4. To handle a *put(key, val)* request, a new *key* → *val* `lmap` is created and merged into `kv_store` (line 10). If `kv_store` already contains a value for the given key, the two values will be merged together using the value lattice's merge function (see Section 3.2 for details). Observe that we use the Bloom^L features described in Section 3.2 to allow traditional Bloom collections (e.g., channels) and lattices (e.g., the `kv_store` lattice) to be used by the same program. Note also that `ip_port` is a built-in function that returns the IP address and port number of the current Bud instance.

The state of two replicas can be synchronized by simply exchanging their `kv_store` maps; the `lmap` merge function automatically resolves conflicting updates made to the same key. This property allows considerable flexibility in how replicas propagate updates.

Object Versioning

The basic KVS design is sufficient for applications that want to store monotonically increasing values such as session logs or increasing counters. To allow storage of values that change in arbitrary ways, we now consider how to support *object versions*. This is a classic technique for recognizing

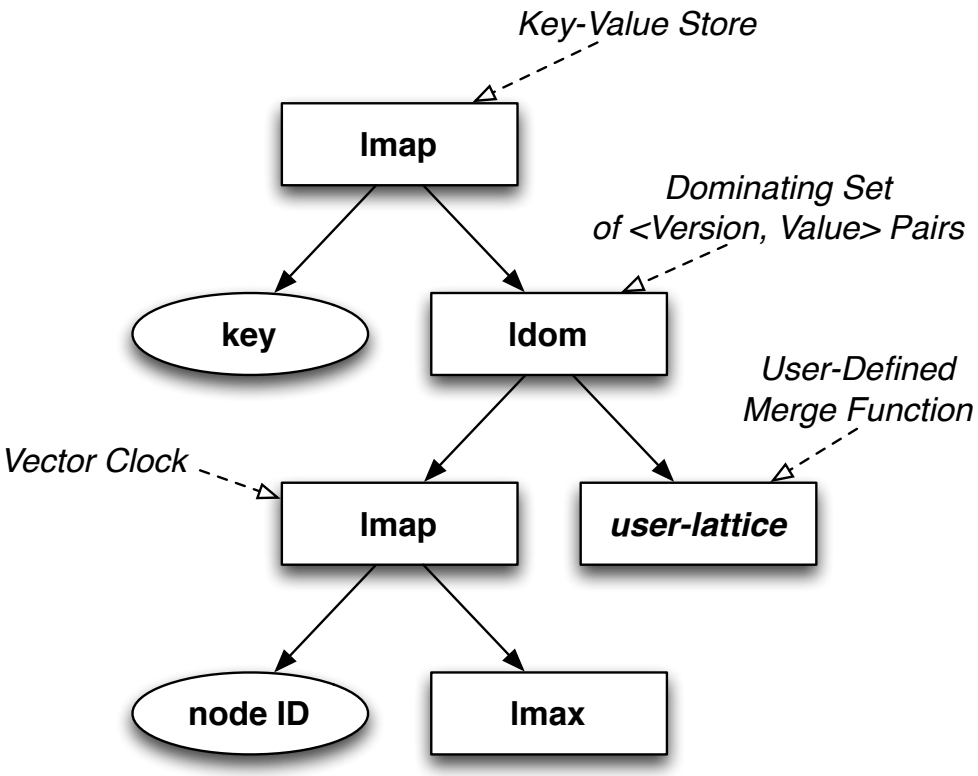


Figure 3.2: Lattice structure of a KVS with object versioning. Rectangles are lattices and ovals are atomic values.

and resolving mutual inconsistency between members of a distributed system [122]. Our design is similar to that used by Amazon Dynamo [50].

Each replica associates keys with sets of $\langle$*vector-clock*, *value*$\rangle$ pairs. The vector clock captures the causal relationship between different versions of a record [56, 113]. Clients store and retrieve $\langle$*vector-clock*, *value*$\rangle$ pairs. When a client updates a value it has previously read, the client increments its own position in the vector clock and includes the updated vector clock V_U with its *put* operation. Upon receiving an update, the server compares V_U with the set of vector clocks associated with the updated key. For each such vector clock V_S, the server considers three cases:

1. $V_U > V_S$: the client's update happens-after V_S, so V_S is replaced with V_U.
2. $V_U < V_S$: the client's update happens-before V_S, so V_S is retained.[7]
3. V_U and V_S are incomparable: the two versions are concurrent, so both V_U and V_S are retained.

By applying these rules, each replica retains a set of mutually concurrent versions for each key. To handle a *get* operation, the server returns a single $\langle$*vector-clock*, *value*$\rangle$ pair by merging together the

[7]This situation might arise due to duplication and reordering of messages by the network.

vector clocks and using a user-supplied reconciliation function to merge together the conflicting values, as described below. We call the set of incomparable version-value pairs a *dominating set*.

From a Bloom^L perspective, each replica still stores a monotonically increasing value—the only difference is that in this scheme, the *version* stored by a replica increases over time, rather than the associated value. Hence, we now consider how to support vector clocks and dominating sets using Bloom^L.

Vector Clocks

A vector clock is a map from node identifiers to logical clocks [56, 113]. Let V_e denote the vector clock for event e; let $V_e(n)$ denote the logical clock associated with node n in V_e. If $V_e < V_{e'}$, e causally precedes e', where

$$V_e < V_{e'} \equiv \forall x[V_e(x) \leq V_{e'}(x)] \wedge \exists y[V_e(y) < V_{e'}(y)]$$

In Bloom^L, a vector clock can be represented as an `lmap` that maps node identifiers to `lmax` values. Each `lmax` represents the logical clock of a single node; this is appropriate because the logical clock value associated with a given node will only increase over time. The merge function provided by `lmap` provides the desired semantics—that is, the built-in least upper bound for an `lmap` that contains `lmax` values is consistent with the partial order given above.[8]

Dominating Sets

We use a custom lattice type `ldom` to represent a dominating set. As discussed above, a dominating set is a set of $\langle version, value \rangle$ pairs; both elements of the pair are themselves represented as lattice values. In the KVS, the version lattice is a vector clock (that is, an `lmap` containing `lmax` values); the value lattice is whatever value the user wants to store in the KVS.[9]

The discussion above suggests a natural merge function for `ldom`: given two input `ldom` elements e and e', the merge function omits all "dominated" pairs. For example, a $\langle version, value \rangle$ pair in e is included in the result of the merge unless there is a pair in e' whose version is strictly greater, where "strictly greater" is defined by the semantics of the version lattice type.

The `ldom` lattice provides two functions, `version` and `value`, that return the least upper bound of the concurrent versions or values, respectively. In the KVS, the `value` function corresponds to reconciling conflicting updates, whereas `version` returns the vector clock associated with the reconciled value. Note that while the `version` of a given `ldom` increases over time (as new versions are observed), the `value` does not; hence, `version` is a monotone function but `value` is not. Since the goal of object versioning is to allow values to change in non-monotonic ways, this is the expected behavior.

[8] The observation that vector time has a lattice structure was made by Mattern [113].

[9] If the user stores a value that does not have a natural merge function, similar systems typically provide a default merge function that collects conflicting updates into a set for eventual manual resolution by the user. Such a strategy corresponds to using `lset` as the value lattice in Bloom^L.

Discussion

Fig. 3.2 shows the lattices used in a KVS with object versioning. Surprisingly, adding support for object versioning did not require *any* changes to the KVS replica code. Instead, clients simply store `ldom` values containing a single ⟨*vector-clock*, *value*⟩ pair, incrementing their position in the vector clock as described above. The KVS replica merges these `ldom` values into an `lmap` as usual; the `ldom` merge function handles conflict resolution in the appropriate manner. Moreover, by composing the KVS from a collection of simple lattices, we found it easy to reason about the behavior of the system. For example, convincing ourselves that KVS replicas will eventually converge only required checking that the individual `ldom`, `lmap`, and `lmax` lattices satisfy the lattice properties, rather than analyzing the behavior of the system as a whole.

Our design compares favorably to traditional implementations of object versioning and vector clocks. For example, the implementation of vector clocks alone in the Voldemort KVS requires 216 lines of Java, not including whitespace or comments [101]. In Bloom^L, vector clocks follow directly from the composition of the `lmap` and `lmax` lattices, and the *entire KVS* requires less than 100 lines of Ruby and Bloom^L code, including the client library. The `ldom` lattice requires an additional 50 lines of Ruby but is completely generic, and could be included as a built-in lattice.

Quorum Reads and Writes

To further demonstrate the flexibility of our implementation, we add an additional feature to our KVS: the ability to submit reads and writes to a configurable number of nodes. If a client reads from R nodes and writes to W nodes in a KVS with N replicas, the user can set $R + W > N$ to achieve behavior equivalent to a quorum replication system [64], or use smaller values of R and W if eventual consistency is sufficient. This scheme allows users to vary R and W on a per-operation basis, depending on their consistency, durability, and latency requirements.

To support this feature, we can use the Bloom^L quorum voting pattern introduced in Listing 3.2. After sending a write to W systems, the KVS client accumulates `kvput_resp` messages into an `lset`. Testing for quorum can be done in a monotonic fashion by mapping the `lset` to an `lmax` (using the `size` method) and then performing a threshold test using `gt_eq` on `lmax`. As expected, this is monotonic: once quorum has been reached, it will never be retracted.

Quorum reads work similarly except that the client must also merge together the R versions of the record it receives. This follows naturally from the discussion in Section 3.4: the client simply takes the least upper bound of the values it receives, which produces the expected behavior. The client can optionally write the merged value back to the KVS (so-called "read repair" [50]); note that the `ldom` merge method also updates the record's vector clock appropriately.

3.5 Case Study: Shopping Carts

In the previous section, we showed how a complete, consistent distributed program can be composed via monotonic mappings between simple lattice types. In this section, we describe how Bloom^L overcomes the "type dilemma" of Bloom. In prior work, we introduced a case study in Bloom that

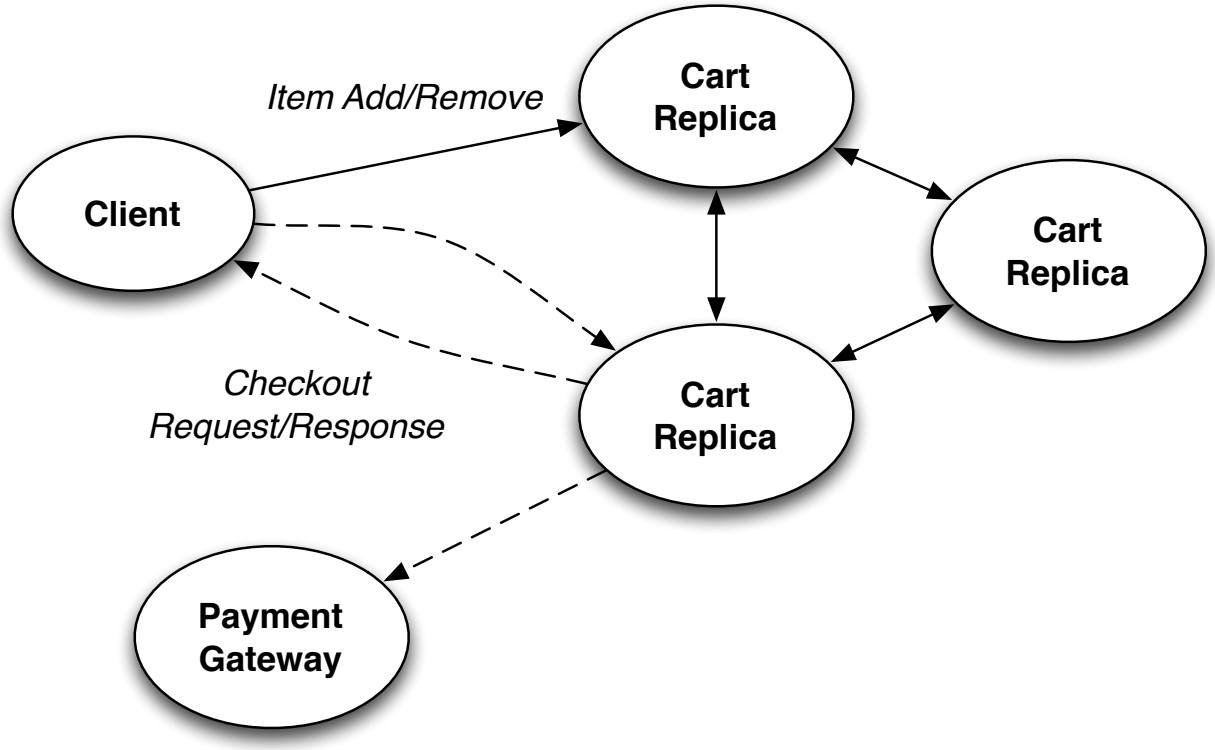


Figure 3.3: Shopping cart system architecture.

seemed to require coordination because of the use of distributed aggregation [11]. By using custom lattice types, the Bloom^L CALM analysis can verify that our revised design is eventually consistent without need for coordination.

Figure 3.3 depicts a simple e-commerce system in which clients interact with a shopping cart service by adding and removing items over the course of a shopping session. The cart service is replicated to improve fault tolerance; client requests can be routed to any server replica. Eventually, a client submits a "checkout" operation, at which point the cumulative effect of their shopping session should be summarized and returned to the client. In a practical system, the result of the checkout operation might be presented to the client for confirmation or submitted to a payment processor to complete the e-commerce transaction. This case study is based on the cart system from Alvaro et al. [11], which was in turn inspired by the discussion of replicated shopping carts in the Dynamo paper [50].

Monotonic Checkout

Alvaro et al. argue that processing a checkout request is non-monotonic because it requires aggregation over an asynchronously computed data set—in general, coordination might be required to ensure that all inputs have been received before the checkout response can be returned to the

```
1  module CartProtocol
2    state do
3      channel :action_msg, [:@server, :session, :op_id] => [:item, :cnt]
4      channel :checkout_msg, [:@server, :session, :op_id] => [:lbound, :addr]
5      channel :response_msg, [:@client, :session] => [:summary]
6    end
7  end

9  module MonotoneReplica
10   include CartProtocol

12   state do
13     lmap :sessions
14   end

16   bloom do
17     sessions <= action_msg do |m|
18       c = LCart.new({m.op_id => [ACTION, m.item, m.cnt]})
19       { m.session => c }
20     end
21     sessions <= checkout_msg do |m|
22       c = LCart.new({m.op_id => [CHECKOUT, m.lbound, m.addr]})
23       { m.session => c }
24     end

26     response_msg <~ sessions do |session, cart|
27       cart.is_complete.when_true {
28         [cart.checkout_addr, session, cart.summary]
29       }
30     end
31   end
32 end
```

Listing 3.6: Cart replica in Bloom^L that supports monotonic (coordination-free) checkout operations.

client. However, observe that the client knows exactly which add and remove operations should be reflected in the result of the checkout. If that information can be propagated to the cart service, any server replica can decide if it has enough information to safely process the checkout operation without needing additional coordination. This design is monotonic: once a checkout response is produced, it will never change or be retracted. Our goal is to translate this design into a monotonic Bloom^L program.

Listing 3.6 contains the server code for this design (we omit the client code for the sake of brevity). Communication with the client occurs via the channels declared in the `CartProtocol` module. Each server replica stores an `lmap` lattice that associates session IDs with `lcart` lattice elements. An `lcart` is a custom lattice that represents the state of a single shopping cart. An `lcart` contains a set of client operations. Each operation has a unique ID; operation IDs are assigned by the client in increasing numeric order without gaps. An `lcart` contains two kinds of operations, *actions* and *checkouts*. An action describes the addition or removal of k copies of an item from the cart. An `lcart` contains at most one checkout operation—the checkout specifies the smallest operation ID that must be reflected in the result of the checkout, along with the address where the checkout response should be sent. The `lcart` merge function takes the union of the operations

in both input carts (operation IDs ensure idempotence). In Listing 3.6, lines 18 and 22 construct `lcart` elements that contain a single action or checkout operation, respectively. These singleton carts are then merged with the previous `lcart` associated with the client's session, if any.

An `lcart` is *complete* if it contains a checkout operation as well as all the actions in the ID range identified by the checkout. Hence, testing whether an `lcart` is complete is a monotone function: it is similar to testing whether an accumulating set has crossed a threshold. Hence, if any server replica determines that it has a complete cart, it can send a response to the client without risking inconsistency. Because this program contains only monotonic operations, CALM analysis can verify that this design is consistent without requiring additional coordination.

Note that the statement that produces a response to the client (lines 26–30) is contingent on having a complete cart. The monotone `summary` method returns a summary of the actions in the cart—an exception is raised if `summary` is called on an incomplete cart. Similarly, attempting to construct an "illegal" `lcart` instance (e.g., an `lcart` that contains multiple checkout operations or actions that are outside the ID range specified by the checkout) also produces an exception, since this likely indicates a logic error in the program.

Discussion

This design is possible because a single client has complete knowledge of the shopping actions in its associated session. Hence, there is no need for additional distributed coordination—the server replicas accumulate knowledge but do not contribute new information themselves. If multiple clients could operate on a single shopping cart, some form of coordination between clients would be needed to ensure a consistent checkout result.

Note that the threshold test for completeness is a crucial part of this design. Until a cart is complete, its content changes in a "non-monotonic" fashion as items are added and removed. However, these non-monotonic changes are hidden inside the `lcart` type and are not directly visible to clients. Clients can only observe the cart's state once the cart is complete; at that point, the cart state is immutable and hence will not change in a non-monotonic fashion. Bloom^L enables `lcart` to expose a limited "safe" interface and to hide transient non-monotonic changes from direct visibility.

One shortcoming of this design is that server replicas may send multiple responses to the client: once a server replica determines that its local state contains a "complete" cart, it can send a response to the client. The client may therefore receive many different (identical) responses—this does not harm correctness but it is clearly not an efficient use of resources. Intuitively, once a copy of the cart has been successfully received by the client, no further copies need to be sent. Unfortunately, adding a rule to implement this logic (e.g., by suppressing checkout responses once an acknowledgment has been received) would be non-monotonic, because it would require either negation or deletion. In Chapter 4, we introduce an automatic program rewrite that solves this problem—Edelweiss would add an acknowledgment protocol to this program and then ensure that acknowledged messages are not retransmitted.

3.6 Related Work

Bloom^L is related to work on concurrency control, distributed storage, parallel programming, and non-monotonic logic.

Semantics-based concurrency control: The traditional correctness criterion for concurrency control schemes is serializability [121]. However, ensuring serializability can be prohibitively expensive, for instance when transactions are long-running or the nodes of a distributed database are connected via an unreliable network. Thus, many methods have been proposed to allow non-serializable schedules that preserve some *semantic* notion of correctness. In particular, several schemes allow users to specify that certain operations can be commuted with other operations; this enlarges the space of legal schedules, increasing the potential for concurrency [55, 62, 155].

O'Neil describes a method for supporting "escrow" transactions, which allow operations that are only commutative when a certain limited resource is available [118]. For example, credits and debits to a bank account might only commute if the bank account balance can be guaranteed to be non-negative. We are currently exploring how to add support for escrow operations to Bloom^L.

To support concurrent editing of shared documents, the groupware community has studied a family of algorithms known as *Operational Transformation* (OT) [54, 145]. OT exploit document semantics to rewrite operations so that, although different sites might observe edits in different orders, each site converges to the same final state. In Chapter 5, we explore the concurrent editing problem in more depth.

Commutativity in distributed systems: Many distributed systems allow users to specify that certain operations are commutative, associative, or idempotent. Helland and Campbell observe that using commutative, associative and idempotent operations is particularly valuable as systems scale and guaranteeing global serializability becomes burdensome [79]. Many distributed storage systems allow users to provide "merge functions" that are used to resolve write-write conflicts between replicas, allowing the system to eventually reach a consistent state (e.g., [85, 103, 129, 147]). Such merge functions typically require knowledge of application semantics to determine when and how concurrent updates can safely be commuted.

Writing a correct merge function is difficult, because the programmer must reason about all possible interleavings of read and write operations. Furthermore, developers must also ensure that application logic respects the loose consistency provided by the storage system. For example, suppose an application reads a value from distributed storage, and then computes and stores a derived value. If a concurrent update is made to the value that was originally read, the merge function will ensure that all replicas of the value will converge, but unless the application logic also recomputes the derived value, overall application state may still be inconsistent. By analyzing application logic in concert with the behavior of the storage layer, Bloom^L and the CALM analysis ensures that replicated values are only used in a "safe" (monotonic) fashion.

Principled eventual consistency: Shapiro et al. recently proposed *Conflict-free Replicated Data Types* (CRDTs), a framework for designing eventually consistent data values [35, 138, 139]. There are two kinds of CRDTs: "operation-based" types (called *CmRDTs*) and "state-based" types (called

CvRDTs). Bloom^L is more closely related to CvRDTs (a CvRDT must be a semilattice), although the two CRDT variants are equivalently expressive. CRDTs and Bloom^L lattice types often use similar design patterns to achieve coordination-free convergence. Unlike Bloom^L, the CRDT approach considers the correctness of replicated values in isolation. This allows CRDTs to be more easily adapted into standalone libraries (e.g., Statebox [85]). However, the narrow focus of CRDTs means that, even if a CRDT is correct, application state may remain inconsistent (Example 1).

Burckhardt et al. advocate using *revision diagrams* to simplify the design of eventually consistent distributed programs [33]. A revision represents a logical replica of system state; revisions can be forked and joined, which represents creating new (logical) replicas and merging replicas together, respectively. Revision a can only be joined to revision b if b is a descendant of the revision that forked a; this constraint ensures that revision graphs are semilattices. Note that this use of lattices differs from Bloom^L: whereas we constrain how data values can change, Burckhardt et al. constrain how replicas can synchronize their states. Another difference is that Burckhardt et al. allow non-deterministic outcomes, whereas we focus on confluence. In subsequent work, Burckhardt et al. have also proposed using abstract data types to encapsulate distributed state [34], although it is unclear whether their language allows user-defined types.

Parallel programming: Lattices have also been applied to languages for parallel programming with shared memory. Kuper and Newton proposed λ_{par}, a deterministic parallel variant of the untyped λ-calculus [92]. Their approach guarantees deterministic program outcomes in the face of concurrent reads and writes to shared state by restricting reads and allowing only monotonically increasing writes to shared variables. Restricted reads in λ_{par} are expressed by supplying a *query set* of lattice elements; a read either blocks or returns a unique element from the query set that is smaller than or equal to the current value of the shared variable (according to the lattice's partial order). This approach to restricting reads is somewhat similar to the use of threshold tests in Bloom^L.

Extending monotonicity in deductive databases: Adding non-monotonic operators to Datalog increases the expressiveness of the language but introduces significant complexities: unrestricted use of non-monotonicity would allow programs that imply logical contradictions (e.g., "p iff $\neg p$") [27]. A simple solution is to disallow recursion through aggregation or negation, which admits only the class of "stratified programs" [15]. Many attempts have been made to assign a semantics to larger classes of programs (e.g., [63, 132, 133, 150]).

In addition to assigning a semantics to programs containing non-monotonic operators, there has also been research on expanding the class of operators considered to be monotone. The prior work most in the spirit of Bloom^L is by Ross and Sagiv on monotonic aggregation [134]. Their framework is broader than the term "aggregation" would suggest. They actually look generally at monotone functions between partially ordered domains, which conceptually includes our lattice least upper bounds, monotone functions and morphisms (though they do not make those distinctions). They extend the notion of monotone programs to include "growth" by substituting a larger element in an atom for a smaller one. As we do, they require the lattice-valued columns to be functionally dependent on the other attributes in a predicate. One of their main results is that if the target domain of a monotone function is a complete lattice, then a program that uses such a function has a unique least fixpoint. While Bloom^L has much in common with Ross and Sagiv, there are several differences.

They want to expand the collection of monotone programs but do not consider confluent distributed programming. In particular, they do not exploit the idempotence of least upper bound (since their source domain need not be a lattice), which is important to us, since it gives confluence with "at-least-once" message delivery. Also, they do not single out monotone functions that are actually morphisms. Hence they do not realize the evaluation flexibility we get from such functions. Finally, they do not propose a practical programming language or a framework that allows user-defined lattices to be composed safely.

Köstler et al. investigate deduction in the presence of a partial ordering of ground atoms, as might be induced by a lattice ordering on predicate columns [90]. They focus on reduced interpretations and models, where any atom dominated by another atom is discarded (unless it belongs to an unbounded maximal chain). One difference between their approach and ours is that reduced models need not satisfy the key dependencies we employ, because a pair of atoms that agree on a key need not have a common least upper bound. Köstler et al. construct a complete lattice over equivalence classes of reduced interpretations. However, the least upper bound of a set of models in this lattice may be infinite, even if each of those models is finite. We suspect our approach and theirs may be isomorphic when the monotone ordering on atoms is induced by lattice-valued columns, where the lattices have no infinite chains.

The approach of Zaniolo and Wang [160] to aggregates in $\mathcal{LDL}$++ is quite distinct from that of Ross and Sagiv, and, hence, from ours. Zaniolo and Wang do not use a lattice as the target domain for an aggregate or other kind of merge function. Rather, they achieve a kind of monotonicity by having an aggregate compute a series of partial results, rather than a single final result. With their formulation, a program computing an average would be judged monotone, where it would not for us. There are several reasons their treatment is not suitable for our purposes. For one, they point out that their programs are not monotone with the inclusion of a predicate for a final result. They do show examples where the partials alone suffice, for example, where the result of an increasing aggregate, such as sum, is compared to an upper bound. However, we see no machinery that would automatically distinguish such a use from an unsafe one, say with average. Further, their use of the `choice()` construct to assign an order to a set so an aggregate function can iterate over it poses two problems for us. One is that they allow different orders to yield different aggregate results; this does not fit with our goal of confluence. More problematic is that in a distributed computation, `choice()` would require communication to obtain a consistent order at different nodes, which goes against our desire for different sites to arrive at the same answer without coordination.

3.7 Discussion and Future Work

A key aspect of Bloom^L is that it enables the *composition* of consistent components. Rather than reasoning about the consistency of an entire application, the programmer can instead ensure that individual lattice methods satisfy *local* correctness properties (e.g., commutativity, associativity, and idempotence). Ensuring that these properties hold for simple lattice types is much simpler than reasoning about the end-to-end behavior of a distributed system. CALM analysis verifies that when these modules are composed to form an application, the complete program satisfies the desired

consistency properties.

Nevertheless, designing a correct lattice type can still be difficult. When developing new lattice types (e.g., the `ldom` lattice described in Section 3.4), we found it useful to write a simple randomized testing framework. Our tool generates a set of random `ldom` values containing a single version-value pair, and then explores all interleavings of merges between these values. Note that violations of the commutativity, associativity, and idempotence properties can be observed without needing domain knowledge about the semantics of the lattice type under test, making our testing tool fairly general-purpose. Nevertheless, considerably more work could be invested to develop a principled test data generation framework for Bloom^L programs, perhaps drawing upon recent work on test generation for Bloom [12]. Another avenue for verifying lattice correctness would be to introduce a restricted DSL for lattice implementations. Such a DSL would likely not need to be Turing complete, which would make formal verification of correctness an easier task.

In this chapter, we have focused on programming with monotonically increasing values. In fact, many distributed programs feature values that increase monotonically for a period but then become immutable. For example, the `lcart` lattice described in Section 3.5 accumulates updates but then eventually becomes "complete" and stops changing. Once a value is immutable, any function can safely be applied to it (whether monotone or not) without risking inconsistency. The `lcart` (and `lbool`) lattices demonstrate that Bloom^L can represent such "monotonic-then-immutable" values, but we suspect that supporting immutability more directly might be useful. For example, a compiler analysis proving that a value is immutable in a certain situation would allow non-monotonic functions to safely be applied to it. Some immutable values can also be represented more efficiently: for example, a complete `lcart` need only store the "summarized" cart state, not the log of client operations.

The ability to commute operations is not always important—commutativity is useful primarily because it allows replicas to safely disagree on the ordering of two operations. This is valuable when agreeing on the order of operations is expensive and new conflicting operations might be introduced concurrently—that is, in the period just after an operation has been initiated in a distributed system. However, after a certain period of time, it is likely that knowledge of the operation has been communicated to all nodes, and hence the position of that operation in a global event order can be agreed upon relatively cheaply: by this point, retaining the metadata needed to allow commutativity is often not useful. An analogy with financial systems is instructive: during the course of a single day, different sites might accept operations (e.g., account debits and credits) which might conflict or violate correctness invariants. During this period, operations might be marked tentative or the system might need to invoke "compensation" logic to recover from conflicts. However, at some point (e.g., the end of the day or the end of the quarter), this uncertainty will be resolved and the "final" state of the system can easily be determined, at which point no future compensation or operation reordering is necessary [79]. The `lcart` lattice is a simple example of this kind of behavior. In the following chapter, we explore automatic program rewrites that exploit a similar pattern: situations in which information is monotonically accumulated for a certain period, but can eventually be safely discarded.

3.8 Conclusion

In this chapter, we proposed Bloom^L, a distributed variant of Datalog that extends logic programming with join semilattices. In so doing, Bloom^L captures and formalizes the common "ACID 2.0" design pattern for writing application logic on top of loosely consistent distributed storage, which makes Bloom^L a useful tool for coordination-free, consistent distributed programming. Like CvRDTs, Bloom^L allows application-specific notions of "progress" to be represented as lattices and goes further by enabling safe mappings between lattices. Bloom^L improves upon our own earlier work by expanding the space of recognizably monotonic programs, allowing more programs to be verified as eventually consistent via CALM analysis. In addition to providing richer semantic guarantees than previous approaches, in our experience Bloom^L provides a natural and straightforward language for building distributed systems.

Chapter 4

Edelweiss: Automatic Distributed Storage Reclamation

"Blossom of snow may you bloom and grow,
bloom and grow forever."
—Oscar Hammerstein, "Edelweiss"

When building loosely consistent distributed systems, a common pattern is to use immutable state whenever possible [46, 59, 77, 111, 112]. This technique avoids the use of mutable shared state, which is widely seen as a common source of problems in distributed programming. Rather than directly modifying shared state, processes instead accumulate and exchange immutable logs of messages or events, a model we call *Event Log Exchange* (ELE): state mutation operations are transformed into an operation (or "event") log. Previously learned information is never replaced or deleted, but can simply be masked by recording new log entries to indicate that the previous information should be ignored.

By using event logs, ELE designs achieve a variety of familiar benefits from database research. For example, rather than determining a conservative global order for modifications to shared state, ELE can allow operations to be applied in different orders at different replicas and reconciled later, reducing the need for coordination and increasing concurrency and availability. ELE allows simple mechanisms for fault tolerance and recovery via log replay, and provides a natural basis for system debugging and failure analysis.

ELE is an attractive approach to simplifying distributed programming, but it introduces its own complexities. If each process accumulates knowledge over time, the required storage will grow without bound. To avoid this, ELE designs typically include a background "garbage collection" or "checkpointing" protocol that reclaims information that is no longer useful. This pattern of logging and background reclamation is widespread in the distributed storage and data management literature, having been applied to many core techniques including reliable broadcast and update propagation [52, 53, 75, 86, 109, 123, 135, 159], group communication [66], key-value storage [5, 58, 159], distributed file systems [72, 102, 127], causal consistency [24, 93, 103], quorum consensus [82], transaction management [3, 37], and multi-version concurrency control [26, 128, 136, 154].

Despite the similarity of these example systems, each design typically includes a storage reclamation scheme that has been developed from scratch and implemented by hand. Such schemes can be subtle and hard to get right: reclaiming garbage too eagerly is unsafe (because live data is incorrectly discarded), whereas an overly conservative scheme can result in hard-to-find resource leaks. Moreover, the conditions under which stored values can be reclaimed depend on program semantics; hence, a hand-crafted garbage collection procedure must be updated as the program is evolved, making software maintenance more difficult.

It would seem that ELE simply trades the difficulties of consistency for a different set of difficulties in space reclamation. In this chapter, we make ELE significantly more attractive by removing the burden of space reclamation from the programmer. We present a collection of program analyses that allow background storage reclamation to be *automatically generated* from program source code.

We introduce Edelweiss, a Bloom sublanguage that omits primitives for mutating or deleting data. Instead, Edelweiss programs describe how local knowledge contributes to the distributed computation. The system computes the complementary garbage collection logic: that is, it automatically and safely discards data that will never be useful in the future.

We validate our work by demonstrating a wide variety of communication and storage protocols implemented as Edelweiss programs with efficient, automatically generated reclamation logic. In this chapter, our demonstrations of Edelweiss include reliable unicast, reliable broadcast, a replicated key-value store, causal consistency, and atomic read/write registers. The garbage collection schemes generated by Edelweiss are often similar to hand-written schemes proposed in the literature for each design. Moreover, removing the need for hand-crafted garbage collection schemes simplifies program design—the resulting programs are more declarative, and the programmer can focus on solving their domain problem rather than worrying about storage.[1]

4.1 Language Definition

Edelweiss is a sublanguage of Bloom that imposes the following restrictions on programs:

1. *Deletion rules cannot be used* (<- operator).
2. *Channel messages are stored persistently*. That is, the lhs of a rule that reads messages from a channel must be persistent.
3. *Channels are derived from persistent collections*. That is, if a channel appears on the lhs of a rule, the rule's rhs must consist of monotone operators over persistent collections.

These conditions ensure that nodes accumulate knowledge over time. Furthermore, once a node decides to send message m to node n, it never "retracts" that decision. Finally, once a node n has received message m, n remembers that message in every subsequent timestep.

These restrictions are natural when building ELE systems that accumulate and exchange immutable values. Nevertheless, Edelweiss would seem to preclude efficient evaluation because nodes only accumulate facts over time. In the remainder of this chapter, we introduce a collection of

[1] An abbreviated version of the material in this chapter appeared in the Proceedings of the VLDB Endowment [41].

Technique	Goal	Requirements	Mechanism	Description
Avoidance of Redundant Messages (ARM)	Avoid sending duplicate messages	Downstream (receiver) logic ignores duplicates	Add logic to send acks; avoid sending ack'ed messages	§4.2
Positive Difference Reclamation (DR+)	Reclaim storage for `X` in `X.notin(Y)`	`X`, `Y` are persistent; logic downstream of `X` is reclaim-safe	Reclaim from `X` upon match in `Y`	§4.2
Negative Difference Reclamation (DR−)	Reclaim storage for `X` and `Y` in `X.notin(Y)`	`X`, `Y` are persistent; logic downstream of `X` and `Y` is reclaim-safe; `notin` quals cover `X`'s keys	Create `range` collection for `X`'s keys; reclaim from `X` and `Y` upon match	§4.4
Range Compression	Store gap-free sequences efficiently	Column values contain one or more gap-free sequences; no non-key columns	`range` collection	§4.2, §4.3
Punctuations [148]	Reduce storage needed for join input collections	Join appears as input to `notin`; punctuation matches join predicate	`sealed` collection, supplied by user, or inferred from rule semantics	§4.3, §4.3, §4.5

Table 4.1: Summary of mechanisms and analysis techniques in this chapter.

mechanisms (Table 4.1) that enable Edelweiss programs to be automatically and safely rewritten into equivalent Bloom programs that use storage efficiently.

4.2 Reliable Unicast

As described in Chapter 2, channels provide asynchronous messaging. Although asynchronous communication matches the capabilities of the physical network, many applications find it convenient to use *reliable unicast*, in which a sender repeatedly transmits a message until it has been acknowledged by the recipient.

Listing 4.1 shows a naive reliable unicast program. Each message contains a unique ID, destination address, and payload. The `sbuf` collection is the sender-side buffer; communication is expressed by copying `sbuf` into the `chn` channel (line 11); the recipient persists delivered messages in `rbuf` (line 12). Note that because `sbuf` is persistent (as declared on line 6), Bloom's semantics [13]

```
1  class Unicast
2    include Bud

4    state do
5      channel :chn, [:id] => [:@addr, :val]
6      table :sbuf, [:id] => [:addr, :val]
7      table :rbuf, sbuf.schema
8    end

10   bloom do
11     chn  <~ sbuf
12     rbuf <= chn
13   end
14 end
```

Listing 4.1: Naive reliable unicast in Edelweiss.

dictate that a new `chn` message will be sent for every timestep at the sender.

Although it is concise and declarative, the naive reliable unicast program has two obvious shortcomings. First, an unbounded number of `chn` messages are derived. Although inefficient, this is not incorrect: Bloom collections have set semantics and the receiver-side logic is *idempotent*, which means that delivering the same message more than once has no effect. Second, the sender-side buffer `sbuf` grows without bound. This is unnecessary in practice: once a message has been successfully delivered to the recipient, it need not be retained by the sender.

We could address both problems by making the program more complex—for example, by arranging for receivers to emit acks and for senders to delete acknowledged messages. However, *these modifications would not change the user-visible behavior of the program*! Acks and storage reclamation are not necessary for correctness—rather, they are only needed to help ensure that resources are used efficiently. We would like the best of both worlds: a concise, declarative program that has an efficient implementation. In the remainder of this section, we introduce a series of techniques that achieve this goal by allowing acks and storage reclamation to be introduced by safe, automatic program transformations from Edelweiss to an equivalent Bloom program.

Avoidance of Redundant Messages (ARM)

We begin by detailing *ARM*, an automatic program rewrite that avoids redundant communication between nodes. This requires identifying when delivering a message multiple times is redundant, and then rewriting the program to avoid duplicate transmissions.

In Edelweiss, detecting when duplicate channel deliveries are redundant is simple: the restrictions in Section 4.1 imply that once *any* message has been delivered, the receiver will persist it and subsequent attempts to send that message can safely be suppressed. Therefore, we can rewrite every Edelweiss rule that inserts messages into a channel to avoid inserting duplicates. For the program in Listing 4.1, avoiding duplicates would be easy if the sender could directly access the receiver's buffer:

```
chn <~ sbuf.notin(rbuf)
```

```
1  class UnicastAckRewrite
2    include Bud

4    state do
5      channel :chn, [:id] => [:@addr, :val]
6      table :sbuf, chn.schema
7      table :rbuf, chn.schema
8      table :chn_approx, [:id]
9      channel :chn_ack, [:@sender, :id]
10   end

12   bloom do
13     chn  <~ sbuf.notin(chn_approx, :id => :id)
14     rbuf <= chn
15     chn_ack <~ chn {|c| [c.source_addr, c.id]}
16     chn_approx <= chn_ack.payloads
17   end
18 end
```

Listing 4.2: Reliable unicast with acknowledgments; ARM-generated code is italicized.

This approach is not possible because Edelweiss nodes can only communicate via message passing. However, a simple variant is possible: receivers can inform senders about messages that have been successfully delivered. Because such communication is asynchronous, the sender will only have a lower bound on the receiver's state—but since the receiver ignores duplicate messages anyway, this does not harm correctness.

There are many ways in which senders can learn a conservative estimate of the receiver's state, such as cumulative, timer-based acks (as in TCP) or "piggybacking" acks onto normal message traffic. For programs involving multiple senders and receivers, even more strategies are possible, such as epidemic gossip [52] or tree-based multicast. Any of these schemes could be used by ARM, since they all accomplish the same purpose of allowing senders to lower-bound the receiver's state. For unicast delivery, a simple scheme suffices: a receiver sends an ack whenever they receive a message.

The result of applying ARM to the naive unicast program is shown in Listing 4.2. ARM automatically introduces a new channel (line 9), which is used to send acks upon successful receipt of a `chn` message (line 15). Senders persist acks (line 16). Finally, ARM rewrites the rule that sends `chn` messages to avoid sending acknowledged messages (line 13). Note that ARM automatically infers that acks only need to contain message IDs (line 15), not the entire message. This is possible because `id` is the key of `chn` (line 5), which means that a given ID is associated with exactly one message.

Positive Difference Reclamation (DR+)

The ARM rewrite allows the simple unicast program in Listing 4.1 to avoid sending an unbounded number of messages, but the rewritten program in Listing 4.2 still does not reclaim acknowledged messages from `sbuf`. In fact, the program's storage consumption has grown because the sender also persists the `chn_approx` collection. In this section, we introduce *DR+*, a program rewrite that

automatically and safely reclaims storage, and show how it can be applied to sbuf; we address chn_approx in the following section.

DR+ exploits the semantics of set difference, which is expressed in Bloom using the notin operator. Consider X.notin(Y), where X and Y are persistent. Recall that X and Y are the "positive" and "negative" inputs to the notin, respectively: as new tuples arrive in Y, any matching tuples in X will no longer appear in the output of the notin. Moreover, because Y is persistent, any X tuple that has a match in Y will *never* appear in the output of the notin again. For the purposes of this rule, X tuples with matches in Y will never contribute to program outcomes and no longer need to be stored.

To reclaim from X, Edelweiss needs to prove that doing so will not change the output of any other rule that references X. In many cases, this is easy to do: for example, if X appears on the rhs of a projection or selection rule with a persistent collection on its lhs, we can reclaim from X (intuitively, the rule makes a persistent "copy" of the X tuple). When X appears in more than one set difference rule, we can reclaim X tuples when the *conjunction* of the reclamation conditions of the rules are satisfied; we give an example in Section 4.4. Finally, reclaiming from collections that appear in joins is more complicated; we discuss this scenario in Section 4.5.

Returning to the reliable unicast example, we observe that sbuf is only referenced by a single rule, where it is used as the positive input to a notin operator (line 13 in Listing 4.2). Because chn_approx is persistent, DR+ will reclaim sbuf tuples that have matches in chn_approx. This is done by adding this rule to the program:

```
sbuf <- (sbuf * chn_approx).lefts(:id => :id)
```

The rhs computes the equijoin of sbuf and chn_approx on id and returns the left join input (tuples from sbuf); the <- operator removes the resulting sbuf tuples. This rule corresponds to our intuition that once the recipient has acknowledged successful delivery of a message, the message can safely be discarded by the sender.

Note that ARM and DR+ are independent program rewrites, but they work together profitably: ARM introduces set difference operations and DR+ exploits the semantics of set difference to safely and automatically reclaim storage.

Range Compression

Lastly, we need to address the storage used by the chn_approx collection. Unfortunately, DR+ is not useful because chn_approx does not appear as the positive input to a notin operator. Moreover, reclaiming tuples from chn_approx is problematic in principle: if we deleted such tuples, we would have no information at the sender to prevent redelivering acknowledged sbuf messages in the future.

Rather than reclaiming from chn_approx, can we instead represent the entire collection using a small amount of storage? Fortunately, this is feasible: recall that chn_approx only contains a single column, the message ID. Since IDs are assigned by a single sender, the sender can choose IDs from a gap-free, totally ordered sequence such as the natural numbers starting at some constant k. Because we expect all messages to eventually be delivered, chn_approx will eventually contain

all the IDs from k to n. Hence, our task is much easier: we need to represent $\{k, \ldots, n\}$, which we can do by storing the smallest and largest elements of the set.

However, some elements of the set $\{k, \ldots, n\}$ might be missing from `chn_approx` at any given time. Hence, rather than a single pair $[k, n]$, we use a set of pairs $\{[k_0, k_1], \ldots, [k_m, k_n]\}$; each pair efficiently represents a gap-free range of numbers, while missing IDs are represented by gaps between the "high" element of one pair and the "low" element of the next. This data structure is a 1-dimensional *range tree* [25]; we call the compression technique it allows *range compression*.

Range compression can be viewed as a generalization of the "low water mark" used by reliable delivery schemes such as TCP, in which senders assign sequence numbers to packets and receivers send acks to indicate the prefix of the sequence they have received. Rather than requiring programmers to manipulate sequence numbers and use integer inequality, range compression achieves a similar degree of efficiency while allowing the program to deal with an unordered set of events. This has two benefits: first, range compression automatically handles situations in which IDs are omitted or delivered out-of-order, without requiring the programmer to explicitly track a "low water mark." Second, set-oriented programs are convenient to develop, particularly in set-oriented languages such as Bloom.

In the current Edelweiss prototype, developers explicitly enable range compression by using a new collection type, `range`. For example, "`range :chn_approx, [:id]`" would replace line 8 in Listing 4.2. In addition, the Edelweiss runtime automatically applies range compression to outbound channel messages when profitable—this allows a single acknowledgment to describe the successful delivery of many `chn` messages. It would be possible to apply range compression to all collections by default, but we haven't found the need to implement this yet.

4.3 Reliable Broadcast

In the previous section, we showed how a declarative Edelweiss program for reliable unicast can be implemented efficiently. In the following sections, we show how the same techniques can be applied to a series of more complicated Edelweiss programs. We begin by generalizing reliable unicast to reliable broadcast and then in Section 4.4 we use reliable broadcast to build a replicated key-value store. In Section 4.5, we then extend the key-value store to provide causal consistency guarantees. Finally, Section 4.6 discusses how to implement atomic read/write registers. Importantly, all of these programs can be written in Edelweiss and implemented efficiently via extended versions of the techniques introduced in Section 4.2.

Fixed Membership

Listing 4.3 shows a naive reliable broadcast program. Any node can send a message by inserting into the `log` collection. The messages in the log are sent to every node in the group (line 11).[2] When a node receives a message, it adds the message to its log (line 12); that node will re-broadcast

[2]Note that + concatenates tuples. Broadcast is expressed as Cartesian product—i.e., a join between `node` and `log` with no join predicate.

```
1  class Broadcast
2    include Bud

4    state do
5      sealed :node, [:addr]
6      table :log, [:id] => [:val]
7      channel :chn, [:@addr, :id] => [:val]
8    end

10   bloom do
11     chn <~ (node * log).pairs {|n,l| n + l}
12     log <= chn.payloads
13   end
14 end
```

Listing 4.3: Reliable broadcast to a fixed set of nodes.

the message in the future. Each message has a unique ID. To assign unique IDs without global coordination, a common technique is to use ⟨*node-id*, *seqnum*⟩ pairs, where *seqnum* is a node-local sequence number.[3] We assume the broadcast group contains a fixed set of nodes; we relax this assumption in Section 4.3.

The program in Listing 4.3 is simple—indeed, it closely resembles the pseudocode for the textbook reliable broadcast algorithm [116]—but as with naive reliable unicast (Listing 4.1), it suffers from unbounded messaging and storage.

Bounded Messaging

As with reliable unicast (Section 4.2), the ARM rewrite automatically avoids unbounded messaging by inserting an acknowledgment protocol. We omit the rewritten program for space reasons, but the same acking scheme can be used. To avoid sending acknowledged messages, line 11 is rewritten to:

```
chn <~ (node * log).pairs {|n,l| n + l}.notin(chn_approx, 0 => :addr, 1 => :id)
```

The `notin` predicate checks for an equality match between the first two columns of the join result against the `addr` and `id` fields of `chn_approx`. Note that unlike with reliable unicast, acks include node addresses as well as message IDs. This is necessary because a message might be delivered successfully to some nodes but not others; moreover, ARM deduces this automatically because the key of `chn` contains both fields (line 7).

Acknowledgments and Logical Clocks

After applying ARM, each node persists two collections that grow over time: `log`, the set of messages, and `chn_approx`, which holds each node's knowledge about the messages that have been received by the other nodes. The storage required for `chn_approx` can be reduced via

[3] We implemented ⟨*node-id*, *seqnum*⟩ pairs as a single 64-bit integer consisting of a 32-bit node ID prepended to a 32-bit sequence number. This is compatible with range compression, since multiple IDs generated by the same node will form a gap-free sequence.

range compression, as described in Section 4.2. Recall that for reliable unicast, chn_approx will eventually be range-compressed to a single value, effectively yielding a logical clock. With broadcast, chn_approx will contain a single "clock" value for each node in the broadcast group and hence behaves similarly to a vector clock [113]. That is, the combination of ARM and range compression essentially "discovers" the relationship between event histories [137] and logical clocks! Edelweiss allows programmers to simply manipulate sets of immutable events; it then automatically produces the corresponding "clock" management code.

Punctuations

To reclaim messages from log, we can use the DR+ rewrite introduced in Section 4.2. However, reclaiming broadcast messages is more complicated than reclaiming unicast messages—intuitively, a unicast message can be reclaimed as soon as it has been successfully delivered to the recipient, whereas a broadcast message can only be reclaimed once it has been delivered to *every* node in the group. This difference is manifest in the program:

```
chn <~ (node * log).pairs {|n,l| n + l}.notin(chn_approx, 0 => :addr, 1 => :id)
```

As chn_approx grows, it matches tuples in the *output* of the join between node and log; our goal is to use tuples in chn_approx to reclaim from the join's *input* collections. To do so, Edelweiss must reason about how the join's inputs can grow over time. For example, to reclaim a tuple t from log, Edelweiss must ensure that all future join outputs that depend on t have already been produced and that all such output tuples have a match in chn_approx.

This can be done by adapting the concept of *punctuations*, which were first introduced for processing queries over unbounded data streams [148]. A punctuation is a guarantee that no more tuples matching a predicate will appear in a collection. For now, we consider a simple class of punctuations: the assertion that no more tuples will ever appear in a collection. Given (X * Y).notin(Z), suppose we want to reclaim a tuple $y \in Y$. A punctuation asserting that no more X tuples will arrive implies that we know about all the X tuples that will ever match y. Hence, once we have seen a match in Z for all the current join results that depend on y, y can safely be reclaimed. Of course, the symmetry of the join operator means that a similar argument allows reclamation from X given a punctuation on Y.

Returning to reliable broadcast, Edelweiss can reclaim tuples from log given a punctuation that no new node tuples will appear. At the beginning of this section, we assumed that the broadcast group is fixed—hence, the necessary punctuation can safely be produced. As a syntactic convenience, Edelweiss defines a new collection type called sealed to hold a collection whose contents are fixed after the system has been initialized. Declaring that node is sealed (line 5 in Listing 4.3) allows the Edelweiss runtime to automatically emit a punctuation for the collection. The rewritten program produced by Edelweiss can be found in Listing A.5, but we give the main idea here: punctuations are represented by tuples in "seal tables" that are defined automatically by Edelweiss. The rules generated by DR+ join against the seal table for node and thereby wait for a punctuation on node before reclaiming any tuples from log. Hence, by exploiting the fact that node is sealed, DR+ confirms our intuition that messages can safely be reclaimed once they have been successfully delivered to all nodes.

Dynamic Membership

Assuming a fixed broadcast group simplifies deciding when a log entry can be reclaimed. If we relax this assumption (by declaring that `node` is a `table` in line 5 of Listing 4.3), DR+ can no longer reclaim tuples from `log`. Indeed, reclaiming from `log` would be unsafe: if a new tuple appeared in `node`, the join between `node` and `log` (line 11) implies that *all* `log` messages should be delivered to the newly joined node. Hence, reclaiming from `log` would change user-visible program behavior.

To allow both dynamic membership and safe reclamation from the `log` table, we need to change the program to identify situations in which `log` messages should *not* be delivered to new nodes; such messages can then be reclaimed. We can achieve this using *epochs*: each epoch has a fixed set of members and each message identifies the epoch to which it belongs. To change the membership of the broadcast group, the system moves to a new epoch with a different set of members. Hence, once the membership of an epoch has been fixed and a message has been delivered to all the members of that epoch, that message can safely be reclaimed.

Listing 4.4 contains an Edelweiss program implementing this design; the corresponding code produced by the Edelweiss compiler can be found in Listing A.6. Note that this program is nearly identical to reliable broadcast with fixed membership, except that the Cartesian product between `node` and `log` (line 11 in Listing 4.3) has been replaced with an equijoin on `epoch` (line 11 in Listing 4.4). DR+ automatically exploits the equijoin predicate to enable reclamation using finer-grained punctuations: given `(X * Y).pairs(:k1 => :k2).notin(Z)` and a punctuation that asserts that no more `Y` tuples will arrive with $k2 = c$, all `X` tuples with $k1 = c$ are now eligible for reclamation. In general, DR+ can exploit punctuations that match the join predicate; since a Cartesian product is essentially a join with no predicate, DR+ can only use whole-relation punctuations for such operators.

Applying DR+ to the epoch-based broadcast program produces the expected results: given a punctuation asserting that no more `node` facts will be observed for epoch k, the rules produced by DR+ automatically reclaim any message in epoch k that has been delivered to all the members of that epoch. The procedure for deciding to move to a new epoch is orthogonal to this program; a common approach is to use a separate (and more expensive) protocol based on distributed consensus [29, 117]. After a new epoch has been decided on, the consensus mechanism would then broadcast a corresponding punctuation, allowing Edelweiss to reclaim messages.

DR+ also allows reclamation from `node` using punctuations on `log`: if we can guarantee that no more `log` facts will arrive for a given epoch, then once every `log` fact in that epoch has been delivered to some node n, n can be reclaimed from `node`. This follows from the symmetry of the join predicate on line 11.

```
1  class BroadcastEpoch
2    include Bud

4    state do
5      table :node, [:addr, :epoch]
6      table :log, [:id] => [:epoch, :val]
7      channel :chn, [:@addr, :id] => [:epoch, :val]
8    end

10   bloom do
11     chn <~ (node * log).pairs(:epoch => :epoch) {|n,l| [n.addr] + l}
12     log <= chn.payloads
13   end
14 end
```

Listing 4.4: Reliable broadcast with epoch-based membership

```
1  class KvsReplica
2    include Bud

4    state do
5      sealed :node, [:addr]
6      channel :ins_chn, [:@addr, :id] => [:key, :val]
7      channel :del_chn, [:@addr, :id] => [:del_id]
8      table :ins_log, [:id] => [:key, :val]
9      table :del_log, [:id] => [:del_id]
10     scratch :view, ins_log.schema
11   end

13   bloom do
14     ins_chn <~ (node * ins_log).pairs {|n,l| n + l}
15     del_chn <~ (node * del_log).pairs {|n,l| n + l}
16     ins_log <= ins_chn.payloads
17     del_log <= del_chn.payloads
18     view <= ins_log.notin(del_log, :id => :del_id)
19   end
20 end
```

Listing 4.5: Key-value store based on reliable broadcast.

4.4 Key-Value Store

In this section, we use reliable broadcast to build a replicated key-value store (KVS).[4] We show that Edelweiss automatically produces a safe, effective storage reclamation scheme for this program.

Using reliable broadcast to build a KVS is a well-known technique [58, 86, 159]. The store contains a set of *keys* and associated *values*. Clients submit *insert* and *delete* operations; replicas apply these operations to maintain their local *view*. Each insert operation has a unique ID and a

[4]The KVS design presented in this section is significantly different from the lattice-based KVS discussed in Section 3.4. The KVS in Section 3.4 maps a key to a single lattice value and does not directly support deletions; whereas the KVS we discuss in this section accumulates a log of both insert and delete operations, allows multiple values with the same key, and computes the current set of "live" key-value pairs dynamically.

delete operation contains the ID of its corresponding insertion. Following prior work [159], we allow multiple insertions of the same key with different IDs; all such key-value pairs are included in the view. Hence, if a key appears multiple times, a given deletion applies to only one of the IDs associated with that key. The KVS is fully replicated. Building this design using reliable broadcast is straightforward: a log of insert and delete operations is broadcast to all nodes, and the set of live keys at any given replica consists of every insertion that has no matching deletion. Listing 4.5 contains a simple Edelweiss program that implements this scheme.

A natural question is how to bound the storage required for operation logs. In prior work [58, 75, 109, 159], researchers proposed hand-crafted protocols that allow safe reclamation by tracking each node's knowledge of the state of the other nodes. We show how a similar scheme can safely and automatically be produced by Edelweiss from the simple, declarative program in Listing 4.5.

Reclaiming Insertions

Because insert and delete operations are used differently, we need to employ two different reclamation strategies. Inserts are used in two places: the broadcast rule (line 14) and the rule to compute the current view (line 18). Applying ARM to the broadcast rule, we get:

```
ins_chn <~ (node * ins_log).pairs {|n,l| n + l}.notin(ins_chn_approx, 0 => :addr, 1 => :id)
```

Observe that `ins_log` only appears as the positive input to two `notin` operators, which makes it a candidate for the DR+ rewrite. As discussed in Section 4.2, we can reclaim from a collection when the absence of a tuple from the collection would not change user-visible program behavior. In this case, an `ins_log` tuple can be discarded when it has a match in both `ins_chn_approx` and `del_log`—since both of those collections are persistent, we know that such an `ins_log` tuple will never contribute to the results of either `notin` ever again. Hence, Edelweiss will safely and automatically reclaim an insertion once (a) it has been delivered to every node, and (b) it has been deleted.

Reclaiming Deletions

DR+ cannot reclaim tuples from `del_log` because it appears as the negative input to a `notin` operator (line 18). We encountered a similar situation with the `chn_approx` collection in reliable unicast (Section 4.2). In that case, range compression was used to store `chn_approx` efficiently, because `chn_approx` will eventually contain the complete set of message IDs and senders can choose IDs from a gap-free, ordered sequence. Unfortunately, the set of deleted IDs is likely to contain many gaps, rendering range compression ineffective. Hence, a new program transformation is needed to reclaim from `del_log`. We first consider the conditions that must be satisfied to allow deletions to be reclaimed, and then generalize this reasoning into an automatic program rewrite.

In the program in Listing 4.5, Edelweiss can determine that each deletion matches at most one insert operation; this is implied because the `notin` matches `ins_log.id` with `del_log.del_id` (line 18) and `id` is a key of `ins_log` (line 8). Hence, once a `del_log` entry d has been matched to an insertion i, we know that *no other insertion* will ever match d. Hence, we might be tempted to

conclude that both *d* and *i* can be discarded, but that would be mistaken: if another copy of *i* appears in `ins_log` (e.g., because the network delivers a duplicate message), *d* will have been reclaimed from `del_log` and hence *i* will incorrectly be included in the replica's view.

Thus, when a replica observes an insertion *i* that matches a deletion *d*, both *i* and *d* can be reclaimed if we can guarantee that *i* will never appear in `ins_log` again. Fortunately, this can be done relatively cheaply: `id` is a key of `ins_log` and we have already explained how range compression can be used to represent the set of all message IDs efficiently (Section 4.2). Edelweiss exploits this fact to store the set of all insertion IDs witnessed by a replica separately, and then only add new insertions to `ins_log` if the insert's ID has not been observed before. In effect, Edelweiss automatically rewrites the program to split `ins_log` into two pieces: the set of insert IDs, which is range-compressed, and the rest of the data associated with each insertion, which is reclaimed when a matching deletion is observed.

Edelweiss extends these ideas into an automatic program rewrite called *Negative Difference Reclamation* (DR−). The rewrite can be applied to expressions of the form `X.notin(Y, :A => :B)`, where `X` and `Y` are persistent and `A` is a key of `X`. The program is rewritten as follows:

1. A new `range` collection is added, `X_keys`; this stores all the key values that have ever been observed for `X`.
2. A rule is added to update `X_keys` as new tuples appear in `X`.
3. Every rule that adds new tuples to `X` is rewritten to include a negation against `X_keys`; that is, prospective `X` tuples whose keys are found in `X_keys` are ignored.
4. Rules are added to reclaim matching tuples from `X` and `Y`. Note that we do *not* reclaim from `X_keys`.

DR+ and DR- are complementary, in that DR+ is effective when the negative input to the `notin` can be range compressed, whereas DR− requires that the keys of the positive input be suitable for range compression. As a heuristic, we use the collection type of the negative `notin` input to decide whether to apply DR+ or DR− (that is, DR− is not applied if the inner input is a `range` collection).

4.5 Causal Consistency

The key-value store presented in Section 4.4 ensures replica convergence but does not provide any guarantees about the *consistency* of the view presented by a replica at any time. Many consistency guarantees have been proposed; recently, several researchers have argued that causal consistency is a good fit for scalable distributed storage [20, 24, 93, 103, 123]. In this section, we use Edelweiss to implement a causally consistent KVS and show how the metadata required for causal consistency is automatically and safely reclaimed.

Background

A causally consistent system respects the causal relationships between operations. Causality is represented as a partial order over operations: *a* "happens before" *b* (written $a \rightsquigarrow b$) if operation *a* could have "caused" or influenced operation *b* [95]. For example, if a client reads a version of

key x that was produced by write w_x and then submits write w_y to key y, $w_x \rightsquigarrow w_y$. In a causally consistent system, a replica's view will only include w_y if it also includes w_x.

A common approach to implementing causal consistency is to annotate each operation with the operations upon which it depends (e.g., w_y depends on w_x in the example above). Before a write can be applied to a replica's view, the replica must first have applied all of the write's dependencies; similarly, a replica can only respond to a read operation when the replica's view reflects all of the read's dependencies. In some systems, dependencies between operations are tracked automatically (e.g., by a client-side library) [103], whereas in other designs, users specify dependencies explicitly [20, 93]. We assume each operation is provided along with its dependencies, which is compatible with either scheme.

An operation is *safe* at a replica if the replica contains all of the operation's (transitive) dependencies. A write operation w to key k is *dominated* if there is a safe write operation w' for key k such that $w \rightsquigarrow w'$. That is, w is dominated if there is another write w' to the same key that has w as a dependency, either directly or transitively. Each replica's view should reflect all the safe, undominated writes it has observed. If there are two writes to the same key and neither dominates the other, the writes are *concurrent*. Some systems handle this situation by invoking a commutative merge function [103, 123]. We include both versions of the key in the view; a client can then read both versions and resolve the conflict by issuing a new write that dominates both previous versions of the key.

Write Operations

To extend the key-value store presented in Section 4.4 to support causal consistency, we begin by considering how to support write operations. As in the simple KVS, each replica broadcasts its log of write operations to the other replicas. However, replicas may need to buffer writes they receive until the dependencies of those writes have been satisfied. Listing 4.6 shows an Edelweiss program fragment that implements this scheme in lines 14–18. Each log entry has a set of dependencies (represented as a nested array in the `deps` column), and log entries are moved from `log` to `safe` when their dependencies are met. The `flat_map` method (line 15) is used to "unnest" the array in the `deps` column.

A replica's view should contain all the safe, undominated writes it has observed, so next we need to determine which writes in `safe` have been dominated. Our initial implementation looked for paths in the transitive closure of the dependency graph—that is, w dominates w' if $w.key = w'.key$ and there is a "path" of transitive dependencies from w that eventually reaches w'. While this design was correct, it prevented dominated writes from being reclaimed by Edelweiss. On closer examination, we realized that in this scheme, reclaiming dominated writes is not permissible because a dominated write to key k might be needed to compute the transitive dependencies of another write k' on a different key. Hence, Edelweiss taught us something surprising about our own program!

In recent work [103], Lloyd et al. avoid the need to retain the complete dependency graph by requiring that a write to key k must include a dependency on a previous write to k (if any exists). We make the same assumption in Listing 4.6; hence, we can identify dominated writes by looking for another write to the same key that includes the dominated write as a *direct* dependency (lines 20–22).

```
1  state do
2    table :log, [:id] => [:key, :val, :deps]
3    table :safe, [:id] => [:key, :val]
4    table :dep, [:id, :target]
5    range :safe_keys, [:id]
6    table :safe_dep, [:target, :src_key]
7    table :dom, [:id]
8    scratch :pending, log.schema
9    scratch :missing_dep, dep.schema
10   scratch :view, safe.schema
11 end

13 bloom do
14   pending <= log.notin(safe_keys, :id => :id)
15   dep <= log.flat_map {|l| l.deps.map {|d| [l.id, d]}}
16   missing_dep <= dep.notin(safe_keys, :target => :id)
17   safe <+ pending.notin(missing_dep, 0 => :id) .map {|p| [p.id, p.key, p.val]}
18   safe_keys <= safe {|s| [s.id]}

20   safe_dep <= (dep * safe).pairs(:id => :id) {|d,s| [d.target, s.key]}
21   dom <+ (safe_dep * safe).lefts(:target => :id, :src_key => :key)
22                           {|d| [d.target]}.notin(dom, 0 => :id)
23   view <= safe.notin(dom, :id => :id)
24 end
```

Listing 4.6: Causal consistency for write operations.

Edelweiss generates an effective reclamation scheme for this program. As expected, `log` entries are reclaimed once they have been delivered to all replicas and their dependencies have been met at the local replica. Tuples in `dep` can be reclaimed once their associated log entry is safe. Interestingly, `safe_dep` and `dom` facts can be reclaimed as soon as they are produced: while logically the set of dominated writes grows over time, Edelweiss observes that a dominated write is only needed to remove tuples from `safe`. Hence `dom` and `safe_dep` facts can be immediately reclaimed.

Facts in `safe` can be reclaimed once they have been dominated. Note that `safe` appears in two joins (lines 20 and 21); Edelweiss must determine that reclaiming from `safe` will not change the results of either join. In general, this might require punctuations on the other join input, but Edelweiss supports several special cases that avoid the need for user-supplied punctuations for this program. On line 20, `dep` is produced by a `flat_map` operation involving the key of `log`; hence, Edelweiss can infer punctuations on the first column of `dep`. This matches our intuition that no new dependencies will be observed for a given write. Similarly, the join on line 21 matches the key column of `safe` with the key columns of `safe_dep`; hence, once a `safe` tuple s has a match in `safe_dep`, Edelweiss knows that no other join results will depend on s.

Read Operations

The KVS in Section 4.4 does not explicitly support read operations: each replica uses the operation log to compute the `view` collection, and clients read by (implicitly) examining the replica's current view. To implement causal consistency for reads, we first need to represent read operations explicitly.

```
state do
  table :read_buf, [:id] => [:key, :deps, :src_addr]
  scratch :read_pending, read_buf.schema
  scratch :read_dep, [:id, :target]
  scratch :missing_read_dep, read_dep.schema
  scratch :safe_read, read_buf.schema
  table :read_resp, resp_chn.schema
end

bloom do
  read_buf <= req_chn {|r| [r.id, r.key, r.deps, r.source_addr]}
  read_pending <= read_buf.notin(read_resp, :id => :id)
  read_dep <= read_pending.flat_map {|r| r.deps.map {|d| [r.id, d]}}
  missing_read_dep <= read_dep.notin(safe_keys, :target => :id)
  safe_read <+ read_pending.notin(missing_read_dep, 0 => :id)
  read_resp <= (safe_read * view).pairs(:key => :key) do |r,v|
    [r.src_addr, r.id, r.key, v.val]
  end
  resp_chn <~ read_resp
end
```

Listing 4.7: Causal consistency for read operations.

In Listing 4.7, a client initiates a read request by sending a message over the `req_chn` channel; when the read's dependencies have been satisfied, the replica responds via the `resp_chn` channel. If a read request is unsafe (i.e., if it specifies dependencies that are not satisfied by the local replica), the replica buffers the request until its dependencies have been met.

Edelweiss provides several features that simplify this program. First, ARM prevents unbounded `resp_chn` messages by inserting client-side acknowledgment logic (we omit the client code for brevity). Second, DR+ reclaims `read_buf` messages when the read's dependencies have been satisfied. Finally, DR− reclaims `read_resp` tuples when the client's acknowledgment is received. Perhaps more importantly, Edelweiss automatically handles the interactions between the reclamation conditions of all these rules and the safety and dominance rules in Listing 4.6, allowing the developer to focus on implementing correct application-level behavior.

4.6 Read/Write Registers

In this section, we use Edelweiss to implement *atomic read/write registers* [96], a common building block for distributed algorithms. An atomic register allows a single writer to interact with multiple concurrent reader processes and guarantees that the values returned by reads are consistent with a serial ordering of operations. Reading an atomic register reflects the latest value written (in contrast to the KVS presented in Section 4.4, in which each insertion for a given key adds to its list of values).

While traditional designs utilize mutable storage, we show how a mutable register interface can be implemented via an Edelweiss program that accumulates an immutable event log. We then extend the program to support atomic writes to multiple registers and multi-register reads that

```
1  class AtomicRegister
2    include Bud

4    state do
5      table :write, [:wid] => [:name, :val]
6      table :write_log, [:wid] => [:name, :val, :prev_wid]
7      table :dom, [:wid]
8      scratch :write_event, write.schema
9      scratch :live, write_log.schema
10   end

12   bloom do
13     write_event <= write.notin(write_log, :wid => :wid)
14     write_log <+ (write_event * live).outer(:name => :name) do |e,l|
15       e + [l.wid.nil? ? 0 : l.wid]
16     end
17     dom <= write_log {|l| [l.prev_wid]}
18     live <= write_log.notin(dom, :wid => :wid)
19   end
20 end
```

Listing 4.8: Atomic read/write registers.

reflect a consistent snapshot. For all of these programs, Edelweiss enables automatic and safe garbage collection, generating space-efficient implementations that are semantically equivalent to the original programs.

Single-register Writes

Listing 4.8 contains an implementation of atomic registers in Edelweiss. As in the KVS program (Section 4.4), the current register values are computed as a view over an append-only event log.

The collection `write_log` records the history of writes to a set of registers. The atomic register model assumes a single writer per register [96], so each entry in `write_log` for a particular register has exactly one predecessor—the previously written value—which it supersedes. The `dom` table contains those write IDs that are dominated by a "more recent" entry in `write_log` (line 17)—i.e., those IDs that are the predecessor (or `prev_wid`) of another record. Line 18 defines the view `live` as the subset of records in `write_log` that are *not* dominated by a more recent record. The single writer assumption implies that `live` contains exactly one record at any time for a given register.

To write to a register, a client inserts into the `write` collection. If the write has not yet been applied to the write log, this generates a `write_event` (line 13). We then insert a new record into `write_log` containing the new value along with the ID of the previous write to that register, or 0 if this is the first write (lines 14–16).

The program has three persistent collections: `write`, `write_log`, and `dom`. Edelweiss uses DR+ to reclaim records from `write` as soon as they are reflected in `write_log`. DR- can be used to reclaim from both `write_log` and `dom`. As discussed in Section 4.4, DR- exploits the fact that the `notin` predicate on line 18 only uses the key column of `write_log`. Hence, Edelweiss knows that once a `write_log` entry has been dominated, both the `write_log` and `dom` facts can

```
1  class AtomicBatchWrites
2    include Bud

4    state do
5      table :write, [:wid] => [:batch, :name, :val]
6      table :write_log, [:wid] => [:batch, :name, :val, :prev_wid]
7      table :commit, [:batch]
8      table :dom, [:wid]
9      scratch :live, write_log.schema
10     scratch :commit_event, write.schema
11   end

13   bloom do
14     commit_event <= (write * commit).lefts(:batch => :batch)
15                                     .notin(write_log, 0 => :wid)
16     write_log <+ (commit_event * live).outer(:name => :name) do |e,l|
17       e + [l.wid.nil? ? 0 : l.wid]
18     end
19     dom <= write_log {|l| [l.prev_wid]}
20     live <= write_log.notin(dom, :wid => :wid)
21   end
22 end
```

Listing 4.9: Atomic registers supporting multi-register writes.

be reclaimed—as long as we can prevent any duplicate `write_log` tuples from appearing. To enable this, Edelweiss automatically creates a range collection that stores all the write IDs ever observed—fortunately, this set can be effectively range compressed.

Multi-register Writes

Next, we show how to support atomic updates to multiple keys (Listing 4.9). That is, we allow writes to be supplied for multiple keys and then eventually *committed*, at which point all the associated writes are applied atomically.

Clients insert values into `write` as before, but these values are not applied to `write_log` until a `commit` record exists for their batch (lines 14–15). As in the atomic register program, we assume that at most one value for every register exists in `commit_event` at any time. For each such register, a fact is inserted into `write_log` reflecting the last effective write for that register (`live.wid`) as its `prev_wid` (lines 16–18).

Edelweiss utilizes the DR+ rewrite and client-supplied punctuations to synthesize garbage collection logic for this program. Lines 14–15 join `write` with `commit`, but the `lefts` operator preserves only records from `write` into the `notin` operator. Hence, DR+ recognizes that it can "push up" the `notin` operator into the join and reclaim redundant records from `write` as soon as they are reflected in `write_log`. In order to reclaim records from `commit`, however, we need to rule out the possibility of a `write` record appearing after its corresponding `commit` record has been deleted. If the client seals `write.batch` as part of batch commit—a promise to produce no future writes within that batch—DR+ uses this additional information to generate rules that safely reclaim from `commit`.

```
1  class AtomicReads
2    include Bud

4    state do
5      table :read, [:batch]
6      range :read_commit, [:batch]
7      table :snapshot, [:effective, :wid, :batch, :name, :val, :prev_wid]
8      range :snapshot_exists, [:batch]
9      scratch :read_begin, read.schema
10     scratch :read_view, snapshot.schema
11   end

13   bloom do
14     snapshot_exists <= snapshot {|r| [r.effective]}
15     read_begin <= read.notin(snapshot_exists, :batch => :batch)
16     snapshot <+ (read_begin * live).pairs {|r,l| r + l}
17     read_view <= snapshot.notin(read_commit, :effective => :batch)
18   end
19 end
```

Listing 4.10: Atomic registers supporting snapshot reads.

Snapshot Reads

Ensuring that all writes within a batch become visible atomically is not sufficient to guarantee consistent reads of multiple registers. Consider the following history in which batch $T2$ commits after $T1$:

```
T1 : W(x = 1), W(y = 1)
T2 : W(x = 2), W(y = 2)
```

Without any synchronization, a multi-register read ($R1$) could view a state not produced by a serial ordering of write batches:

```
R1: R(x = 1), R(y = 2)
```

We could rule out this anomaly by forcing multi-register reads to participate in a concurrency control scheme with write batches and commit according to a serializable ordering. In a workload in which reads are common or long-running, however, such a scheme can have undesirable effects, interfering with write batches by causing them to wait or abort. An alternative approach is to use a multiversioning scheme in which read batches do not interact with writes, but nevertheless perceive a *snapshot* of the store consistent with a serial ordering of writes [128, 136, 154]. This requirement implies that it is not necessarily safe to reclaim entries in the log as soon as they are dominated by a more recent write—these entries must persist in some form until we are certain that no active multi-register reads will reference them.

Listing 4.10 shows a simple extension of the implementation shown in Listing 4.9 that supports snapshot reads. Clients initiate a read transaction by inserting a unique *batch* identifier into `read`, and retrieve register values for that batch via the view `read_view`. When the read-only batch is complete, clients insert into `read_commit`.

To simplify the presentation, we provide an intuitive but inefficient snapshotting algorithm. When a read occurs for a batch for which no corresponding snapshot exists (line 15), a copy of `live` (the view containing the current value for each register) is copied into `snapshot` (line 16). Subsequent reads within the batch are then served from this snapshot, allowing concurrent writes to proceed without interference. At any time, the output view `read_view` contains the *active* set of snapshots: those referenced by read-only batches that have not yet committed.

Edelweiss needs to determine when facts from `read` and `snapshot` can safely be reclaimed. In both cases, a straightforward application of DR+ automatically generates deletion logic. Records in `read` can be reclaimed as soon as their `batch` is reflected in the *range* relation `snapshot_exists`. A record in `snapshot` is only necessary to derive a record in `read_view` while its read batch is active; as soon as its `batch` appears in `read_commit`, it too can be reclaimed.

4.7 Evaluation

In this section, we evaluate two aspects of Edelweiss. First, we verify that the storage reclamation logic produced by Edelweiss works effectively. Second, we evaluate the quantitative and qualitative benefits of programming in Edelweiss. We show that Edelweiss enables significant reductions in code size and complexity.

Storage Reclamation

To validate that Edelweiss is effective at safely and automatically reclaiming storage, we study how the causal KVS (Section 4.5) behaves in two scenarios. First, we show that Edelweiss automatically discards dominated writes. Second, we report the behavior of the causal KVS during a network partition, confirming the expected behavior that partitions prevent storage from being reclaimed [103].

Dominated Writes

As discussed in Section 4.5, Edelweiss automatically infers that a write operation in the causal KVS can be discarded when (a) the write has been replicated to all nodes, and (b) the write has been dominated by another write, because dominated writes no longer contribute to the replica's current view. To verify this behavior, we created a single-site KVS and measured the storage requirements of the system over time. We submitted write operations at a constant rate (50 writes per second), but varied the percentage of writes that updated a previously written key. As the fraction of updates increases, the number of undominated writes in the replica's view decreases, and hence we expect the program to require less storage.

Figure 4.1 reports the results of this experiment for several update percentages. As the fraction of updates in the workload increases, more dominated writes are observed and hence Edelweiss automatically reclaims more stored tuples.

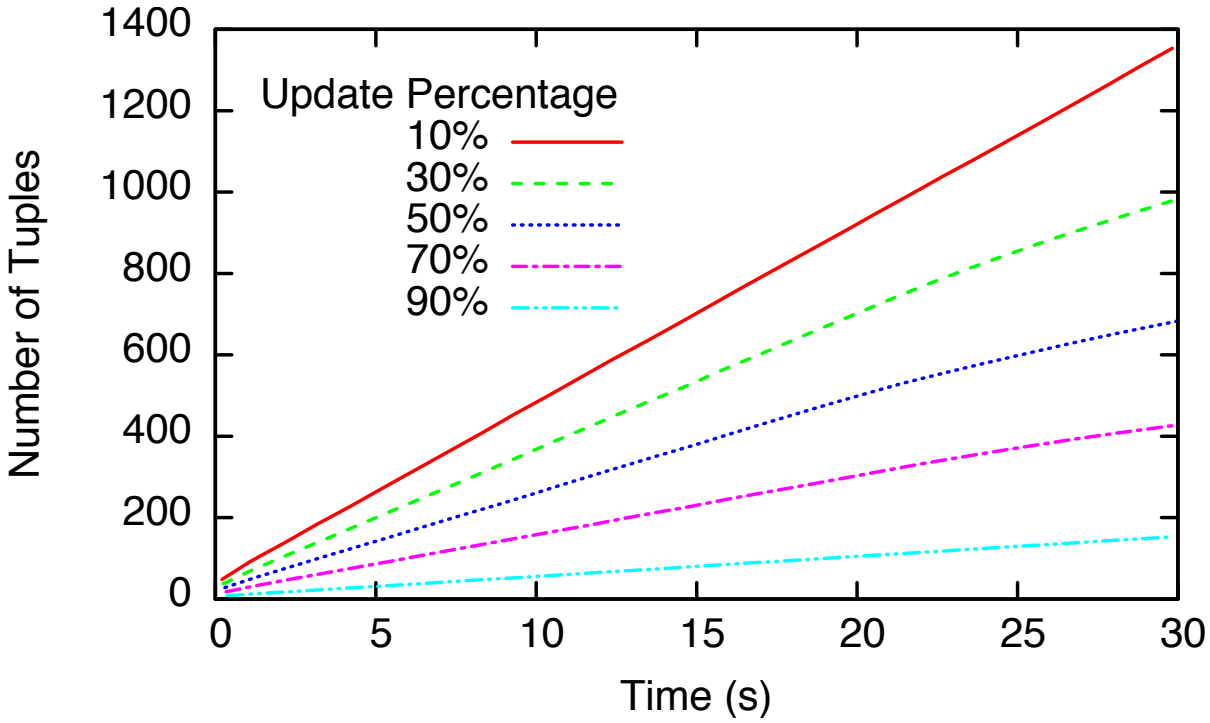


Figure 4.1: Storage consumed by a causal KVS replica for different update percentages.

Network Partitions

Next, we consider how the reclamation protocol generated by Edelweiss behaves during network partitions. When the network is partitioned, write operations cannot be replicated to all replicas; hence, those operations must be retained until the partition heals, temporarily increasing storage requirements.

To study this behavior, we created a simple causal KVS cluster with two replicas, *A* and *B*. A single client continuously submits writes to replica *A*. Every write is a dominating update, so when the network is connected we expect the storage required at each replica to remain constant over time. We then simulated two network partitions by dropping all channel messages sent between *A* and *B*.

Figure 4.2 reports the storage required by replica *A* for this experiment. The network is connected for the first 30 seconds of the experiment; as expected, the replica's storage requirements do not increase. Starting at 30 seconds and continuing for the next 50 seconds, we simulated a network partition. The reclamation logic generated by Edelweiss does not allow write operations to be discarded until they have been successfully replicated, and hence the number of tuples retained by replica *A* grows.

After 80 seconds, the partition is healed. Replica *A* promptly sends its backlog of write operations to *B*, and the ARM-generated acknowledgment logic at *B* informs *A* that the writes have been delivered successfully. Edelweiss can then safely discard those write operations, leading to an

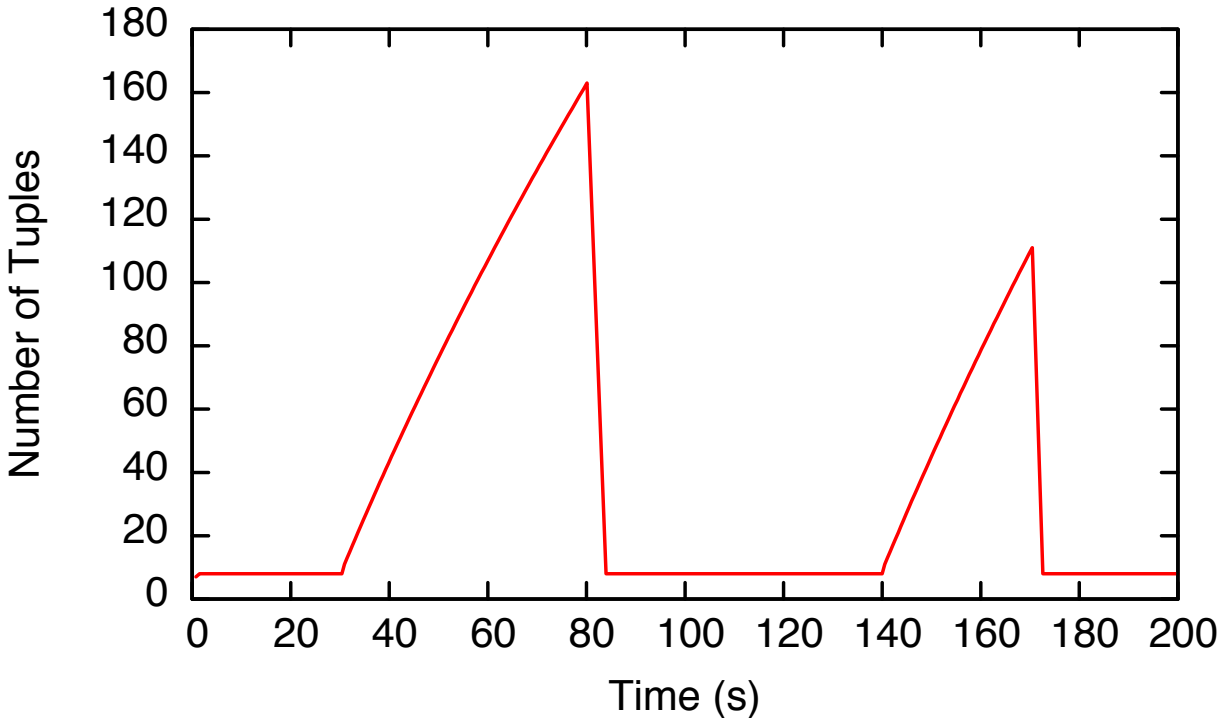


Figure 4.2: Storage consumed by a causal KVS replica. Network partitions are simulated at 30 and 140 seconds.

immediate drop in storage. As the experiment continues, a similar pattern of behavior can be observed: storage remains stable as new writes arrive while the network is connected, then grows during the partition that begins at 140 seconds, and finally shrinks again once the partition is healed.

Program Size and Complexity

Next, we consider the code size and qualitative complexity for several common distributed algorithms implemented using Edelweiss. In Table 4.2, the "Input" column shows the number of rules in each Edelweiss program, while the "Rewritten" column shows the number of rules in the corresponding Bloom program that is produced automatically by the Edelweiss compiler. That is, the rewritten programs include mechanisms for knowledge propagation and storage reclamation that are inferred automatically by Edelweiss. In some cases, the rewritten programs contain a small number of redundant rules that could be avoided by a careful Bloom developer. Similarly, a developer might choose to ignore reclamation conditions that are supported by our rewrites—for example, if punctuations for a certain collection will never be supplied, reclamation rules that depend on those punctuations can be omitted. Nevertheless, when examining the rewritten programs by hand, we found they had a similar structure to hand-crafted acknowledgment and reclamation mechanisms we have written in the past.

	# of Rules	
Description	**Input**	**Rewritten**
Reliable unicast (§4.2)	2	5
Reliable broadcast, fixed (§4.3)	2	8
Reliable broadcast, epoch-based (§4.3)	2	12
Causal broadcast (§A.2)	6	14
Request-response pattern (§A.2)	7	16
Key-value store (§4.4)	5	23
Key-value store with causal consistency (§4.5)	19	62
Atomic registers (§4.6)	4	11
Atomic registers, multi-key writes (§4.6)	4	17
Atomic registers, snapshot reads (§4.6)	8	23

Table 4.2: Code size comparison.

All of the programs in Table 4.2 are concise, particularly in comparison to implementations using traditional imperative languages: for example, a recent causally consistent key-value store prototype required about 13,000 lines of C++ code [103]. Nevertheless, the Edelweiss programs are smaller than their Bloom counterparts by a factor of two or more. Perhaps more importantly, Edelweiss relieves programmers of the need to reason about when storage can safely be deallocated. Instead, the programmer specifies when information remains useful to their application and Edelweiss produces a reclamation scheme that is consistent with those requirements. For example, in the KVS (Section 4.4), the Edelweiss program specifies when (logically) deleted keys should be omitted from the view—the programmer does *not* need to consider when the associated insert and delete operations should be physically reclaimed.

This also means that if the program's semantics do not allow safe reclamation, the result is a storage leak rather than data loss. We observed this first-hand: in the initial version of the key-value store, we arranged for delete operations to specify a key to be removed, rather than an insertion ID. As a result, Edelweiss was unable to reclaim delete operations. While we were initially puzzled, we eventually realized that reclaiming deletions in this program would be unsafe in principle: the program allows multiple insertions with the same key (and different IDs), so reclaiming deletions would change the behavior of the program. As described in Section 4.5, Edelweiss helped us identify a similar logic error in our initial implementation of dominated writes in the causal KVS.

4.8 Implementation

The bulk of the Edelweiss implementation is a source-to-source compiler: given a program written in the Edelweiss sublanguage, it produces a Bloom program with the same behavior that also contains logic for transmitting and storing acknowledgments (ARM), reclaiming facts that are no longer

useful (DR+, DR-), and generating certain kinds of punctuations automatically.[5] Acknowledgments and punctuations are represented as facts in Bloom collections, rather than adopting a special-purpose data format. In principle, this source-to-source translation phase could be implemented entirely outside the Bloom compiler, but because the translator relies on parsing and semantic analysis of the input Edelweiss program, we found it convenient to implement the translator as a preprocessing phase in a modified version of the Bloom compiler.

Implementing Edelweiss as a source-to-source compiler had several advantages. First, because the semantics of an Edelweiss program are exactly those of the generated Bloom code, we avoided the need to define a separate semantics for Edelweiss. This also meant that the behavior of an Edelweiss program can be determined by examining relatively high-level Bloom code, rather than, say, a lower-level intermediate language or query plan format. We exploited this capability while developing Edelweiss: we often found it convenient to write out or modify the generated Bloom code by hand, in some cases before we had even begun implementing Edelweiss itself. For example, we started the reliable unicast case study (Section 4.2) by building a simple acknowledgment protocol by hand and then reasoning about the program analysis techniques that would be required to generate such a protocol automatically. Finally, using a source-to-source compiler encouraged loose coupling between the different Edelweiss transformations: for example, DR+ and DR- can be applied to a broad class of Edelweiss programs that contain negation—including both negation constructs that have been written manually by developers and those generated automatically by the ARM mechanism. Another benefit is that Edelweiss programmers can make use of the same facilities employed by the generated Bloom code, such as `range` collections and punctuations.

Implementing the source-to-source compiler required about 2200 lines of Ruby. The analysis itself is fairly straightforward: the primary challenge was the low-level representation of Bloom programs used by Bud. This implied that our analysis code was forced to manipulate Ruby abstract syntax trees, rather than a higher-level representation, which made the engineering somewhat labor-intensive. We also implemented a simple rule optimizer to eliminate obvious redundancies in the generated rules: for example, if the rules generated by Edelweiss produce an intermediate collection that appears on the lhs of exactly one rule and the rhs of exactly one rule, the collection can be eliminated and its definition "inlined" into the rule where it appeared on the rhs. We implemented the optimizer in part to make the LOC figures reported in Table 4.2 more representative, although the rewrites it performs also likely improve the runtime performance of the generated code.

In addition to the Edelweiss translator, the implementation of the `range` and `sealed` collection types required about 300 lines of Ruby code.

4.9 Discussion

We began the initial work that resulted in Edelweiss because of frustrations we encountered building purely monotone programs in Bloom and Bloom^L. For example, consider the quorum vote program in Listing 3.2. If a quorum of messages is received, the coordinator sends a message on the

[5] Edelweiss automatically generates punctuations for `sealed` collections (Section 4.3) and for situations in which functional dependencies imply that a subset of a collection will henceforth be sealed (Section 4.5).

`result_chn` channel. However, because the program is monotonic, once a condition occurs that causes the program to send a message (e.g., the threshold test on line 19 of Listing 3.2 is satisfied), that condition will always remain true in the future. Hence, the monotonic coordinator program will send an unbounded number of messages on `result_chn`, one for each timestep. Similar behavior can also be observed for the monotone checkout program in Section 3.5; the reliable unicast program presented earlier in this chapter is the simplest form of this pattern.

In each case, unbounded messaging could have been avoided by introducing an acknowledgment protocol; alternatively, we could have modified the Bloom runtime to perform such acknowledgments automatically. However, neither option seemed attractive: adding explicit acknowledgments would have made our programs syntactically non-monotonic (because of the need for negation to suppress duplicate messages), which would have defeated CALM analysis. Modifying the language runtime would have added more "hidden" semantics to the language, making it more difficult to reason about the communication pattern a given program would result in. Edelweiss strikes a middle path between these approaches: it avoids forcing developers to add explicit acknowledgments to their programs, and because it produces a vanilla Bloom program as its output, the semantics of that program can be inspected. Thus, Edelweiss expands the space of monotonic programs that have efficient implementations.

We designed Edelweiss to be a Bloom sublanguage—any valid Edelweiss program is a valid Bloom program with the same user-observable behavior. However, a more aggressive approach would be to build a new language around the design patterns observed in Edelweiss. Most languages for distributed programming view network messages as transient "events": they are delivered to the recipient once and then cease to be. This is the approach taken by Overlog and Bloom, as well as other languages like Erlang. Edelweiss demonstrates that transient events need not be part of the programming abstraction: efficient implementations are still possible even if the language conceptually persists all network messages. Using this insight to build a complete programming language would be an interesting topic for future work. A first step in this direction would be to revise the `channel` collection type to be persistent, rather than a type of scratch collection. This would eliminate some boilerplate code from Edelweiss programs by avoiding the need to copy from a persistent collection into a channel at the sender-side, and from the channel back into a persistent collection at the receiver.

Another connection between Edelweiss and language design relates to the use of `scratch` collections. In Edelweiss, scratch collections play a limited role: all "base" data appears in persistent (`table`) collections, and scratches are only used as a way to name intermediate query results over those tables. In comparison, scratches play a more prominent role in Bloom: the transient behavior of scratches can be used as a way to "smuggle" deletions into programs, e.g., by ensuring that event data is "processed" once and then discarded. Edelweiss shows that using scratch collections in this manner is not necessary; such a restriction could be incorporated into a future language design.

4.10 Related and Future Work

Our work on Edelweiss is related to prior work by several research communities. As noted at the beginning of this chapter, the pattern of operation logging with periodic background reclamation appears frequently in distributed storage and data management. Proposed designs typically differ along a few dimensions, such as how "common knowledge" about the state of other nodes is represented, how this knowledge is communicated (e.g., bundled with log messages [58, 159] or sent via a separate mechanism [24, 109]), and the criteria for determining that an operation is "stable" and can be discarded (e.g., some systems use wall-clock time, waiting a period of time in which replica synchronization is likely to have occurred [52]; in others, the system explicitly tracks the logical clocks of all nodes, which requires all nodes to be available to allow log entries to be reclaimed [159]).

Garbage collection has been extensively studied by the programming language community for both single-site [87] and distributed [2] programs. Traditional garbage collection is applied to a reference graph: subgraphs that are not reachable from one or more "root" vertices can safely be reclaimed. This relies on the property that such objects will never be reachable in the future, which is a special-case of the kind of "henceforth no longer useful" properties exploited by Edelweiss. It would be interesting to see how naturally Edelweiss could be enhanced to synthesize traditional GC schemes.

Multi-version concurrency control (MVCC) is a notable example of the ELE pattern: updates generate new row versions, and readers have a "snapshot" that defines which versions they can legally observe [26, 128]. When a row version has been updated and the old version no longer appears in the snapshot of any active reader, it can safely be reclaimed.[6] This scheme bears a close resemblance to the atomic register design presented in Section 4.6. However, real-world MVCC implementations contain many details we have omitted, such as allowing multiple read/write operations per transaction, dealing with page-level storage, and optimizations to reduce the storage requirements for MVCC metadata and to cache the results of row visibility checks. Building a correct MVCC implementation is extremely difficult in practice, in part because two disparate code paths must be kept in careful alignment—the rules that define which row versions are visible to a transaction, and the garbage collection logic that determines when a row version can be reclaimed. Edelweiss suggests that this redundancy can be eliminated: given a declarative specification of the reader snapshot logic, the garbage collection code might be inferred automatically. Extending Edelweiss to support this scenario is an exciting avenue for future work.

In this chapter, we have focused on systems in which knowledge can eventually be discarded; there is a related design pattern in which nodes accumulate knowledge and then periodically summarize or reorganize it, e.g., in the form of a "checkpoint." Examples include write-ahead logging in database systems [115], log-structured file systems [131], and rollback-based recovery in

[6] In order to ensure this property, the system must guarantee that no future reader will be able to observe the old row version. This can be done by ensuring that snapshots are monotonically increasing with respect to row versions: that is, the snapshot of a new reader is at least as "recent" as that of the previous reader. Most MVCC systems satisfy this property, although it does prevent arbitrary "time travel" to prior versions of the database [143].

distributed systems [144]. We believe that Edelweiss could be extended to support the checkpointing pattern, although this remains a subject for future work.

Our work on Edelweiss also relates to efforts to provide principled foundations for eventually consistent systems. In Chapter 3, we reviewed the CALM Theorem, which shows that monotonic logic programs are deterministic ("confluent") and hence eventually consistent. This shares similarities to Edelweiss and the way in which the ELE model sidesteps consistency concerns. The base collections in an Edelweiss program (e.g., the event logs that are replicated to every node) are sets that grow monotonically over time; the CALM Theorem implies that each replica of such a collection will eventually converge to the same state, once all event log messages have been delivered. However, most of the programs in this chapter apply non-monotonic operators (particularly negation) to those base collections, typically to determine which of the log entries are currently "live." For example, in the KVS presented in Section 4.4, the set of insertion and deletion log entries grows monotonically but the set of "live" key-value pairs does not. Hence, the program as a whole is not confluent. Extending CALM consistency analysis to support the ELE pattern in Edelweiss is an interesting topic for future work.

In this chapter, we focused on programs that accumulate knowledge as a *set* of facts that grows over time. As described in Chapter 3, this notion of "growing over time" can be generalized from sets to join semilattices. Lattices often require a form of periodic garbage collection to restore efficiency [138]; extending Edelweiss to lattices is a natural direction for future work.

Our development of reclamation techniques for Edelweiss has been somewhat ad hoc and driven by the practical programs we have studied. Given a program for which the current Edelweiss prototype does not reclaim storage, it is often unclear whether the problem lies in the program or in the Edelweiss implementation—that is, could a more sophisticated analysis successfully reclaim storage for the program or is the program "un-reclaimable" in principle? Both theoretical and practical developments would be useful here. To the former, a first step would be to formal characterize the class of Edelweiss programs that can be evaluated with bounded storage. To the latter, Edelweiss could be enhanced to provide feedback to developers about how program semantics influence storage requirements. For example, Edelweiss could describe the circumstances under which a given fact can be reclaimed or generate an execution trace in which prematurely reclaiming a fact leads to incorrect program behavior.

Edelweiss's support for automatically avoiding redundant network messages (ARM) bears a resemblance to semi-naive evaluation [22]. As discussed in Section 3.3, semi-naive evaluation works by avoiding redundant derivations in the evaluation of a recursive logic program. Program evaluation proceeds through a series of rounds, where each subsequent round only considers the consequences of "new" conclusions reached in the previous round. The acknowledgment rewrite has a similar flavor: senders only send "new" (unacknowledged) messages, because the consequences of "old" (acknowledged) messages have already been derived. ARM can be viewed as "network-oriented semi-naive evaluation": that is, ARM is necessary because the Bloom runtime employs semi-naive evaluation for local rule evaluation but not for communication between nodes.

4.11 Conclusion

Edelweiss demonstrates for the first time that the benefits of the ELE pattern for distributed programming do not require custom state reclamation code. The fact that this result arose from a basis in Bloom is not coincidental. Rewrites like Difference Reclamation were inspired directly from the use of a declarative, set-oriented language. Moreover, the clear data dependencies in a declarative language made our analysis code easy to write. It is an open question whether our techniques can be transferred to (immutable versions of) more popular imperative languages, which could be quite useful in practice. Meanwhile, it is our experience that distributed programming design patterns like ELE are quite well-served by declarative distributed languages, and we believe that the design and analysis of such languages is a fruitful direction for further exploration.

Chapter 5

DiCE: Declarative Concurrent Editing

In the previous two chapters, we examined two design patterns that are used in a variety of loosely consistent distributed systems. We showed how programming language support can help developers apply these patterns more effectively. In this chapter, we instead turn to a particularly challenging class of loosely consistent programs, *concurrent editing systems*.

In a concurrent editing system, users at different sites manipulate a shared document (e.g., a text file) [54]. To ensure a responsive user interface and allow disconnected operation, each user's edits are immediately applied to their local site and then propagated to the other sites asynchronously. This introduces a familiar consistency concern: different sites might receive editing operations in different orders, but the system must ensure that every site nevertheless observes correct behavior (e.g., all sites should eventually agree on the content of the document).

Despite being among the first examples of loose consistency in the literature, concurrent editing systems are notoriously difficult to implement correctly; in fact, many published designs have been shown to be incorrect [84]. Typical concurrent editors employ a system of string-oriented transformation rules that describe how remote operations should be integrated into each site's local state, but reasoning about the behavior of these rules is very difficult. In this chapter, we explore whether declarative programming languages such as Bloom can simplify the design and implementation of concurrent editing systems.

We begin by reviewing the concurrent editing problem and showing how it can be formulated over an abstract graph representing the relationships between edits (Section 5.1). We describe a set of high-level, declarative rules over the graph structure that define how concurrent edits should be merged together (Section 5.2). Unlike the programs presented in Chapter 3, these rules are not syntactically monotonic—in fact, they cannot even be evaluated by a traditional Datalog interpreter. To resolve this problem, we develop an extension to Bloom called Bloom^{PO}, which allows our concurrent editing design to be evaluated (Section 5.3). We then show how Bloom^{PO} allows a simple, declarative implementation of concurrent editing (Section 5.4).

5.1 Problem Definition

In a concurrent editing system, users submit *insert* and *delete* operations to add and remove content from a shared document. Each user's edits are applied to their local site immediately and propagated to other sites asynchronously. Hence, different sites might receive editing operations in different orders; the challenge is to ensure that despite this possibility, users observe reasonable behavior.

Sun et al. propose that concurrent editing systems should satisfy the "CCI" correctness criteria [146]:

1. *Convergence*: Given the same set of editing operations (perhaps delivered in different orders), every site reaches the same state.

2. *Causality*: Each site applies editing operations in an order that is consistent with the happens-before relation; that is, causal relationships between editing operations are respected.

3. *Intention Preservation*: When an operation is applied to a remote site, the effect of applying the operation preserves the user's intention when the operation was generated.

As explained below, DiCE satisfies all three criteria.

Representing Edit Operations

In DiCE, the document consists of a set of immutable *atoms*; an atom represents a word, sentence, or other document fragment, depending on the application. Each atom has a globally unique ID; such IDs can easily be created using $\langle$*site-id*, *local-id*$\rangle$ pairs, where *local-id* is a site-local unique identifier.[1] Each atom ID is mapped to the document fragment associated with that atom; because the content associated with an atom plays no role in DiCE, in the following we simply treat atoms as unique IDs.

DiCE supports two editing operations:

1. *insert*(b, a, c) adds atom b to the document, placing it somewhere after atom a and somewhere before atom c. Any given atom b will be inserted at most once.

2. *delete(x)* removes atom x from the document.

Note that *insert*(b, a, c) does not imply that a, b, and c are adjacent in the document—while that will typically be the case when an insert operation is initially generated, it will often cease to be true as more insertions are made. Hence, each insert operation specifies a partial order: *insert*(b, a, c) is equivalent to $a \prec b \prec c$. The set of all insertions must be acyclic. To denote the beginning and end of a document, DiCE uses two "sentinel" atoms, denoted `BEGIN` and `END` respectively.

Deletions are supported in a straightforward and monotonic manner: each site accumulates the set of deletions it has seen and simply omits deleted atoms from the view of the document shown to

[1] Note that we use scalar atom IDs in this chapter for the sake of simplicity. As described in Section 4.3 of Chapter 4, a $\langle$*site-id*, *local-id*$\rangle$ pair can easily be mapped to a single scalar ID.

the user (i.e., deletions are "tombstone" records). Because deletions form a set that strictly grows over time, it is easy to see that eventually all sites will agree on the set of deleted atoms [11].[2] Hence, we do not further discuss support for deletion in the remainder of this chapter.

Representing The Solution

A set of insert operations implies a partial order over atoms; the task of a concurrent editing system is to compute a *monotone linearization* of that partial order. That is, we want to find a total order $<_F$ over atoms such that, if $a <_F b$ holds for any set S of editing operations, $a <_F b$ remains true for any superset of S. Intuitively, such a linearization technique could be used to build an effective concurrent editor: each site can order the atoms that they have received and present the resulting document to the user, safe in the knowledge that any editing operations presently unknown will not contradict the site's local ordering. Moreover, because F is a monotone function from a set of edit operations to a set of orderings, prior work has shown that all sites will converge to the same final state once all editing operations have been delivered [11].

Note that finding a monotone linearization of a growing set of partial orders is not possible in general. For example, given $a \prec b$ and $a \prec c$, two linearizations are possible: $a < b < c$ and $a < c < b$. If the set of partial orders can grow in an arbitrary (acyclic) manner, neither linearization can safely be chosen: the system might later discover either $b \prec c$ or $c \prec b$, which might contradict whichever linearization was selected. Fortunately, the constraints in Section 5.1 prevent this scenario. Partial orderings are only produced by insertions and each atom is only inserted once; hence, new partial orders will be of the form $a \prec b_{new} \prec c$, where b_{new} is a previously unknown atom ID. Hence, new partial orders reflect how a new atom is ordered with respect to the existing atoms, rather than altering the ordering between two previously known atoms.

Graphs: Explicit Order and Causality

To visualize the information implied by a sequence of insert operations, two graph structures are useful. Consider the following set of edits:

```
insert(A, BEGIN, END)
insert(B, BEGIN, END)
insert(C, B, END)
insert(D, A, C)
insert(E, C, END)
```

Figure 5.1 shows the corresponding *explicit order graph*: there is a vertex for each atom and an edge from x to y if the set of operations contains either *insert*(x, _, y) or *insert*(y, x, _). That is, the

[2] DiCE does not currently support garbage collection for deletions; because deleted atom IDs are still referenced by other insertions, removing all traces of a deleted atom would require the equivalent of an atomic commit protocol between all sites [124, 138]. Moreover, garbage collection would preclude supporting features like viewing prior versions of the document and "undoing" the effects of previous edits [157].

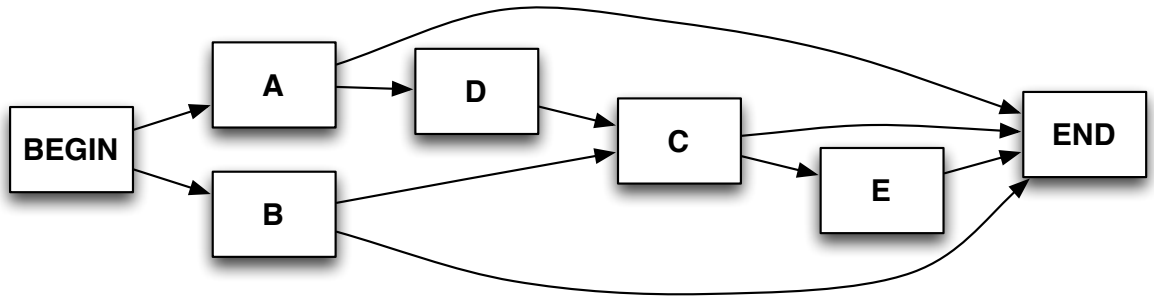


Figure 5.1: An example explicit order graph.

explicit order graph represents the constraints on the ordering of atoms in the document that are implied by the insert operations.

Figure 5.2 depicts the *causal graph* associated with this example: there is an edge from x to y if the set of operations contains either *insert*(y, x, _) or *insert*(y, _, x).[3] The graph depicts the causal relationships between edits: that is, it represents a "happens-before" relation. This differs from Lamport's happens-before relation [95], which conservatively identifies all the *potential* causal predecessors of an event. In DiCE, each insertion operation explicitly identifies the other atoms upon which the newly inserted atom depends. Hence, we can use these dependencies to define a more precise notion of causality, sometimes called *explicit causality* [93].

Using explicit causality has two benefits. First, no additional logical clocks or metadata are needed to record causal dependencies. Second, because these explicit causal relationships are a subset of potential causality, more operations can be applied at remote sites without needing to wait for spurious dependencies [18]. For example, either `D` or `E` might be regarded as a dependency of the other under potential causality (depending on message timing). Using explicit causality, we can observe that both `D` and `E` can be correctly ordered without knowledge of the other atom and hence need not be causally related.

The *ancestors* of an atom are the atom's transitive causal dependencies: x is an ancestor of y iff there is a path from x to y in the causal graph. For example, the ancestors of `E` in Figure 5.2 are {`BEGIN`, `B`, `C`, `END`}. If x is an ancestor of y, we write $x \rightsquigarrow y$ ("x happens-before y").

5.2 Initial Solution: Abstract DiCE

In this section, we describe how to solve the concurrent editing problem introduced in Section 5.1. We begin by showing how DiCE satisfies the causality and intention preservation properties (as defined in Section 5.1). We then show how DiCE finds a monotone linearization of the user's edit sequence by combining three partial orders: the *explicit* order ($\prec_e$), the *tiebreak* order ($\prec_t$), and the *implied-by-ancestor* order ($\prec_i$). Finally, we sketch how these orders can be combined to yield a monotone linearization of the user's editing operations. However, the resulting algorithm is not yet

[3] Note that we chose to have `END` precede `BEGIN` in the causal order; this ordering is arbitrary.

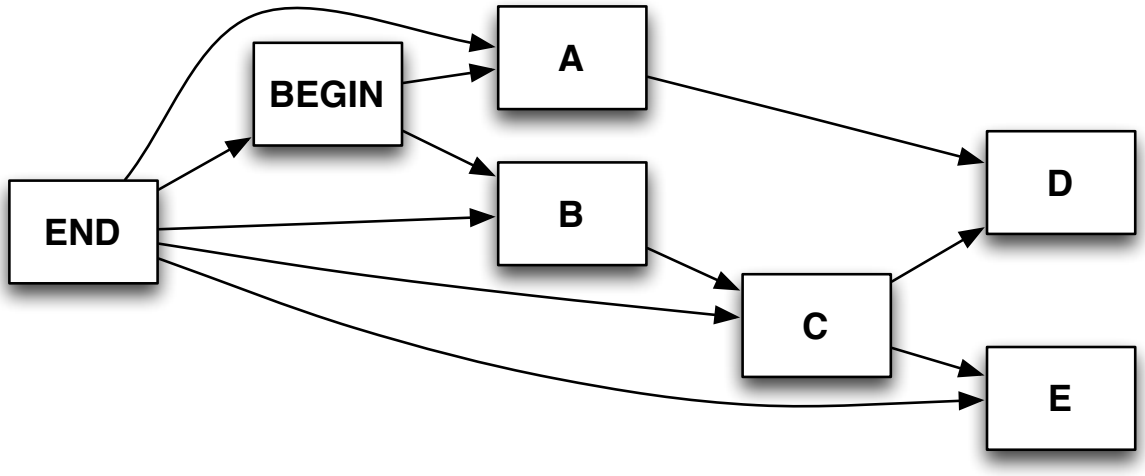


Figure 5.2: An example causal graph.

practical: as we will explain in Section 5.3, the design cannot be evaluated by a traditional Bloom (or Datalog) runtime.

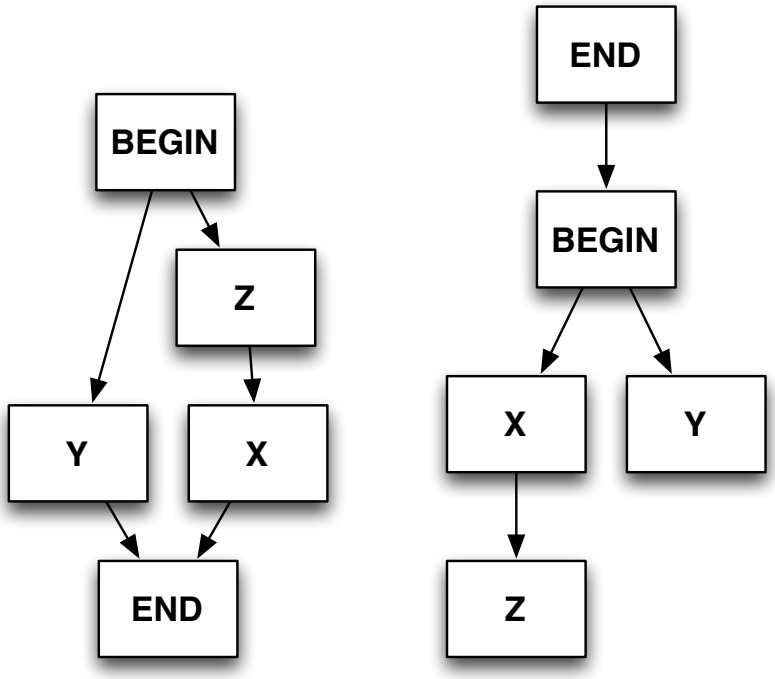
Causality and Intention Preservation

To ensure that the set of delivered atoms at each site respects causal order, DiCE can simply buffer remote edit operations until all of the causal ancestors of that edit operation have been delivered. The transitive structure of the ancestor graph implies that we can implement this constraint by only checking the immediate dependencies of each edit. That is, each site maintains a buffer B of delivered edits and a set S of "safe" edits whose dependencies have been satisfied. S initially contains the sentinel atoms, $\{\texttt{BEGIN}, \texttt{END}\}$. An element $\langle a \prec b \prec c \rangle \in B$ is added to S iff S contains b's immediate dependencies (a and c). Because all the elements in S have their dependencies satisfied, this ensures the entire ancestry of b has been observed at the local site.

DiCE preserves user intentions because it treats user-generated operations as immutable values. Since each site applies exactly the same set of operations and each operation is unchanged from its originating site, the intention of each operation is preserved [120]. This contrasts with traditional concurrent editing systems based on operational transformations: because some operations are modified before they can be applied, the system must ensure that these modifications preserve the user's original editing goals. Sun et al. discuss several subtle situations in which this problem can arise [145]. The immutability of operations in DiCE avoids these concerns.

Explicit Orders

In the remainder of this section, we describe how DiCE ensures convergence by constructing a monotone linearization from several partial orders. The *explicit* order $\prec_e$ is the most fundamental of



(a) Explicit order graph. (b) Causal graph.

Figure 5.3: A scenario in which applying tiebreaks in different orders is problematic.

these: it contains all the ordering constraints implied by the user-generated editing operations, either directly or transitively. That is, $x \prec_e y$ iff there is a path from x to y in the explicit order graph.

Tiebreak Orders

When edits are generated by multiple sites concurrently, the resulting partial order will contain atoms that are mutually incomparable. To decide how to order these edits, DiCE uses a *tiebreak* order, $<_t$, an arbitrary total order over atoms. For example, if atom IDs are represented as pairs of the form $\langle \textit{site-id}, \textit{local-id} \rangle$, the tiebreak order can use numeric inequality: $a <_t b$ iff $a.\textit{site-id} < b.\textit{site-id} \vee (a.\textit{site-id} = b.\textit{site-id} \wedge a.\textit{local-id} < b.\textit{local-id})$.

The tiebreak order is likely inconsistent with the explicit order given by user-generated editing operations. Naturally, the ordering constraints that result from editing operations should be preferred, so DiCE only uses the tiebreak order for pairs of atoms that are incomparable under $\prec_e$. That is, tiebreaks are only used to order concurrent edits. To represent this fact, DiCE actually uses $\prec_t$, a partial order that is a subset of $<_t$: $x \prec_t y$ iff $x <_t y$ and neither $x \prec_e y$ nor $y \prec_e y$ is true.

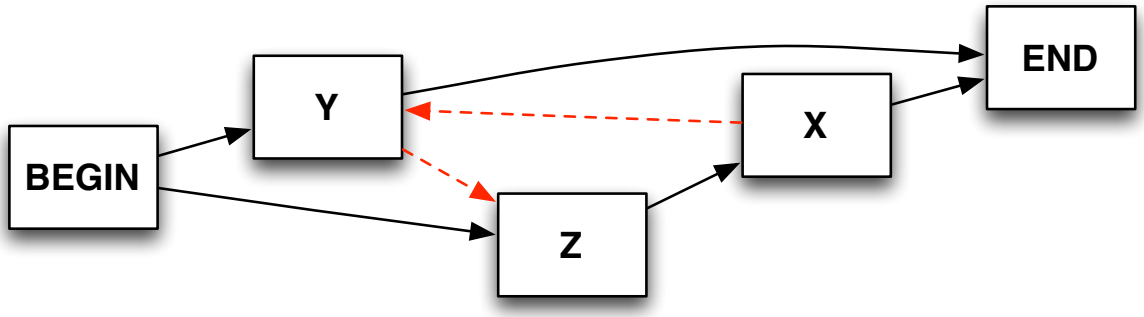


Figure 5.4: Candidate tiebreak orders (dashed edges) for the scenario in Figure 5.3.

Implied-By-Ancestor Orders

However, using the tiebreak order for every pair of concurrent edits is not adequate. Consider the scenario in Figure 5.3, which corresponds to the following operations:

```
insert(X, BEGIN, END)
insert(Y, BEGIN, END)
insert(Z, BEGIN, X)
```

(Note that for the sake of clarity, we omit some transitively redundant edges from both graphs.) Atom Y is concurrent with both X and Z. Suppose the tiebreak order is the standard dictionary order over characters; hence $X <_t Y <_t Z$. If we use the tiebreak between X and Y, we find that Y comes after X, whereas the tiebreak between Y and Z indicates that Z follows Y. However, we also have the user-provided constraint that Z comes before X. Combining all three orders results in a cycle in the order graph (Figure 5.4), and hence an invalid linearization.

To avoid introducing such inconsistencies, DiCE must consider the transitive consequences of each tiebreak ordering that it adopts. For example, in the scenario above, the system should use either $X \prec_t Y$ or $Y \prec_t Z$; once either tiebreak has been applied, the other tiebreak need not (and cannot) be used.

This raises the question: which tiebreak order should be used? To ensure convergent results, all sites should make the same decision about which tiebreaks to employ, despite the fact that they might receive editing operations in different orders. The solution relies on the fact that all sites apply edit operations in causal order. Hence, although different sites might contain different sets of edits, any site attempting to determine the ordering between atoms a and b must contain all the ancestors of both a and b. To ensure convergence, DiCE guarantees that the order between any two atoms is a function of *only* the causal ancestors of a and b.

In the scenario above, X is a causal ancestor of Z; this implies that a site might need to determine the order between X and Y without knowledge of Z. Therefore, the tiebreak $X \prec_T Y$ must be used; once this tiebreak has been adopted, this implies that Z must also precede Y. We call the fact that Z must precede Y an *implied-by-ancestor* order, because the ordering of Z's causal ancestor (X) implies something about Z's order in the document.

The situations in which implied-by-ancestor orders should be used are defined by the following rules:

Definition 1 *Let x, y, z be atoms such that $y \rightsquigarrow x$ (y is an ancestor of x). There is an implied-by-ancestor order between x and z iff either:*

1. $x \prec_e y$ *and* $y \prec_t z$ *implies* $x \prec_i z$.
2. $y \prec_e x$ *and* $z \prec_t y$ *implies* $z \prec_i x$.

Combining Partial Orders

DiCE works by computing the partial orders described above ($\prec_e$, $\prec_t$, and $\prec_i$), and then combining these orders to produce the desired monotone linearization ($<_F$). Computing $\prec_e$ is straightforward (it is the transitive closure of the constraints implied by the user's edit operations), but determining $\prec_t$ and $\prec_i$ is more involved. This is because there is a dependency between these orders: we only want to use the tiebreak order $a \prec_t b$ if there is no ordering between a and b in either $\prec_e$ or $\prec_i$, but $\prec_i$ itself depends on $\prec_t$. This has two implications. First, we cannot compute $\prec_i$ and $\prec_t$ independently; the computation of the two partial orders must be interleaved. Second, care must be taken to ensure that these orders are computed in the correct manner: otherwise a subsequent computation might invalidate the result of a previous computation. That is, in order to safely conclude $a \prec_t b$, we need to ensure that $b \prec_i a$ does not hold, and furthermore that it will never hold in the future.

To see how such dependency problems can be avoided, observe that we can unambiguously determine the order between two edits that have no causal ancestors. That is, suppose A and B are inserted into an empty document at different sites concurrently. Clearly, A and B are incomparable under $\prec_e$; moreover, because neither edit has any causal ancestors, A and B will also be incomparable under $\prec_i$, so $\prec_t$ can safely be used. This idea can be extended to more complex documents, so long as an invariant is maintained: before using the tiebreak order between A and B, all possible implied-by-ancestor orders involving A and B must have first been considered. Maintaining this invariant is possible because the set of causal ancestors of an edit is finite; moreover, due to the causality requirement, all the causal ancestors of an edit E must be known at a replica before that replica attempts to determine where E appears in the document.

We extend this sketch to a complete algorithm in Section 5.4, but first we consider the language and runtime support required to implement such a program in Bloom.

5.3 Bloom^{PO}

Although the DiCE ordering algorithm is not complicated, it cannot easily be evaluated by Bloom (or by any other traditional Datalog implementation). In this section, we describe why the classical Datalog stratification algorithm cannot support DiCE. We review *universal constraint stratification*, a generalization of traditional stratification that is a good fit for our requirements. We then propose Bloom^{PO}, an extension to Bloom that supports constraint stratification, and describe how we extended the Bloom runtime to evaluate Bloom^{PO} programs.

Stratified Negation

The use of negation in Datalog programs must be restricted, to ensure that all legal programs have a reasonable semantic interpretation (traditionally, a unique minimal model). Traditional Datalog implementations rely on *stratified negation*: the relations in the program are divided into a series of *strata*, such that each use of negation in a rule refers to a relation in a lower stratum [69]. Each stratum can then be evaluated in order (i.e., the rules in that stratum are computed to fixpoint). This guarantees that the input to a negation operator is completely determined before the operator is evaluated, which ensures that the output of the negation operator will never "shrink" or need to be retracted. Stratified negation can be viewed as rewriting a non-monotonic input program into a fixed number of semi-positive programs arranged in a sequence—the IDB of one stratum becomes the EDB of the next higher stratum [69].

Stratified negation disallows programs that have "cycles through negation": if relation R depends on relation S, S cannot depend on the negation of R (either directly or transitively). Unfortunately, precisely such a cycle through negation can be observed in the DiCE algorithm described in Section 5.2 because of the mutual dependency between the $\prec_t$ and $\prec_i$ partial orders. Suppose that we represent each partial order with a relation—e.g., `tiebreak(A,B)` is true iff `A` $\prec_t$ `B`, and similarly for `implied_anc` and $\prec_i$. The problem arises because of the following dependencies:

- We want to use the tiebreak order for A and B if there is **not** an implied-by-ancestor or explicit ordering between A and B.
- We want to use the implied-by-ancestor order for A and B if there is a tiebreak order between A and C and an explicit order between B and C (Definition 1).

Choosing to represent `tiebreak` and `implied_anc` as separate relations is not fundamental—for example, the problem would remain if we used a single relation to represent both orders. Regardless of how orderings are represented, stratified negation requires that the program's relations be divided into a *fixed* number of strata, and that all the relations in a stratum are computed to fixpoint before proceeding to the next stratum. This is not powerful enough to express DiCE, because the manner in which the computation of $\prec_t$ and $\prec_i$ should be interleaved depends on the input data, rather than the syntax of the program.

Constraint Stratification

Intuitively, DiCE can be evaluated by starting with the "oldest" edits in the causal graph and moving "forward." Given two edits a and b, if no implied order is found by considering the ancestors of these edits, the tiebreak order can safely be used.[4] This is a kind of stratification condition: once all the ancestors of a and b have been explored, no further implied orders between a and b will be discovered, and hence we can safely take the negation of `implied_anc(a,b)`. Note that it is important to observe that the negation is applied "backward" in the causal graph, and that the causal graph is acyclic; hence, no tuples can actually participate in a cycle through negation, because that

[4] As discussed in Section 5.2, this relies on the fact that the set of ancestors of an edit is fixed.

would imply a cycle in the causal graph. Hence, whereas stratified negation assigns the program's *relations* into a fixed number of strata, we would like a scheme that divides the program's *data* into strata based on the "happens-before" partial order ($\rightsquigarrow$) between edits.

A suitable technique known as *universal constraint stratification* (UCS) was proposed by Ross [133]. UCS expands the space of stratifiable program by exploiting database integrity constraints. *Monotonicity constraints* are particularly useful: a monotonicity constraint (MC) $F : X \prec_{mc} Y$ over a relation $R(X, Y)$ asserts that, for every tuple $r \in R$, $r.X$ is less than $r.Y$ according to the partial order $\prec_{mc}$ [32]. Note that $\prec_{mc}$ is not fixed or encoded in the program's schema, but instead is a constraint on the content of the database. For example, suppose we represent the causal graph associated with a set of operations as a relation, `causal_hist(From, To)`; a tuple `causal_hist(A, B)` means that edit `A` is a causal ancestor of edit `B`. This relation satisfies the MC $A \prec_{mc} B$ under the happens-before partial order ($\rightsquigarrow$); this also implies that the causal graph is acyclic.

UCS allows programs that contain cycles through negation provided that any tuples that traverse the cycle would necessarily violate a database integrity constraint. Since we are only interested in executions that satisfy the constraints, the apparent cyclic dependency will never be satisfied, and hence the program need not be rejected. Intuitively, UCS allows programs that apply negation only to "smaller" facts in the database. If the implementation ensures that all "smaller" facts have been computed before their successors (according to the relevant monotonicity constraint), the input to a negation operator will be fixed, and hence the output of the negation will not shrink.

Implementation

Bloom^{PO} is an extension to Bloom that supports constraint stratified programs. To achieve this, Bloom^{PO} allows users to define collections that satisfy monotonicity constraints. The Bloom^{PO} runtime stores these collections in a custom data structure and treats them differently during query evaluation: as suggested by Section 5.3, UCS requires that the order in which facts are produced during query evaluation respects the monotonicity constraints.

Ross shows that UCS is a syntactic property, and hence "safe" cycles through negation can be discovered by automatic analysis [133]. However, the current version of Bloom^{PO} does not automatically stratify the program or check that cycles through negation imply a contradiction of the monotonicity constraints; instead, the user provides a manual assignment of the program's rules into strata. Although we found this process to be fairly straightforward, automatic constraint stratification could be supported by a future version of Bloom^{PO}.

Collection Types

Bloom^{PO} defines two new collection types, `po_table` and `po_scratch`. These are similar to the `table` and `scratch` collections in Bloom, except that they also carry an associated monotonicity constraint: both `po_table` and `po_scratch` must have exactly two columns, where the second column is smaller than the first column according to some partial order.

Listing 5.1 shows an example Bloom^{PO} program (originally presented by Ross [133]). The task is to determine whether a complex mechanism is functional. The mechanism consists of *parts*; each part consists of zero or more sub-parts. A part cannot be a sub-part of itself, either directly or transitively (i.e., the part graph is acyclic). A part is *working* if it has been directly *tested*, or if it has at least one child component and all of its child components are working. In Bloom^{PO}, the hierarchical nature of the `part` collection is represented using the `po_table` collection type (line 5).

The program in Listing 5.1 contains a cycle through negation between `working`, `part`, and `has_suspect_part`. However, the negative dependency between `working` and `has_suspect_part` (line 16) checks the functionality of a "smaller" part in the hierarchy, and hence is safe as long as the part hierarchy is acyclic [133]. As noted above, the current version of Bloom^{PO} requires programs with cycles through negation to be manually stratified, as dictated by the `stratum` declarations on lines 11, 15, and 19.

Intuitively, the part hierarchy problem can be solved by first considering *leaf parts* (those with no children). A part with no sub-parts can only be `working` if it appears in `tested` (line 12 of Listing 5.1). Hence, if a leaf part has *not* been directly tested, it must be faulty.[5] Next, the parts that contain *only* leaf parts as children can be considered: consider P, a part that has not been directly tested and that has leaf part L as one of its sub-parts. If L does not appear in `working`, we can safely conclude that P is faulty (via the rules on lines 16 and 20); we know that because L is a leaf part, it will never appear in `working` in the future. In this manner, we can move from the leaves of the hierarchy toward the root, only considering a part once the faultiness of *all* of its sub-parts has been determined. The monotonicity constraint on `part` ensures that at least one leaf part exists (in a non-empty hierarchy), and that enumerating the graph in this manner will terminate without revisiting a previously visited node (i.e., the part graph is acyclic). In the next section, we show how the Bloom^{PO} runtime automatically ensures that the graph is traversed in this order.

Stratified Enumeration

To evaluate a constraint stratified program, Bloom^{PO} must ensure that the tuples in `po_table` and `po_scratch` collections are examined in an order that is consistent with their monotonicity constraints. To enable this, we can use monotonicity constraints to divide a collection into a sequence of disjoint sets $S_0, \ldots, S_n$, which we call the *poset strata* of the collection. Poset strata are easiest to visualize using a graph diagram: Fig. 5.5 depicts the poset strata diagram for the causal graph in Fig. 5.2. For the sake of exposition, we assume the collection has two columns, x and y; a tuple $t(x_0, y_0)$ is represented by an edge from x_0 to y_0 in the diagram. First, we assign graph vertices to poset strata: a vertex y appears in poset stratum i iff i is the smallest integer such that $i \geq 0$ and, for all of y's incoming edges $t(x, y)$, i is greater than the poset strata of x. Next, we can assign edges (and therefore tuples in the associated collection) into strata: the strata of edge $t(x, y)$ is the poset

[5] Note that a leaf part appears only in the `child` column of the `part` tuples for its immediate parent parts. Therefore, the rule on line 20 of Listing 5.1 does not apply to leaf parts.

```
1  class PartHierarchy
2    include Bud

4    state do
5      po_table :part, [:id, :child]
6      table :tested, [:id]
7      scratch :working, [:id]
8      scratch :has_suspect_part, [:id]
9    end

11   stratum 0 do
12     working <= tested
13   end

15   stratum 1 do
16     has_suspect_part <= part.notin(working, :child => :id).map {|p| [p.id]}
17   end

19   stratum 2 do
20     working <= part {|p| [p.id]}.notin(has_suspect_part)
21   end
22 end
```

Listing 5.1: Ross's part hierarchy program [133] in Bloom^{PO}.

strata of y.[6]

Rather than enumerating the entire content of each relation in an arbitrary order (as in a Datalog implementation based on stratified negation), in Bloom^{PO} we want to enumerate one poset stratum of each collection at a time, proceeding from the first stratum to the last. We call this *stratified enumeration*. To perform stratified enumeration efficiently, Bloom^{PO} stores `po_table` and `po_scratch` collections in a custom data structure called a *stratified graph*. In a stratified graph, each node contains a list of parent nodes, as well as the length of the longest path from that node to a leaf—this length defines the poset stratum in which the node belongs.

Listing 5.2 contains a stratified graph implementation in Ruby.[7] The `insert(x,y)` method adds a new edge $x \rightarrow y$ to the graph, creating nodes for x and y if necessary (the new edge is assumed to preserve the acyclicity of the graph). An instance of `StratifiedGraph` supports a single stratified enumeration; the state of that enumeration is captured by the `current_stratum` and `frontier` instance variables. The `each` method invokes a function on each edge in the current poset stratum, `advance_stratum` advances the enumerator to the next stratum (returning `true` unless that stratum is empty), and `reset` returns the enumerator to the first (lowest) stratum. Note that `each` and `advance_stratum` take care to account for edges that "jump" to higher strata, such as `END` → `E`; although `E` is reachable from `END` in a single hop, it must not be returned by `each` until *all* of the ancestors of `E` have appeared in earlier strata. This is done via the `@frontier` instance

[6]Poset strata are related to the notion of *consistent cuts* in distributed systems [113]. The union of a poset stratum with all of its predecessor strata is a consistent cut: that is, for all $0 \leq j \leq n, \bigcup_0^j S_j$ is a consistent cut. However, the inverse does not hold: there are consistent cuts that cannot be written as a union of poset strata.

[7]We present a slightly simplified version of the data structure for the sake of clarity; for example, we omit the details of how the data structure integrates with Bud's delta-based dataflow runtime.

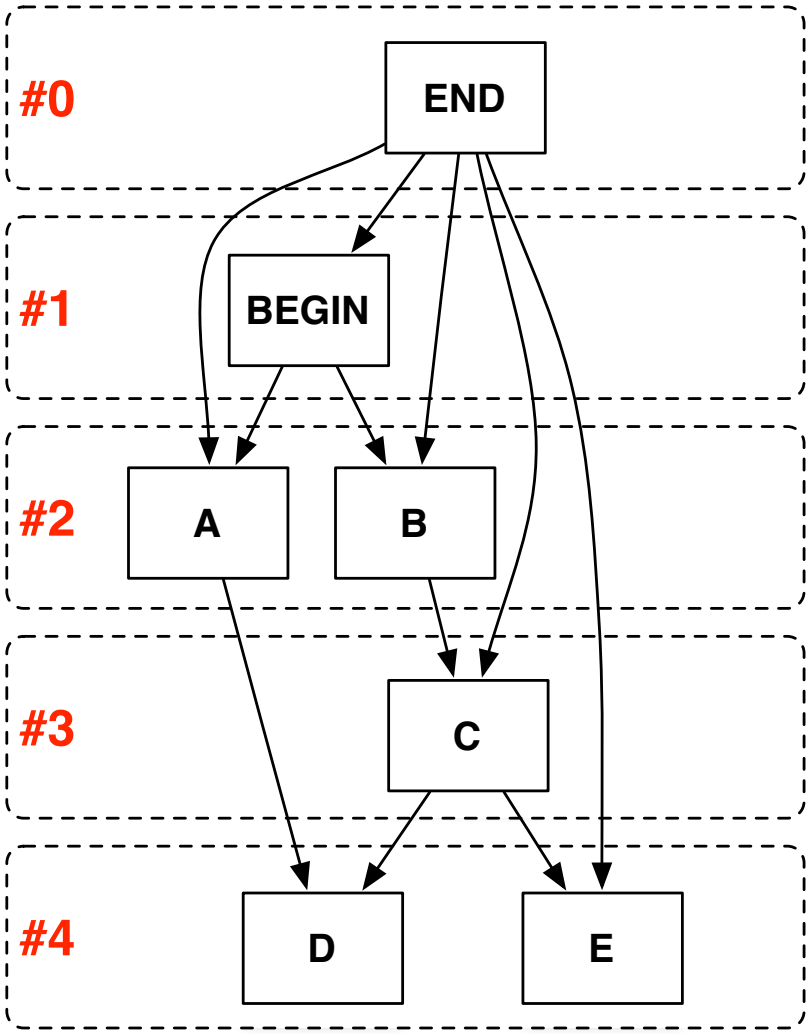


Figure 5.5: Poset strata for the causal graph in Figure 5.2.

variable: if an element of the frontier has a parent that belongs to some future stratum, the frontier element is preserved until that future stratum has been reached (lines 41–43). To avoid returning such preserved frontier elements, `each` skips elements of the current frontier that belong to earlier strata (line 30).

The `insert` method requires $O(n)$ time: in the worst-case, each insertion creates a new leaf that requires updating the `path_len` values of every other node in the graph. Fortunately, the practical performance of the data structure is much better for the programs we present in Section 5.4—stratified graphs are used to represent causal histories, which typically grow by extending a previous root node rather than adding new leaves or interior nodes. For example, adding the edges `C` → `D` and `C` → `E` to the graph in Figure 5.5 does not require updating the `path_len` values of any other nodes, and hence requires only constant time.

Data-Dependent Fixpoint Loop

Stratified enumeration ensures that the content of each `po_table` and `po_scratch` collection is produced in an order that is consistent with the monotonicity constraints. In addition, Bloom^{PO} must ensure that all the consequences of one poset stratum have been computed before proceeding to the next poset stratum. This is very similar to how fixpoint computation works in a traditional Datalog implementation based on stratified negation: for each syntactic stratum in the program, the implementation runs the rules in that stratum to *fixpoint* (i.e., until no more new derivations will be made). Hence, Bloom^{PO} extends this concept to include a second kind of stratification: we want to loop over each poset stratum and syntactic stratum in the program, evaluating to fixpoint at every step.

Listing 5.3 contains an implementation of this idea in Ruby. The `simple_fixpoint` method (invoked on line 26) takes the fixpoint of a set of rules, i.e., it evaluates the rules until no more new results are produced.[8] If the program does not use any poset-valued collections, Bloom^{PO} takes the fixpoint of each syntactic stratum in turn, as in a traditional implementation of stratified negation (lines 24–30). For each `po_table` or `po_scratch` collection, Bloom^{PO} performs a stratified enumeration of that collection; for every poset stratum of every `po_table` or `po_scratch` collection, we take a traditional syntactic fixpoint (line 15). That is, Bloom^{PO} evaluates a set of nested loops, with one loop for each `po_table` or `po_scratch` collection in the program. Note that the order in which these loops are nested is not significant.

Bloom^{PO} requires one further change to the fixpoint procedure. In a Datalog system based on stratified negation, the stratum of any rule where relation R appears on the right-hand side must be greater than or equal to the stratum of every rule where R appears on the left-hand side; i.e., no rule that accesses R appears in a lower stratum than that of any rule that computes R. In a constraint stratified program, this cannot always be guaranteed: for example, in Listing 5.1, `working` is accessed in stratum 1 but computed in stratum 2. This is a natural consequence of the fact that there is a cycle through negation in this program. Unfortunately, it means that after taking the fixpoint of each syntactic stratum in the program, more facts might be derivable: evaluating a rule in stratum i might produce a new tuple that will be accessed by a rule in strata $< i$. Bloom^{PO} identifies all rules that might produce this behavior; we call the associated dataflow edges the *backward edges* of the program. Whenever a new tuple is derived via a backward edge, the syntactic fixpoint is re-run (line 28 of Listing 5.3).

5.4 DiCE in Bloom^{PO}

Having developed the language constructs and runtime support needed to support constraint stratified programs in Bloom^{PO}, we now turn to how to use Bloom^{PO} to implement DiCE. We present the concurrent editing program in Section 5.4, but begin by describing how to solve a simplified version of the problem.

[8] This is typically done using semi-naive evaluation [22] to improve performance, but the details are not relevant to this discussion.

List Append

Appending to a linked list in a concurrent manner is quite similar to concurrent editing. In concurrent editing, an *insert*(b, a, c) operation specifies that atom b appears between atoms a and c (i.e., $a \prec_e b \prec_e c$), whereas in the list append problem, *append*(b, a) adds an atom b after atom a (i.e., $a \prec_e b$). As in concurrent editing, this also implies an explicit causal relationship between operations: $a \rightsquigarrow b$. Note that, unlike in concurrent editing, $\prec_e$ and $\rightsquigarrow$ are equivalent; hence the associated explicit order and causal graphs are the same. As in concurrent editing, we assume that list elements have globally unique IDs and we ignore the actual content associated with each ID; we also assume the existence of a sentinel element `LIST_START` that represents the beginning of a list.

Listing 5.4 contains a solution to the list append problem in Bloom^{PO}. The program computes three partial orders: the transitive closure of the "explicit" constraints derived directly from the append operations (`explicit_tc`), the tiebreak orders used to determine the placement of concurrent appends (`tiebreak`), and the orders over child nodes implied by their causal ancestors (`implied_anc`). The program combines these three partial orders into the `ord` collection (lines 44–46), which represents the output monotone linearization order.

List append operations arrive in the collection `input_buf` but are not processed until their explicit causal dependencies have been satisfied (line 20). Note that the rule on line 20 does *not* ensure that a set of edits are processed in causal order: rather, given a set of edits, it admits all those edits whose dependencies are satisfied by the node's local state. To ensure that subsequent rule evaluation respects the $\rightsquigarrow$ partial order, we copy the transitive closure of the user's edits into the `causal_ord` collection, which is declared as a `po_scratch` (line 8).

To compute the `implied_anc` collection, we use a *second* poset collection, `cursor`. The idea here is that the `causal_ord` collection produces tuples in poset strata according to $\rightsquigarrow$; for each such batch, we want to start with the causally oldest edits and proceed down the graph in causal order, checking each pair of edits for `implied_anc` orders as we go (lines 33–37). Hence, we need two poset collections because we are essentially doing a nested loop over the causal graph; the `to_check` collection holds the Cartesian product of the tuples in the "frontiers" of the two loops. Pairs of tuples for which no implied ordering is found can safely be tiebroken (lines 41–43).

The abstract definition of implied-by-ancestor rules for concurrent editing (Definition 1 in Section 5.2) can easily be adapted to the list append problem:

Definition 2 *Let x, y, z be list elements such that $y \rightsquigarrow x$ (y is an ancestor of x). Then $z \prec_i x$ iff $z \prec_t y$.*

Interestingly, the rule that computes `implied_anc` orders (lines 33–37) is essentially a translation of this definition into Bloom syntax.

Concurrent Editing

Listing 5.5 contains a solution to the concurrent editing problem in Bloom^{PO}. As we have discussed, list append and concurrent editing are closely related; this similarity can observed in their respective Bloom^{PO} programs. In both programs, two poset collections are used: one to ensure that operations

are considered in causal order and the other to ensure that before two edits are tiebroken, any relevant orders implied by their ancestors have been considered.

The primary difference between the two problems is that each concurrent editing atom has two causal ancestors, while each list append operation only has one. As a result, in concurrent editing the causal graph and the explicit order graph are not the same—i.e., given $a \prec_e b$, either $a \rightsquigarrow b$ or $b \rightsquigarrow a$ might be true. Hence, the causal and explicit orders must be computed separately (lines 22–25 and 31–34, respectively), and the rest of the program must employ one order or the other, as needed.

Similarly, the rules for computing implied ancestor orders are more complex, because each ancestor has two explicit order constraints that need to be considered (lines 36–41 and 42–47, respectively). This follows from the fact that Definition 1 has two subclauses, whereas Definition 2 only has one. This correspondence confirms that our Bloom^{PO} program closely resembles the abstract solution to the concurrent editing problem.

5.5 Discussion

When we embarked upon this project, we expected that implementing a concurrent editing system in Bloom^L would be relatively straightforward and similar in spirit to other monotonic programs discussed in this thesis: a concurrent editor computes a set of orderings that grows over time, which seemed to be a natural fit for the monotonic, lattice-oriented programming style encouraged by Bloom^L (Chapter 3). After completing a successful implementation, we were pleased with the brevity of the resulting program and the correspondence between the specification and the Bloom^{PO} program text, but the program was nevertheless quite tricky to develop. In part, this may be because the algorithm is fundamentally order-sensitive: a correct solution must be careful to examine edits in causal order.

Stratification is the traditional tool for handling such order dependencies in logic programs; hence, it is not surprising that we found the need to employ more sophisticated stratification techniques than simple stratified negation. Encoding Bloom^{PO} order dependencies using collection types (i.e., `po_scratch` and `po_table`) allows concise programs whose rules are identical to classical Bloom rules. On reflection, Bloom^{PO} might actually be too concise, in that the semantics of the poset collection types is crucial to interpreting the meaning of a program: unlike in classical Bloom, iterating over a poset-valued collection implies doing a stratified enumeration. Exposing the order that this enumeration follows in the rule syntax (or in a debugging tool) might help make the semantics of Bloom^{PO} programs more obvious to a casual reader.

By forcing us to carefully analyze the ordering dependencies in our design, using Bloom^{PO} to implement DiCE provides a straightforward opportunity for parallel evaluation. Poset strata define synchronization points: within a given poset stratum, parallel evaluation can be used to improve performance without changing program behavior. Automatic parallelization of monotonic Datalog programs has been well-studied [61, 158, 161]. In contrast, parallelizing a traditional concurrent editing system would be very difficult—we are not aware of any existing systems that support parallel execution.

The Bloom^{PO} programs for both the list append and concurrent editing problems represent the output linearization as a set of $\langle b, a\rangle$ tuples, where each tuple represents the fact that a precedes b. Although this representation is elegant and abstract, it is expensive to compute and to store: a document with n atoms requires $O(n^2)$ space, which implies a lower bound of $O(n^2)$ time for any program that represents the output in this format. Moreover, it contains many redundant facts: because the output linearization must satisfy transitivity, it should be possible to only store the transitive reduction of the total order (e.g., given $\langle b, a\rangle$ and $\langle c, b\rangle$, it should be possible to omit computing or storing $\langle c, a\rangle$).

One way to represent the transitive reduction would be to use a tree-like data structure: the hierarchy of the tree implicitly represents the fact that, if x is a child of y, all the children of x are descendants of y. Although we spent some timing exploring this idea, we found it difficult to preserve the declarative, abstract character of our programs but still compute a tree-like data structure. An interesting idea would be to build a compiler that can recognize the equivalence between a Datalog program that computes all $O(n^2)$ facts and a tree-based program that avoids redundancy, and then to rewrite the program from the declarative but inefficient format to a more efficient variant based on tree manipulation. The result might resemble a relational query planner, albeit one specialized to accept a certain class of programs and emit "plans" that target a non-traditional (tree-oriented) query executor.

5.6 Related Work

Concurrent editing was first studied by Ellis and Gibbs [54], but has since been investigated by many researchers. A popular approach to building concurrent editing systems is known as *Operational Transformation* (OT) [145]. OT-based systems work by defining rewrite rules which describe how remote operations should be transformed before they are applied to the local site. The intent of these rules is to modify an operation o to account for the effects of operations that were not known by the remote site that generated o. For example, suppose that a document initially contains the text "`abc`". Site n_0 deletes the first character, while site n_1 deletes the second character; these operations are represented as op_0 = `Delete(0)` and op_1 = `Delete(1)`, respectively. If these operations are applied by a remote site without transformation, several problems occur. First, op_0 and op_1 are not commutative and hence the final state of a site depends on the order in which the operations are applied: $op_0 op_1$ yields "`b`", while $op_1 op_0$ yields "`c`". Second, the final state might not reflect user intentions—e.g., the user at n_1 intended to delete "`b`", but in the $op_0 op_1$ execution trace, their deletion was applied to "`c`" instead. To correct these problems, OT-based systems ensure that each site applies edit operations in the same order (e.g., $op_0 op_1$), and then transforms each operation o to account for other operations that precede o in the chosen order but were not known to the site that generated o. For example, if the chosen execution order is $op_0 op_1$, op_1 might be rewritten to `Delete(0)` to reflect the fact that op_0 has already been applied.

Many OT-based concurrent editing systems have been proposed (e.g., [99, 100, 130, 145, 151]), including commercial products such as Google Wave [67]. However, OT-based systems are notoriously difficult to implement correctly, in part because it is difficult to reason about the

behavior and correctness of string-oriented rewrite rules. Indeed, many published OT algorithms have been shown to be incorrect [84], including the original dOPT algorithm proposed by Ellis and Gibbs [145].

In Chapter 3, we reviewed prior work on using Convergent and Commutative Replicated Data Types (CRDTs) [138] to ensure that replicas of a loosely consistent system eventually converge. Several CRDT designs for concurrent editing have been proposed, including WOOT [120], Logoot [156], Logoot-Undo [157], and Treedoc [124]. WOOT and DiCE share many similarities: both adopt an abstract, graph-theoretic approach to the concurrent editing problem, use explicit causality rather than potential causality, and use causal order to decide how tiebreaks between concurrent operations should be applied. WOOT and DiCE differ in that WOOT encodes the algorithm in an imperative manner, whereas DiCE uses a declarative language; this allows a close correspondence between the high-level algorithm description (Section 5.2) and the Bloom^{PO} program text (Section 5.4).

Searching [76] and sorting [47] of partial orders has been the subject of recent work in the theory community. The work of Heeringa et al. has some similarities to the graph data structure Bloom^{PO} uses to support stratified enumeration of posets (Section 5.3), but our setting differs from Heeringa et al. in several respects. First, we need to support insertion and stratified enumeration but not the deletion or predecessor operations supported by Heeringa et al. Second, we need to support arbitrary dynamic partial orders, whereas Heeringa et al. assume the input is a "tree-like" partial order described by a Hasse diagram.

5.7 Conclusion

Concurrent editors are notoriously difficult to implement correctly. In this chapter, we recounted our efforts to use a distributed logic language to simplify the development of concurrent editing systems. The results show promise: the resulting programs are concise and declarative. This brevity goes beyond avoiding the need for low-level boilerplate code; more importantly, our programs exhibit a close correspondence between abstract specification and executable source code. However, our experience confirmed that implementing a correct concurrent editor requires carefully controlling the *order* in which different atoms are compared, which is an awkward requirement for traditional declarative programming languages. By adapting Ross's work on universal constraint stratification, we showed how such ordering constraints can be incorporated into logic programs in a relatively seamless way. Nevertheless, implementing and debugging DiCE took more effort than we had hoped would be required.

We observe that, in contrast with concurrent editing, the domains where declarative programming has had the most success tend to be those where programs can be written in a largely order-insensitive manner and still achieve correct results. Several open questions remain: is concurrent editing fundamentally order-sensitive? If so, is our approach of introducing order constraints into logic programs the best one—or alternatively, would another language paradigm be a better fit for expressing such order-sensitive programs? A promising direction might be to retain the declarative

nature of Bloom^{PO}, but to find a syntax (and provide developer tools) that make the ordering dependencies of the program more clear.

```ruby
class StratifiedGraph
  Node = Struct.new(:id, :parents, :path_len)

  def initialize
    @nodes = {}
    reset
  end

  def reset
    @frontier = @nodes.values.select {|n| n.path_len == 0}.to_set
    @current_stratum = 0
  end

  def insert(x, y)
    @nodes[x] ||= Node.new(x, [].to_set, 0)
    @nodes[y] ||= Node.new(y, [].to_set, 0)
    @nodes[y].parents << @nodes[x]
    update_path_len(@nodes[x], @nodes[y].path_len + 1)
  end

  def update_path_len(n, new_len)
    return if n.path_len >= new_len
    n.path_len = new_len
    n.parents.each {|p| update_path_len(p, new_len + 1)}
  end

  def each(&blk)
    @frontier.each do |n|
      n.parents.each do |p|
        blk.call(n, p) if p.path_len == @current_stratum + 1
      end
    end
  end

  def advance_stratum
    @current_stratum += 1
    @frontier = @frontier.flat_map do |n|
      n.parents.map do |p|
        if p.path_len == @current_stratum
          p
        elsif p.path_len > @current_stratum
          n
        end
      end.compact
    end.to_set

    return @frontier.all? {|n| n.parents.empty?}
  end
end
```

Listing 5.2: Stratified graph implementation in Ruby.

```
1  def fixpoint
2    if @posets.empty?
3      syntactic_fixpoint
4    else
5      poset_fixpoint(0)
6    end
7  end

9  def poset_fixpoint(idx)
10   curr_poset = @posets[idx]
11   curr_poset.reset

13   while true
14     if curr_poset == @posets.last
15       syntactic_fixpoint
16     else
17       poset_fixpoint(idx + 1)
18     end

20     break unless curr_poset.advance_stratum
21   end
22 end

24 def syntactic_fixpoint
25   while true
26     @syntactic_strata.each {|s| simple_fixpoint(s)}

28     break unless @backward_edges.any? {|e| e.saw_delta?}
29   end
30 end
```

Listing 5.3: The fixpoint loop in Bloom^{PO}.

```
1  class ListAppend
2    include Bud

4    state do
5      table :input_buf, [:id] => [:pred]
6      table :explicit, [:id, :pred]
7      table :explicit_tc, explicit.schema
8      po_scratch :causal_ord, explicit.schema
9      po_scratch :cursor, explicit.schema
10     scratch :to_check, [:x, :y]

12     scratch :check_tie, explicit.schema
13     table :tiebreak, explicit.schema
14     table :implied_anc, explicit.schema
15     table :ord, explicit.schema
16   end

18   stratum 0 do
19     # Buffer inputs until their predecessor has been received
20     explicit <= (input_buf * explicit).lefts(:pred => :id)

22     # Compute the transitive closure of the explicit constraints
23     explicit_tc <= explicit
24     explicit_tc <= (explicit * explicit_tc).pairs(:pred => :id)
25                    {|e,t| [e.id, t.pred] unless t == LIST_START}

27     causal_ord <= explicit_tc
28     cursor     <= explicit_tc

30     to_check <= (cursor * causal_ord).pairs {|c,s| [c.id, s.id] if c.id != s.id}
31     to_check <= (cursor * causal_ord).pairs {|c,s| [s.id, c.id] if c.id != s.id}

33     implied_anc <= (to_check * explicit_tc * tiebreak).combos(to_check.x => explicit_tc.id,
34                                                               to_check.y => tiebreak.pred,
35                                                               explicit_tc.pred => tiebreak.id) do |_,e,t|
36       [e.id, t.pred]
37     end
38   end

40   stratum 1 do
41     check_tie <= to_check {|c| [[c.x, c.y].max, [c.x, c.y].min]}
42     tiebreak <= check_tie.notin(explicit_tc, :id=>:pred, :pred=>:id)
43                          .notin(implied_anc, :id=>:pred, :pred=>:id)
44     ord <= explicit_tc
45     ord <= tiebreak
46     ord <= implied_anc
47   end
48 end
```

Listing 5.4: Concurrent list append in Bloom^{PO}.

```
class ConcurrentEditor
  include Bud

  state do
    table :input_buf, [:id] => [:pre, :post]
    scratch :input_has_pre, input_buf.schema
    table :safe, input_buf.schema
    po_table :causal_ord, [:to, :from]
    po_scratch :cursor, causal_ord.schema
    scratch :to_check, [:x, :y]
    table :explicit, [:id, :pred]
    table :explicit_tc, explicit.schema
    scratch :check_tie, explicit.schema
    table :tiebreak, explicit.schema
    table :implied_anc, explicit.schema
    table :ord, explicit.schema
  end

  stratum 0 do
    input_has_pre <= (input_buf * safe).lefts(:pre => :id)
    safe <= (input_has_pre * safe).lefts(:post => :id)
    causal_ord <= safe {|s| [s.id, s.pre] unless s.pre.nil?}
    causal_ord <= safe {|s| [s.id, s.post] unless s.post.nil?}
    causal_ord <= (safe * causal_ord).pairs(:pre => :to) {|s,r| [s.id, r.from]}
    causal_ord <= (safe * causal_ord).pairs(:post => :to) {|s,r| [s.id, r.from]}
    cursor <= causal_ord

    to_check <= (cursor * causal_ord).pairs {|c,s| [c.to, s.to] if c.to != s.to}
    to_check <= (cursor * causal_ord).pairs {|c,s| [s.to, c.to] if c.to != s.to}

    explicit <= safe {|s| [s.id, s.pre] unless s.pre.nil?}
    explicit <= safe {|s| [s.post, s.id] unless s.post.nil?}
    explicit_tc <= explicit
    explicit_tc <= (explicit * explicit_tc).pairs(:pred => :id) {|e,t| [e.id, t.pred]}

    implied_anc <= (to_check * causal_ord * tiebreak * explicit_tc).combos(
                                to_check.x => causal_ord.to, to_check.y => tiebreak.id,
                                causal_ord.from => tiebreak.pred, causal_ord.to => explicit_tc.pred,
                                causal_ord.from => explicit_tc.id) do |tc,c,t,e|
      [t.id, c.to]
    end
    implied_anc <= (to_check * causal_ord * tiebreak * explicit_tc).combos(
                                to_check.x => causal_ord.to, to_check.y => tiebreak.pred,
                                causal_ord.from => tiebreak.id, causal_ord.to => explicit_tc.id,
                                causal_ord.from => explicit_tc.pred) do |tc,c,t,e|
      [c.to, t.pred]
    end
  end

  stratum 1 do
    check_tie <= to_check {|c| [[c.x, c.y].max, [c.x, c.y].min]}
    tiebreak <= check_tie.notin(implied_anc, :id => :pred, :pred => :id)
                         .notin(explicit_tc, :id => :pred, :pred => :id)
    ord <= explicit_tc
    ord <= implied_anc
    ord <= tiebreak
  end
end
```

Listing 5.5: Concurrent editing (DiCE) in Bloom^{PO}.

Chapter 6

Concluding Remarks

To simplify distributed programming on top of loosely consistent storage infrastructure, we need to help programmers reason about the ways in which asynchrony, concurrency, and partial failure affect the correctness of their designs. This requires answering questions about both individual modules—e.g., whether a single component behaves correctly for all possible combinations of message reorderings and node failures—as well as about the entire system—for example, how does composing individual modules together affect the end-to-end consistency of the entire system? In this thesis, we explored how both questions can be addressed by introducing new language features. In many cases, our work draws inspiration from informal design patterns already employed by practitioners to tackle these challenges.

We found that all three of the language variants explored in this thesis made the development of correct distributed programs significantly easier. The resulting programs were typically very concise but remained readable. Perhaps more importantly, adopting a language-centric approach made it easy to analyze end-to-end behaviors of entire systems: for example, to reason about how lattices in Bloom^L are composed, and to ensure that the garbage collection schemes produced by Edelweiss account for the semantics of all the rules in the program.

Although developing new language variants can produce impressive results, it also increases the effort required to apply these results to mainstream systems built with traditional programming languages. We were aware of this concern when we began the project and tried to ameliorate it by making Bloom an "internal" domain-specific language (DSL) [60], reusing the Ruby syntax and type system. On reflection, this strategy was not entirely successful, perhaps because the semantics of Bloom are sufficiently different from those of Ruby that syntax-level integration does not change the fact that Bloom is fundamentally a language and not a library. In some cases, our work could have been packaged as a library instead: for example, rather than delivering Bloom^L as a language, we could have instead provided a library of lattice types and functions for composing them together. This approach merits further investigation—although it does seem to be less powerful than designing a new language, and not all language features might be conveniently encoded as libraries.

This work was initially motivated by the belief that traditional programming languages do not provide effective support for the challenges raised by modern distributed computing. Our work in this thesis has only confirmed this conviction. While the "right" language for distributed computing

remains an open question, we strongly believe that further innovation in language design is needed to enable mainstream developers to fully exploit the opportunities presented by pervasive distributed computing.

Bibliography

[1] D. J. Abadi. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *IEEE Computer*, 45(2):37–42, 2012.

[2] S. E. Abdullahi and G. A. Ringwood. Garbage collecting the Internet: a survey of distributed garbage collection. *ACM Computing Surveys*, 30(3):330–373, Sept. 1998.

[3] D. Agrawal, A. E. Abbadi, and R. C. Steinke. Epidemic algorithms in replicated databases. In *PODS*, 1997.

[4] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, Aug. 1995.

[5] J. E. Allchin. *An architecture for reliable decentralized systems*. PhD thesis, Georgia Institute of Technology, 1983.

[6] P. Alvaro, T. J. Ameloot, J. M. Hellerstein, W. R. Marczak, and J. den Bussche. A declarative semantics for Dedalus. Technical Report UCB/EECS-2011-120, EECS Department, UC Berkeley, Nov. 2011.

[7] P. Alvaro, P. Bailis, N. Conway, and J. M. Hellerstein. Consistency without borders. In *SoCC*, 2013.

[8] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, and R. Sears. BOOM Analytics: exploring data-centric, declarative programming for the cloud. In *EuroSys*, 2010.

[9] P. Alvaro, T. Condie, N. Conway, J. M. Hellerstein, and R. Sears. I do declare: consensus in a logic language. *ACM SIGOPS Operating Systems Review*, 43(4):25, Jan. 2010.

[10] P. Alvaro, N. Conway, J. M. Hellerstein, and D. Maier. Blazes: Coordination analysis for distributed programs. In *ICDE*, 2014.

[11] P. Alvaro, N. Conway, J. M. Hellerstein, and W. R. Marczak. Consistency analysis in Bloom: a CALM and collected approach. In *CIDR*, 2011.

[12] P. Alvaro, A. Hutchinson, N. Conway, W. R. Marczak, and J. M. Hellerstein. BloomUnit: Declarative testing for distributed programs. In *DBTest*, 2012.

[13] P. Alvaro, W. R. Marczak, N. Conway, J. M. Hellerstein, D. Maier, and R. Sears. Dedalus: Datalog in time and space. In *Datalog Reloaded*. Springer Berlin / Heidelberg, 2011.

[14] T. J. Ameloot, F. Neven, and J. Van den Bussche. Relational transducers for declarative networking. In *PODS*, 2011.

[15] K. R. Apt, H. A. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann, 1988.

[16] S. Babu and H. Herodotou. Massively Parallel Databases and MapReduce Systems. *Foundations and Trends in Databases*, 5(1):1–104, 2013.

[17] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Highly available transactions: Virtues and limitations. In *VLDB*, 2014.

[18] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. The potential dangers of causal consistency and an explicit solution. In *SoCC*, 2012.

[19] P. Bailis and A. Ghodsi. Eventual Consistency Today: Limitations, Extensions, and Beyond. *ACM Queue*, 11(3), 2013.

[20] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Bolt-on causal consistency. In *SIGMOD*, 2013.

[21] J. Baker, C. Bond, J. C. Corbett, J. J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, 2011.

[22] I. Balbin and K. Ramamohanarao. A generalization of the differential approach to recursive query evaluation. *Journal of Logic Programming*, 4(3):259–262, 1987.

[23] Basho Technologies. Riak. `http://basho.com/riak/`.

[24] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication. In *NSDI*, 2006.

[25] J. L. Bentley. Decomposable searching problems. *Information Processing Letters*, 8(5):133–136, June 1979.

[26] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–221, 1981.

[27] N. Bidoit. Negation in rule-based database languages: a survey. *Theoretical Computer Science*, 78:3–83, 1991.

[28] K. Birman, G. Chockler, and R. van Renesse. Toward a cloud computing research agenda. *SIGACT News*, 40(2):68–80, June 2009.

[29] K. P. Birman and T. A. Joseph. Exploiting virtual synchrony in distributed systems. In *SOSP*, 1987.

[30] E. Brewer. Towards robust distributed systems (invited talk). In *PODC*, 2000.

[31] E. Brewer. CAP Twelve Years Later: How the "Rules" Have Changed. *IEEE Computer*, 45:23–29, 2012.

[32] A. Brodsky and Y. Sagiv. Inference of monotonicity constraints in Datalog programs. In *PODS*, 1989.

[33] S. Burckhardt, M. Fähndrich, D. Leijen, and M. Sagiv. Eventually consistent transactions. In *ESOP*, 2012.

[34] S. Burckhardt, M. Fähndrich, D. Leijen, and B. P. Wood. Cloud types for eventual consistency. In *ECOOP*, 2012.

[35] S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski. Replicated data types: Specification, verification, optimality. In *POPL*, 2014.

[36] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI*, 2006.

[37] S. Ceri, M. A. W. Houtsma, A. M. Keller, and P. Samarati. Independent updates and incremental agreement in replicated databases. *Distributed and Parallel Databases*, 3(3):225–246, July 1995.

[38] D. Chu, L. Popa, A. Tavakoli, J. M. Hellerstein, P. Levis, S. Shenker, and I. Stoica. The design and implementation of a declarative sensor network system. In *SenSys*, 2007.

[39] K. Clay. Amazon.com goes down, loses $66,240 per minute. `http://www.forbes.com/sites/kellyclay/2013/08/19/amazon-com-goes-down-loses-66240-per-minute/`, Aug. 2013.

[40] T. Condie, D. Chu, J. M. Hellerstein, and P. Maniatis. Evita Raced: Metacompilation for declarative networks. In *VLDB*, 2008.

[41] N. Conway, P. Alvaro, E. Andrews, and J. M. Hellerstein. Edelweiss: Automatic storage reclamation for distributed programming. In *VLDB*, 2014.

[42] N. Conway, W. R. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier. Logic and lattices for distributed programming. In *SoCC*, 2012.

[43] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. In *VLDB*, 2008.

[44] O. Cooper. TelegraphCQ: From streaming database to information crawler. Master's thesis, University of California, Berkeley, 2004.

[45] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *OSDI*, 2012.

[46] J. Creps. The log: What every software engineer should know about real-time data's unifying abstraction. `http://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying`, Dec. 2013.

[47] C. Daskalakis, R. M. Karp, E. Mossel, S. Riesenfeld, and E. Verbin. Sorting and selection in posets. In *SODA*, 2009.

[48] S. B. Davidson. *An Optimistic Protocol for Partitioned Distributed Database Systems*. PhD thesis, Princeton University, 1982.

[49] S. B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in partitioned networks. *ACM Computing Surveys*, 17(3):341–370, Sept. 1985.

[50] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.

[51] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, Dec. 2004.

[52] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *PODC*, 1987.

[53] A. R. Downing, I. B. Greenberg, and J. M. Peha. OSCAR: an architecture for weak-consistency replication. In *Databases, Parallel Architectures, and Their Applications*, 1990.

[54] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *SIGMOD*, 1989.

[55] A. A. Farrag and M. T. Özsu. Using semantic knowledge of transactions to increase concurrency. *ACM Transactions on Database Systems*, 14(4):503–525, Dec. 1989.

[56] C. J. Fidge. Timestamps in message passing systems that preserve the partial ordering. In *Australian Computer Science Conference*, 1988.

[57] S. Finkelstein, T. Heinzel, R. Brendle, I. Nassi, and H. Roggenkemper. Transactional intent. In *CIDR*, 2011.

[58] M. J. Fischer and A. Michael. Sacrificing serializability to attain high availability of data in an unreliable network. In *PODS*, 1982.

[59] M. Fowler. Event sourcing. `http://martinfowler.com/eaaDev/EventSourcing.html`, Dec. 2005.

[60] M. Fowler. DomainSpecificLanguage. `http://www.martinfowler.com/bliki/DomainSpecificLanguage.html`, May 2008.

[61] S. Ganguly, A. Silberschatz, and S. Tsur. A framework for the parallel processing of Datalog queries. In *SIGMOD*, 1990.

[62] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems*, 8(2):186–213, June 1983.

[63] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *ICLP*, 1988.

[64] D. K. Gifford. Weighted voting for replicated data. In *SOSP*, 1979.

[65] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[66] R. A. Golding. *Weak-consistency group communication and membership*. PhD thesis, University of California, Santa Cruz, 1992.

[67] Google Wave. `http://www.waveprotocol.org/`.

[68] J. Gray. Notes on Data Base Operating Systems. In *Operating Systems, An Advanced Course*, pages 393–481. Springer-Verlag, 1978.

[69] T. J. Green, S. S. Huang, B. T. Loo, and W. Zhou. Datalog and Recursive Query Processing. *Foundations and Trends in Databases*, 5(2):105–195, 2013.

[70] A. Gupta and I. S. Mumick, editors. *Materialized Views: Techniques, Implementations, and Applications*. MIT Press, 1999.

[71] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD*, 1993.

[72] R. G. Guy, G. J. Popek, and J. Page T.W. Consistency algorithms for optimistic replication. In *ICNP*, 1993.

[73] C. Hale. You Can't Sacrifice Partition Tolerance. `http://codahale.com/you-cant-sacrifice-partition-tolerance/`, Oct. 2010.

[74] J. R. Hamilton. I love eventual consistency but... `http://perspectives.mvdirona.com/2010/02/24/ILoveEventualConsistencyBut.aspx`, Feb. 2010.

[75] A. Heddaya, M. Hsu, and W. E. Weihl. Two phase gossip: Managing distributed event histories. *Information Sciences*, 49(1):35–57, 1989.

[76] B. Heeringa, M. C. Iordan, and L. Theran. Searching in dynamic tree-like partial orders. In *WADS*, 2011.

[77] P. Helland. Immutability changes everything! `http://vimeo.com/52831373`. Talk at *RICON*, 2012.

[78] P. Helland. Life beyond distributed transactions: an apostate's opinion. In *CIDR*, 2007.

[79] P. Helland and D. Campbell. Building on quicksand. In *CIDR*, 2009.

[80] P. Helland and D. Haderle. Engagements: Building Eventually ACiD Business Transactions. In *CIDR*, 2013.

[81] J. M. Hellerstein. The Declarative Imperative: Experiences and conjectures in distributed logic. *SIGMOD Record*, 39(1):5–19, 2010.

[82] M. Herlihy. A quorum-consensus replication method for abstract data types. *ACM Transactions on Computer Systems*, 4(1):32–53, Feb. 1986.

[83] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[84] A. Imine, M. Rusinowitch, G. Oster, and P. Molli. Formal design and verification of operational transformation algorithms for copies convergence. *Theoretical Computer Science*, 351(2):167–183, Feb. 2006.

[85] B. Ippolito. statebox, an eventually consistent data model for Erlang (and Riak). `http://bit.ly/1oys8ZA`, May 2011. Mochi Media Labs.

[86] P. R. Johnson and R. H. Thomas. Maintenance of duplicate databases. RFC 677, IETF, Jan. 1975.

[87] R. Jones, A. Hosking, and E. Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 2011.

[88] L. Kawell Jr., S. Beckhardt, T. Halvorsen, R. Ozzie, and I. Greif. Replicated document management in a group communication system. In *CSCW*, 1988.

[89] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, 1992.

[90] G. Köstler, W. Kießling, H. Thöne, and U. Güntzer. Fixpoint iteration with subsumption in deductive databases. *Journal of Intelligent Information Systems*, 4(2):123–148, 1995.

[91] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. MDCC: Multi-data center consistency. In *EuroSys*, 2013.

[92] L. Kuper and R. R. Newton. LVars: Lattice-based data structures for deterministic parallelism. In *FHCP*, 2013.

[93] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems*, 10(4):360–391, Nov. 1992.

[94] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[95] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[96] L. Lamport. On interprocess communication – parts I and II. *Distributed Computing*, 1(2):77–101, 1986.

[97] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[98] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *OSDI*, 2012.

[99] D. Li and R. Li. Preserving operation effects relation in group editors. In *CSCW*, 2004.

[100] R. Li and D. Li. A new operational transformation framework for real-time group editors. *IEEE Transactions on Parallel and Distributed Systems*, 18(3):307–319, 2007.

[101] LinkedIn, Inc. Voldemort vector clock class. `https://raw.github.com/voldemort/voldemort/master/src/java/voldemort/versioning/VectorClock.java`. Accessed February 20, 2012.

[102] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp file system. In *SOSP*, 1991.

[103] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *SOSP*, 2011.

[104] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *NSDI*, 2013.

[105] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking. *Communications of the ACM*, 52(11):87–95, Nov. 2009.

[106] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. In *SOSP*, 2005.

[107] B. T. Loo, J. M. Hellerstein, R. Huebsch, S. Shenker, and I. Stoica. Enhancing P2P file-sharing with an Internet-scale query processor. In *VLDB*, 2004.

[108] B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative routing: extensible routing with declarative queries. In *SIGCOMM*, 2005.

[109] A. Malpani and D. Agrawal. Efficient dissemination of information in computer networks. *The Computer Journal*, 34(6):534–541, Dec. 1991.

[110] W. R. Marczak, P. Alvaro, N. Conway, J. M. Hellerstein, and D. Maier. Confluence Analysis for Distributed Programs: A Model-Theoretic Approach. In *Datalog 2.0*, 2012.

[111] N. Marz. How to beat the CAP theorem. `http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html`, Oct. 2011.

[112] N. Marz and J. Warren. *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*. Manning, 2014.

[113] F. Mattern. Virtual time and global states of distributed systems. In *Workshop on Parallel and Distributed Algorithms*, 1989.

[114] Microsoft Developer Network. Compensating transaction pattern. `http://msdn.microsoft.com/en-us/library/dn589804.aspx`.

[115] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162, Mar. 1992.

[116] S. Mullender, editor. *Distributed Systems*. Addison-Wesley, second edition, 1993.

[117] B. M. Oki and B. H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *PODC*, 1988.

[118] P. E. O'Neil. The Escrow transactional method. *ACM Transactions on Database Systems*, 11(4):405–430, Dec. 1986.

[119] Oracle Corporation. Streams Conflict Resolution. `http://docs.oracle.com/cd/B10501_01/server.920/a96571/conflict.htm`.

[120] G. Oster, P. Urso, P. Molli, and A. Imine. Data consistency for P2P collaborative editing. In *CSCW*, 2006.

[121] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, Oct. 1979.

[122] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, SE-9(3):240–247, 1983.

[123] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *SOSP*, 1997.

[124] N. Preguiça, J. M. Marques, M. Shapiro, and M. Letia. A commutative replicated data type for cooperative editing. In *ICDCS*, 2009.

[125] D. Pritchett. BASE: An Acid Alternative. *ACM Queue*, 6(3):48–55, May 2008.

[126] Project Voldemort. `http://www.project-voldemort.com`.

[127] D. Ratner. *Roam: A Scalable Replication System for Mobile and Distributed Computing*. PhD thesis, University of California, Los Angeles, 1998.

[128] D. P. Reed. *Naming and Synchronization In A Decentralized Computer System*. PhD thesis, Massachusetts Institute of Technology, 1978.

[129] P. Reiher, J. Heidemann, D. Ratner, G. Skinner, and G. Popek. Resolving file conflicts in the Ficus file system. In *USENIX Summer Conference*, 1994.

[130] M. Ressel, D. Nitsche-Ruhland, and R. Gunzenhäuser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *CSCW*, 1996.

[131] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, Feb. 1992.

[132] K. A. Ross. Modular stratification and magic sets for DATALOG programs with negation. In *PODS*, 1990.

[133] K. A. Ross. A syntactic stratification condition using constraints. In *ILPS*, 1994.

[134] K. A. Ross and Y. Sagiv. Monotonic aggregation in deductive databases. In *PODS*, 1992.

[135] S. K. Sarin and N. A. Lynch. Discarding obsolete information in a replicated database system. *IEEE Transactions on Software Engineering*, SE-13(1):39–47, 1987.

[136] O. T. Satyanarayanan and D. Agrawal. Efficient execution of read-only transactions in replicated multiversion databases. *IEEE Transactions on Knowledge and Data Engineering*, 5(5):859–871, 1993.

[137] R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 7(3):149–174, 1994.

[138] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of convergent and commutative replicated data types. Technical Report RR-7506, INRIA, 2011.

[139] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *13th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, 2011.

[140] W. E. J. Snaman and D. W. Thiel. The VAX/VMS distributed lock manager. *Digital Technical Journal*, (5):29–44, 1987.

[141] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *SOSP*, 2011.

[142] SQLite Query Language: ON CONFLICT clause. `http://sqlite.org/lang_conflict.html`.

[143] M. Stonebraker and G. Kemnitz. The POSTGRES Next Generation Database Management System. *Communications of the ACM*, 34(10):78–92, Oct. 1991.

[144] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, Aug. 1985.

[145] C. Sun and C. Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *CSCW*, 1998.

[146] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Transactions on Computer-Human Interaction*, 5(1):63–108, Mar. 1998.

[147] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *SOSP*, 1995.

[148] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):555–568, May 2003.

[149] D. Tweney. 5-minute outage costs Google $545,000 in revenue. `http://venturebeat.com/2013/08/16/3-minute-outage-costs-google-545000-in-revenue/`, Aug. 2013.

[150] A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):619–649, 1991.

[151] N. Vidot, M. Cart, J. Ferrié, and M. Suleiman. Copies convergence in a distributed real-time collaborative environment. In *CSCW*, 2000.

[152] W. Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[153] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A Note on Distributed Computing. Technical Report SMLI TR-94-29, Sun Microsystems Laboratories Inc., Nov. 1994.

[154] W. E. Weihl. Distributed version management for read-only actions. *IEEE Transactions on Software Engineering*, SE-13(1):55–64, 1987.

[155] W. E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37(12):1488–1505, 1988.

[156] S. Weiss, P. Urso, and P. Molli. Logoot: A scalable optimistic replication algorithm for collaborative editing on P2P networks. In *ICDCS*, 2009.

[157] S. Weiss, P. Urso, and P. Molli. Logoot-Undo: Distributed collaborative editing system on P2P networks. *IEEE Transactions on Parallel and Distributed Systems*, 21(8):1162–1174, 2010.

[158] O. Wolfson. Parallel evaluation of Datalog programs by load sharing. *Journal of Logic Programming*, 12(4):369–393, Apr. 1992.

[159] G. T. J. Wuu and A. J. Bernstein. Efficient solutions to the replicated log and dictionary problems. In *PODC*, 1984.

[160] C. Zaniolo and H. Wang. Logic-based user-defined aggregates for the next generation of database systems. In *The Logic Programming Paradigm: Current Trends and Future Directions*, pages 401–424. Springer Verlag, 1999.

[161] W. Zhang, W. Ke, and S.-C. Chau. Data partition and parallel evaluation of Datalog programs. *IEEE Transactions on Knowledge and Data Engineering*, 7(1), 1995.

Appendix A

Additional Program Listings

This appendix contains the complete source code for several of the programs described in this thesis.

A.1 Bloom^L

Lattice-Based KVS

Dominating Set Lattice Type

```
class DomLattice < Bud::Lattice
  wrapper_name :ldom

  def initialize(i=nil)
    unless i.nil?
      reject_input(i) unless i.kind_of? Hash
      reject_input(i) unless i.keys.all? {|k| k.kind_of? Bud::Lattice}
      reject_input(i) unless i.values.all? {|v| v.kind_of? Bud::Lattice}
    end
    @v = i
  end

  def merge(i)
    i_val = i.reveal
    return i if @v.nil?
    return self if i_val.nil?

    rv = {}
    preserve_dominants(@v, i_val, rv)
    preserve_dominants(i_val, @v, rv)
    wrap_unsafe(rv)
  end

  morph :version do
    compute_reconcile
    @reconcile.first unless @reconcile.nil?
  end

  def value
    compute_reconcile
```

```
    @reconcile.last unless @reconcile.nil?
  end

  private
  def preserve_dominants(target, other, rv)
    target.each_pair do |k1, val|
      # A key/value pair is included in the result UNLESS there is another key
      # in the other input that dominates it. Note that there can be at most one
      # such dominating key in either of the inputs.
      next if other.keys.any? {|k2| k2.merge(k1) == k2 && k1 != k2}
      rv[k1] = val
    end
  end

  private
  def compute_reconcile
    return if @v.nil? or @reconcile
    @reconcile = [@v.keys.reduce(:merge), @v.values.reduce(:merge)]
  end
end
```

Listing A.1: The `ldom` lattice type used in the Bloom^L KVS case study (Section 3.4).

Lattice-based KVS

```
module KvsProtocol
  state do
    channel :kvput, [:reqid, :@addr] => [:key, :val, :client_addr]
    channel :kvput_response, [:reqid] => [:@addr, :replica_addr]
    # If the key does not exist in the KVS, we return a "bottom" value. Since
    # there is not a single bottom value shared across all lattice types (and
    # since we don't have type parameters), we require that the user specify the
    # class to use to construct the bottom value (if needed).
    channel :kvget, [:reqid, :@addr] => [:key, :val_class, :client_addr]
    channel :kvget_response, [:reqid] => [:@addr, :val, :replica_addr]

    # Initiate an async operation to replicate the contents of this replica to
    # the replica at the given address.
    channel :kvrepl, [:@addr, :target_addr]
  end
end

# Simple KVS replica in which we just merge together all the proposed values for
# a given key. This is reasonable when the intent is to store a monotonically
# increasing lmap of keys.
class KvsReplica
  include Bud
  include KvsProtocol

  state do
    lmap :kv_store
  end

  bloom do
    kv_store <= kvput {|c| {c.key => c.val}}
    kvput_response <~ kvput {|c| [c.reqid, c.client_addr, ip_port]}
    kvget_response <~ kvget {|c| [c.reqid, c.client_addr,
                                  kv_store.at(c.key, c.val_class), ip_port]}
  end
end
```

```
class ReplicatedKvsReplica < KvsReplica
  state do
    channel :repl_propagate, [:@addr] => [:kv_store]
  end

  bloom do
    repl_propagate <~ kvrepl {|r| [r.target_addr, kv_store]}
    kv_store <= repl_propagate {|r| r.kv_store}
  end
end

# Simple KVS client that issues put and get operations against a single KVS
# replica. KVS replica address is currently fixed on instantiation. This class
# is not thread-safe.
class KvsClient
  include Bud
  include KvsProtocol

  def initialize(addr, val_class)
    @reqid = 0
    @addr = addr
    @val_class = val_class
    super()
  end

  def read(key)
    reqid = make_reqid
    r = sync_callback(:kvget, [[reqid, @addr, key, @val_class, ip_port]], :kvget_response)
    r.each {|t| return t.val if t.reqid == reqid}
    raise
  end

  def write(key, val)
    reqid = make_reqid
    r = sync_callback(:kvput, [[reqid, @addr, key, val, ip_port]], :kvput_response)
    r.each {|t| return if t.reqid == reqid}
    raise
  end

  # NB: To make it easier to provide a synchronous API, we assume that "to" is
  # local (i.e., we're passed the _instance_ of Bud we want to replicate to, not
  # just its address).
  def cause_repl(to)
    q = Queue.new
    cb = to.register_callback(:repl_propagate) do |t|
      q.push true
    end
    sync_do { kvrepl <~ [[@addr, to.ip_port]] }

    q.pop
    to.unregister_callback(cb)
  end

  private
  def make_reqid
    @reqid += 1
    "#{ip_port}_#{@reqid}"
  end
end
```

```
# More sophisticated KVS client that supports quorum-style operations over a set
# of KVS replicas. Currently, put/get replica sets are fixed on instantiation,
# and we wait for acks from all replicas in the set.
class QuorumKvsClient
  include Bud
  include KvsProtocol

  state do
    table :put_reqs, [:reqid] => [:acks]
    table :get_reqs, [:reqid] => [:acks, :val]
    scratch :w_quorum, [:reqid]
    scratch :r_quorum, [:reqid] => [:val]
  end

  bloom do
    put_reqs <= kvput_response {|r| [r.reqid, Bud::SetLattice.new([r.replica_addr])]}
    w_quorum <= put_reqs do |r|
      r.acks.size.gt_eq(@w_quorum_size).when_true { [r.reqid] }
    end

    get_reqs <= kvget_response {|r| [r.reqid, Bud::SetLattice.new([r.replica_addr]), r.val]}
    r_quorum <= get_reqs do |r|
      r.acks.size.gt_eq(@r_quorum_size).when_true { [r.reqid, r.val] }
    end
  end

  def initialize(put_list, get_list, val_class)
    @reqid = 0
    @put_addrs = put_list
    @get_addrs = get_list
    @r_quorum_size = get_list.size
    @w_quorum_size = put_list.size
    @val_class = val_class
    super()
  end

  def read(key)
    reqid = make_reqid
    get_reqs = @get_addrs.map {|a| [reqid, a, key, @val_class, ip_port]}
    r = sync_callback(:kvget, get_reqs, :r_quorum)
    r.each {|t| return t.val if t.reqid == reqid}
    raise
  end

  def write(key, val)
    reqid = make_reqid
    put_reqs = @put_addrs.map {|a| [reqid, a, key, val, ip_port]}
    r = sync_callback(:kvput, put_reqs, :w_quorum)
    r.each {|t| return if t.reqid == reqid}
    raise
  end

  private
  def make_reqid
    @reqid += 1
    "#{ip_port}_#{@reqid}"
  end
```

```
end
```

Listing A.2: Server replica and two client implementations for the lattice-based KVS.

Lattice-Based Shopping Cart

Cart Lattice

```
ACTION_OP = 0
CHECKOUT_OP = 1

# The CartLattice represents the state of an in-progress or checked-out shopping
# cart. The cart can hold two kinds of items: add/remove operations, and
# checkout operations. Both kinds of operations are identified with a unique ID;
# internally, the set of items is represented as a map from ID to value. Each
# value in the map is an array, where the first element is either ACTION_OP or
# CHECKOUT_OP.
#
# For ACTION_OPs, the rest of the array contains "item_id" and "mult", where
# mult is the incremental change to the number of item_id's in the cart
# (positive or negative).
#
# For CHECKOUT_OPs, the rest of the array contains "lbound" and "checkout_addr".
# lbound identifies the smallest ID number that must be in the cart for it to be
# complete; we also assume that carts are intended to be "dense" -- that is,
# that a complete cart includes exactly the operations with IDs from lbound to
# the CHECKOUT_OP's ID. checkout_addr is the address we want to contact with the
# completed cart state (we stash it here for convenience). A given cart can have
# at most one CHECKOUT_OP.
#
# Upon an attempt to construct a cart with illegal action messages (e.g.,
# messages with IDs before the lbound or after the checkout message's ID), we
# raise an error. We could instead ignore/drop such messages; this would still
# yield a convergent result. We also raise an error if multiple checkout
# messages are merged into a single cart; this is naturally a non-confluent
# situation, so we need to raise an error.
#
# Why bother with a custom lattice to represent the cart state? The point is
# that checkout becomes a monotonic operation, because each replica of the cart
# can decide when it is "complete" independently (and consistently!).
class CartLattice < Bud::Lattice
  wrapper_name :lcart

  def initialize(i={})
    # Sanity check the set of operations in the cart
    i.each do |k,v|
      reject_input(i) unless (v.class <= Enumerable && v.size == 3)
      reject_input(i) unless [ACTION_OP, CHECKOUT_OP].include? v.first
    end

    checkout = get_checkout(i)
    if checkout
      ubound, _, lbound, _ = checkout.flatten

      # All the IDs in the cart should be between the lbound ID and the ID of
      # the checkout message (inclusive).
      i.each {|k,_| reject_input(i) unless (k >= lbound && k <= ubound) }
    end
```

```
    @v = i
    @is_complete = nil  # computed lazily below
  end

  def merge(i)
    rv = @v.merge(i.reveal) do |k, lhs_v, rhs_v|
      raise Bud::Error unless lhs_v == rhs_v
      lhs_v
    end
    return CartLattice.new(rv)
  end

  monotone :is_complete do
    @is_complete = compute_is_complete if @is_complete.nil?
    Bud::BoolLattice.new(@is_complete)
  end

  monotone :summary do
    @is_complete = compute_is_complete if @is_complete.nil?
    raise Bud::Error unless @is_complete

    actions = @v.values.select {|v| v.first == ACTION_OP}
    summary = {}
    actions.each do |a|
      _, item_id, mult = a
      summary[item_id] ||= 0
      summary[item_id] += mult
    end

    # Drop deleted cart items and return an array of pairs
    summary.select {|_,v| v > 0}.to_a.sort
  end

  monotone :checkout_addr do
    checkout = get_checkout(@v)
    raise Bud::Error unless checkout
    checkout.flatten.last
  end

  private
  def get_checkout(i)
    lst = i.select {|_, v| v.first == CHECKOUT_OP}
    reject_input(i) unless lst.size <= 1
    lst.first   # Return checkout action or nil
  end

  def compute_is_complete
    checkout = get_checkout(@v)
    return false unless checkout

    ubound, _, lbound, _ = checkout.flatten
    (lbound..ubound).each do |n|
      return false unless @v.has_key? n
    end

    return true
  end
end
```

Listing A.3: The `lcart` lattice type used in the Bloom^L shopping cart case study (Section 3.5).

Cart Server Replica

```
module MonotoneCartProtocol
  state do
    channel :action_msg, [:@server, :session, :op_id] => [:item, :cnt]
    channel :checkout_msg, [:@server, :session, :op_id] => [:lbound, :addr]
    channel :response_msg, [:@client, :session] => [:items]
  end
end

module MonotoneReplica
  include MonotoneCartProtocol

  state do
    lmap :sessions
  end

  bloom do
    sessions <= action_msg do |m|
      c = CartLattice.new({m.op_id => [ACTION_OP, m.item, m.cnt]})
      { m.session => c }
    end

    sessions <= checkout_msg do |m|
      c = CartLattice.new({m.op_id => [CHECKOUT_OP, m.lbound, m.addr]})
      { m.session => c }
    end

    # Note that we will send an unbounded number of response messages for each complete cart.
    response_msg <~ sessions.to_collection do |session, cart|
      cart.is_complete.when_true {
        [cart.checkout_addr, session, cart.summary]
      }
    end
  end
end

module MonotoneClient
  include MonotoneCartProtocol

  state do
    table :serv, [] => [:addr]
    scratch :do_action, [:session, :op_id] => [:item, :cnt]
    scratch :do_checkout, [:session, :op_id] => [:lbound]
  end

  bloom do
    action_msg <~ (do_action * serv).pairs do |a,s|
      [s.addr, a.session, a.op_id, a.item, a.cnt]
    end
    checkout_msg <~ (do_checkout * serv).pairs do |c,s|
      [s.addr, c.session, c.op_id, c.lbound, ip_port]
    end
  end
end
```

Listing A.4: A cart server replica that supports monotone checkout (Section 3.5).

A.2 Edelweiss

Fixed Reliable Broadcast

```
class BroadcastAll_Rewrite
  include Bud

  state do
    channel :chn, [:@addr, :id] => [:val]
    channel :chn_ack, [:@rce_sender, :addr, :id]
    range :chn_approx, [:addr, :id]
    table :log, [:id] => [:val]
    sealed :node, [:addr]
    scratch :r0_node_log_joinbuf, [:node_addr, :log_id, :log_val]
    scratch :r0_node_log_missing, [:node_addr, :log_id, :log_val]
    range :seal_log, [:ignored]
    range :seal_node, [:ignored]
  end

  bloom do
    chn <~ (node * log).pairs { |n, l| (n + l) }.notin(chn_approx, 0 => :addr, 1 => :id)
    log <- (log * seal_node).lefts.notin(r0_node_log_missing, :id => :log_id, :val => :log_val)
    node <- (node * seal_log).lefts.notin(r0_node_log_missing, :addr => :node_addr)
    chn_ack <~ chn {|c| [c.source_addr, c.addr, c.id]}
    chn_approx <= chn_ack.payloads
    log <= chn.payloads
    r0_node_log_joinbuf <= (node * log * chn_approx).combos(chn_approx.addr => node.addr,
                                                            chn_approx.id => log.id) {|x,y,z| x + y}
    r0_node_log_missing <= (node * log).pairs {|x,y| x + y}.notin(r0_node_log_joinbuf)
  end
end
```

Listing A.5: Rewritten code produced by Edelweiss for the reliable broadcast program (Listing 4.3).

Epoch-Based Reliable Broadcast

```
class BroadcastEpoch_Rewrite
  include Bud

  state do
    channel :chn, [:@addr, :id] => [:epoch, :val]
    channel :chn_ack, [:@rce_sender, :addr, :id]
    range :chn_approx, [:addr, :id]
    scratch :del_log_r0, [:id] => [:epoch, :val]
    scratch :del_node_r0, [:addr, :epoch]
    table :log, [:id] => [:epoch, :val]
    table :node, [:addr, :epoch]
    scratch :r0_node_log_joinbuf, [:node_addr, :node_epoch, :log_id, :log_epoch, :log_val]
    scratch :r0_node_log_missing, [:node_addr, :node_epoch, :log_id, :log_epoch, :log_val]
    range :seal_log, [:ignored]
    range :seal_log_epoch, [:epoch]
    range :seal_node, [:ignored]
    range :seal_node_epoch, [:epoch]
  end

  bloom do
    chn <~ (node * log).pairs(:epoch => :epoch) { |n, l| ([n.addr] + l) }.notin(chn_approx,
                                                                                0 => :addr, 1 => :id)
```

```
    log <- del_log_r0
    node <- del_node_r0
    chn_ack <~ chn {|c| [c.source_addr, c.addr, c.id]}
    chn_approx <= chn_ack.payloads
    del_log_r0 <= (log * seal_node).lefts.notin(r0_node_log_missing,
                                          :id => :log_id, :epoch => :log_epoch, :val => :log_val)
    del_log_r0 <= (log * seal_node_epoch).lefts(:epoch => :epoch).notin(r0_node_log_missing,
                                                                        :id => :log_id,
                                                                        :epoch => :log_epoch,
                                                                        :val => :log_val)
    del_node_r0 <= (node * seal_log).lefts.notin(r0_node_log_missing,
                                           :addr => :node_addr, :epoch => :node_epoch)
    del_node_r0 <= (node * seal_log_epoch).lefts(:epoch => :epoch).notin(r0_node_log_missing,
                                                                         :addr => :node_addr,
                                                                         :epoch => :node_epoch)
    log <= chn.payloads
    r0_node_log_joinbuf <= (node * log * chn_approx).combos(node.epoch => log.epoch,
                                                            chn_approx.addr => node.addr,
                                                            chn_approx.id => log.id) {|x,y,z| x + y}
    r0_node_log_missing <= (node * log).pairs(node.epoch => log.epoch) {|x,y| x + y}
                                      .notin(r0_node_log_joinbuf)
  end
end
```

Listing A.6: Rewritten code produced by Edelweiss for epoch-based reliable broadcast (Listing 4.4).

Causal Broadcast

```
class BroadcastCausal
  include Bud

  state do
    sealed :node, [:addr]
    table :log, [:id] => [:val, :deps]
    channel :chn, [:@addr, :id] => [:val, :deps]

    table :safe_log, log.schema

    scratch :pending, log.schema
    scratch :flat_dep, [:id, :dep]
    scratch :missing_dep, flat_dep.schema
  end

  bloom do
    chn <~ (node * log).pairs {|n,l| n + l}
    log <= chn.payloads
    pending <= log.notin(safe_log, :id => :id)
    flat_dep <= pending.flat_map {|l| l.deps.map {|d| [l.id, d]}}
    missing_dep <= flat_dep.notin(safe_log, :dep => :id)
    safe_log <+ pending.notin(missing_dep, :id => :id)
  end
end
```

Listing A.7: Causal broadcast in Edelweiss.

Request-Response

```
class RequestResponse
  include Bud
```

```
  state do
    channel :req_chn, [:@addr, :client, :id] => [:key]
    channel :resp_chn, [:@addr, :id] => [:key, :val]

    table :req_log, [:client, :id] => [:key]
    table :resp_log, [:client, :id] => [:key, :val]
    table :did_resp, req_log.key_cols
    table :state

    scratch :need_resp, req_log.schema

    # Client-side state
    table :read_req, req_chn.schema
    table :read_resp, resp_chn.schema
  end

  bloom do
    req_log <= req_chn.payloads
    resp_chn <~ resp_log

    need_resp <= req_log.notin(did_resp, :client => :client, :id => :id)
    resp_log <= (need_resp * state).outer(:key => :key) do |r,s|
      r + [s.val || "MISSING"]
    end
    did_resp <+ resp_log {|r| [r.client, r.id]}
  end

  bloom :client do
    req_chn <~ read_req
    read_resp <= resp_chn
  end
end
```

Listing A.8: Request-response pattern in Edelweiss.