Data-centric Programming for Distributed Systems

by

Peter Alexander Alvaro

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 Rastislav Bodík
Professor Michael Franklin
Associate Professor Tapan Parikh

Fall 2015

Data-centric Programming for Distributed Systems

Abstract

Data-centric Programming for Distributed Systems

by

Peter Alexander Alvaro

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Joseph M. Hellerstein, Chair

Distributed systems are difficult to reason about and program because of fundamental *uncertainty* in their executions, arising from sources of nondeterminism such as asynchrony and partial failure. Until relatively recently, responsibility for managing these complexities was relegated to a small group of experts, who hid them behind infrastructure-backed abstractions such as distributed transactions. As a consequence of technology trends including the rise of cloud computing, the proliferation of open-source storage and processing technologies, and the ubiquity of personal mobile devices, today nearly all non-trivial applications are physically distributed and combine a variety of heterogeneous technologies, including "NoSQL" stores, message queues and caches. Application developers and analysts must now (alongside infrastructure engineers) take on the challenges of distributed programming, with only the meager assistance provided by legacy languages and tools which reflect a single-site, sequential model of computation.

This thesis presents an attempt to avert this crisis by rethinking both the languages we use to implement distributed systems and the analyses and tools we use to understand them. We begin by studying both large-scale storage systems and the coordination protocols they require for correct behavior through the lens of declarative, query-based programming languages. We then use these experiences to guide the design of a new class of "disorderly" programming languages, which enable the succinct expression of common distributed systems patterns while capturing uncertainty in their semantics. We first present DEDALUS, a language that extends Datalog with a small set of temporal operators that intuitively capture *change* and *temporal uncertainty*, and devise a model-theoretic semantics that allows us to formally study the relationship between programs and their *outcomes*. We then develop BLOOM, which provides—in addition to a programmer-friendly syntax and mechanisms for structuring and reuse—powerful analysis capabilities that identify when distributed programs produce *deterministic* outcomes despite widespread nondeterminism in their executions.

On this foundation we develop a collection of program analyses that help programmers to reason about whether their distributed programs produce correct outcomes in the face of asynchrony and partial failure. Blazes addresses the challenge of asynchrony, providing assurances that distributed systems (implemented in Bloom or in parallel dataflow frameworks such as Apache Storm)

produce consistent outcomes in all executions. When it cannot provide this assurance, Blazes augments the system with carefully-chosen synchronization code that ensures deterministic outcomes while minimizing global coordination. Lineage-driven fault injection (LDFI)—which addresses the challenge of partial failure—uses data lineage to reason about whether programs written in DEDALUS or BLOOM provide adequate redundancy to overcome a variety of failures that can occur during execution, including component failure, message loss and network partitions. LDFI can find fault-tolerance bugs using an order of magnitude fewer executions than random fault injection strategies, and can provide guarantees that programs are bug-free for given configurations and execution bounds.

For Beatrice and Josephine

Contents

List of Figures

List of Tables

Listings

Acknowledgments

I principally dedicate this thesis to the two people without whom I could never even have begun, let alone completed it: my advisor Joe Hellerstein and my wife Severine Tymon. Let me share a few words about these two exceptional people.

Were it not for Joe's early and unwavering support, patience and generosity, I could never have started down the road to becoming a researcher. I will never forget the risk he took inviting me—a self-taught programmer with a bachelor's degree in English literature—to join his research team. Joe continually encouraged me to ask ever *bigger* questions; more importantly, he taught me to question my answers; most importantly of all, he trained me to write those answers down carefully. I can only hope to emulate (or at least imitate) Joe's creativity, vision and humility as I start down the road to being a teacher and mentor. Joe also showed me that it is possible to put one's family before everything else and still do world-class research. A lesson like this cannot be taught, but must be lived.

There is only one person who took a bigger risk on me than Joe did: my wife Severine. Not a day goes by that I not reminded of how lucky I am to have her as a partner and a best friend. As luck would have it, our areas of expertise have converged over the years: I discuss my research with her nearly every day, and she has been an exceptional editor and critic. While I was in graduate school Severine supported me with boundless patience; along the way, we built a family together. I don't have words to describe how much I love her, or to demonstrate how profound is her contribution to my life and work.

I am extremely grateful to (and honored by) my thesis committee. Ras Bodík was an early and dedicated collaborator, and played a critical role shaping the design of the DEDALUS and BLOOM languages. Michael Franklin provided invaluable pragmatism and skepticism, helping to keep my research principled and grounded in real use cases. Tapan Parikh provided critical perspective as well as patience.

After my committee, I must thank Divy Agrawal, my first academic mentor. It was Divy who encouraged me to leave the comfort of industry and go back to school: I will never forget it. Thanks to Divy and my first research collaborator, Dmitriy Ryaboy, I published my first research paper.

It was an honor to collaborate with David Maier, who contributed to my work on DEDALUS and Blazes. It was Dave who coined the term 'sealing,' a technique described in Chapter 4. Alan Fekete was a colleague, mentor and friend during his sabbatical at Berkeley.

I was extremely lucky to work with many exceptional researchers during my time at Berkeley. It is difficult to imagine undertaking this thesis effort without the experiences I shared with them. Better still, I am proud to call them friends. In roughly chronological order, they include Tyson Condie, Russell Sears, Neil Conway, William Marczak, Haryadi Gunawi, Peter Bailis, Joshua Rosen and Andrew Hutchinson. Tyson was my first mentor at Berkeley; he continues to be a mentor and friend, as well as a model of an exceptional early career researcher. Rusty was a close collaborator who helped to hone my systems sense. Neil was my closest peer and friend throughout graduate school. Every good idea I had developed further in conversation with him. Only for the past year have I had to carry on without Neil as a partner, and I must admit that it is hard! The countless hours I spent hiking and discussing model theory with Bill Marczak led to the formalization of the semantics of Dedalus. My collaboration with Haryadi on failure testing provided much of the inspiration for my later work on lineage-driven fault injection. Peter is a force of nature: his energy, enthusiasm and vision continue to be an inspiration to me, and I am proud to call him a friend and colleague. Josh Rosen's intuition and programming prowess were fundamental to the success of the LDFI project. In six short months, Andy went from being a student to a coauthor, helping to develop many of the ideas that I later incorporated into Blazes.

Writing papers is hard, and I benefited greatly from the generous help of outside readers. This list includes Sara Alspaugh, Emily Andrews, Peter Bodik, Kuang Chen, Sean Cribs, Raul Castro Fernandez, Armando Fox, Ali Ghodsi, Andy Gross, Kyle Kingsbury, Tim Kraska, Coda Hale, Pat Helland, Chris Meiklejohn, Aurojit Panda, Ariel Rabkin, Alex Rasmussen, Dmitriy Ryaboy, Colin Scott, Evan Sparks, Doug Terry, Alvaro Videla and Shivaram Ventakaran.

My research was supported in part by the National Science Foundation, and gifts from Microsoft Research, EMC, and NTT Multimedia Laboratories.

Chapter 1

Introduction

Modern data-intensive systems are increasingly *distributed* across large collections of machines, both to store and process the rapidly-increasing data volumes now produced by even modest-sized enterprises and to satisfy the growing hunger of analysts for “big data.” Distributed systems are difficult to program and reason about because of two fundamental sources of *uncertainty* in their executions and outcomes. First, due to asynchronous communication, nondeterminism in the ordering and timing of messages delivery can “leak” into program outcomes, leading to data inconsistencies. Second, due to partial failure of components and communication attempts, programs may compute incomplete outcomes or corrupt persistent state. Traditional solutions (e.g., distributed transactions) that hide these complexities from the programmer are considered by many to be untenable at scale, and are increasingly replaced with ad-hoc solutions that trade correctness guarantees for acceptable and predictable performance.

The challenges of programming distributed systems are exacerbated by the fact that they are no longer the sole domain of experts. The relatively recent accessibility of large-scale computing resources (e.g., the public cloud), and proliferation of reusable data management components (e.g., “NoSQL” stores, data processing frameworks, caches and message queues) have created a crisis: all programmers must learn to be distributed programmers. Few tools exist to assist application programmers, data analysts and mobile developers to struggle with these tradeoffs.

In this thesis, we tackle the problem of making distributed systems easier to program and reason about, and report progress in the space of languages, analyses and frameworks. The remainder of this chapter provides an outline of the work presented throughout this thesis.

Disorderly Programming

We hypothesize that many of the challenges of programming distributed systems arise from the mismatch between the sequential model of computation adopted by most programming languages—in which programs are specified as an ordered list of operations to perform on an ordered array of memory cells—and the inherently disorderly nature of distributed systems, in which no total order of events exists. Nondeterminism in message delivery order can cause distributed programs to produce nondeterministic *results*, which complicates testing and debugging. For programs that replicate state, this nondeterminism can lead to replica divergence and other consistency anomalies. Exerting control over delivery order to ensure that sequential programs are executed in lock-step can incur unacceptable performance penalties in a distributed execution.

At the other end of the spectrum, purely "declarative" languages like SQL—which shift the programmer's focus from *computation* to *data*—are set-based and lack the ability to even express ordered operations. In exchange for this limited expressivity, statements in such languages can be safely evaluated in a data-parallel, coordination-free manner.

Disorderly programming—a theme that we explore in this thesis through language design—extends the declarative programming paradigm with a minimal set of ordering constructs. Instead of overspecifying order and then exploring ways to relax it (e.g., by using threads and synchronization primitives), a disorderly programming language encourages programmers to underspecify order, to make it easy (and natural) to express safe and scalable computations. As we will show, disorderly programming also enables powerful analysis techniques that recognize when additional ordering constraints are required to produce correct results. Mechanisms ensuring these constraints can then be expressed and inserted at an appropriately coarse grain to achieve the needs of core tasks like mutable state and distributed coordination.

Data-centric system design

Chapter 2 details how the disorderly programming hypothesis arose from my research team's experiences using *query languages* to implement distributed systems. The BOOM (Berkeley Orders Of Magnitude) project explored the conjecture that using *data-centric* programming techniques and *declarative* languages could dramatically simplify the implementation, maintenance and evolution of large-scale distributed systems. To validate this conjecture, we used a distributed logic programming language based on Datalog to implement two layers of a modern storage stack. First, we implemented the Paxos consensus protocol; Section 2.2 shows how natural it was to translate the specification to a high-level implementation. Next, we used the same language to implement BOOM-FS, an API-complaint HDFS clone [13], which performed competitively with the reference implementation despite having been written in orders of magnitude less code. Section 2.3 then describes how we extended this simple core with features previously unavailable in HDFS, including a Paxos-backed replicated master [15] for high availability, a partitioned namespace for scalability and state-of-the-art tracing and monitoring facilities.

Dedalus: A formal disorderly language

Despite these successes applying a declarative language to protocol and application implementation, the early generation of our distributed logic languages were fraught with semantic difficulties. While ideally suited to expressing relationships among data elements (and hence, according to the disorderly programming hypothesis, the majority of interesting distributed computation), traditional query languages cannot unambiguously express relationships *between states* in time. Such relationships (e.g., atomicity, sequentiality, mutual exclusion and state mutation) are required to unambiguously specify or implement critical coordination protocols and algorithms, including locking, atomic commit and consensus. Worse still, the semantics of existing query languages failed to capture the fundamental uncertainty in distributed executions, in which the consequences of certain deductions can be lost or arbitrarily delayed. Together, these weaknesses made it challenging to reason about whether our applications and protocols upheld their correctness contracts in all possible distributed executions.

Section 3.1 describes the disorderly programming language Dedalus [20, 127], which extends Datalog with a small set of temporal operators meant to intuitively capture *state change* and *uncertainty*—the signature features of distributed systems—within a relational logic paradigm. Dedalus preserves many of the key benefits of earlier distributed query languages: by uniformly treating state, events, and messages as *data*, it encourages high-level, disorderly implementations of distributed protocols and applications, delivering the benefits of the declarative programming paradigm. More importantly, by capturing asynchronous communication and the possibility of failure in its formal semantics, Dedalus lays a foundations that enables the formal study of the relationship between logical semantics and consistency in distributed systems.

Bloom: A disorderly language for distributed programming

Dedalus programs are executable, despite the fact that they read like specifications and are amenable (as we shall see) to powerful analysis techniques. Nevertheless Dedalus lacks many desirable features for a practical programming language, including mechanisms for encapsulation and reuse and a syntax familiar to programmers. Section 3.3 presents Bloom [17, 52, 51], a general-purpose programming language that provides these and other usability features while preserving the semantic core of Dedalus.

More importantly, Bloom realizes a group of analyses and tools that we collectively refer to as *confluence analysis*, which provide programmers with strong guarantees that their distributed programs are robust to uncertainty in their executions.

Confluence Analysis

If we assume that all message are eventually delivered (an assumption that we will relax later in the thesis), it is easy to see that programs that compute the same result for all inputs orderings are *eventually consistent* [166]. Disorderly programming languages make writing order-insensitive programs the natural mode, and hence make it it easy to write eventually consistent programs.

However, in order to remain sufficiently general to program arbitrary systems, such languages cannot prevent programmers from writing programs that are (by design) sensitive to message ordering. Therefore disorderly languages are an incomplete solution unless they provide analyses that recognize exactly when a program's outcomes may depend upon particular delivery orderings. When programs are order-sensitive—that is, when nondeterminism in delivery order can lead to nondeterminism in outcomes—these analyses can identify efficient repair strategies that ensure deterministic results.

Given the foundation of disorderly languages for programming distributed systems, the remainder of this thesis focuses on programming tools that help tame distributed uncertainty and provide assurances about program outcomes.

Monotonicity Analysis: Ensuring consistent outcomes despite asynchrony

Some programs are robust in the face of uncertainty, producing deterministic outcomes despite pervasive nondeterminism in their executions. *Monotonicity analysis*, described in Section 3.3 identifies such programs by focusing on when program logic causes distributed data to change in predictable ways regardless of uncertainty in delivery and scheduling timing.

A unique feature of the Bloom language is its ability to perform a static "consistency analysis" of submitted programs, providing visual programmer feedback (based on an *annotated dataflow graph* representation of the program logic) that identifies computations that could produce nondeterministic results when evaluated in a distributed system. This static analysis is based on the CALM Theorem [17], which establishes that monotonic programs produce deterministic outcomes despite nondeterminism in message delivery order. Because both nonmonotonic (i.e., order-sensitive) operations and asynchronous (i.e. order-sacrificing) communication are exposed in Bloom's syntax, monotonicity analysis can both warn programmers of the potential for inconsistent outcomes and *pinpoint* individual program statements as repair candidates.

Blazes: coordination analysis and synthesis

Monotonicity analysis is essentially a filter; programs that pass are guaranteed to have coordination-free deterministic outcomes in all executions. However, intuition tells us that some programs simply *require* coordination. Large-scale systems are likely to involve a combination of order-sensitive and order-insensitive components—in such cases, must we throw out the baby with the bathwater and fall back on a classic "strongly consistent" system architecture? Can we exploit monotonicity and other application-specific semantics to achieve "minimally-coordinated" executions?

Blazes [16, 12] extends monotonicity analysis into a language-independent framework that not only identifies potential consistency anomalies in under-coordinated systems, but remedies them by augmenting the given program with judiciously-chosen coordination code. Blazes' analysis is based on a pattern of properties and composition: it begins with key properties of individual software components, including order-sensitivity, statefulness, and replication; it then reasons transitively about compositions of these properties across dataflows that span components. Second, Blazes automatically *generates* application-aware coordination code to prevent consistency anomalies with a minimum of coordination. The key intuition exploited by Blazes is that even when components are order-sensitive, it is often possible to avoid the cost of global ordering without sacrificing consistency. In many cases, Blazes can ensure consistent outcomes via a more efficient and manageable protocol of asynchronous point-to-point communication between producers and consumers—called *sealing*—that indicates when partitions of a stream have stopped changing. These partitions are identified and "chased" through a dataflow via techniques from functional dependency analysis.

When systems are written in Bloom or Dedalus, Blazes utilizes static monotonicity analysis to identify code segments as sources of potential consistency anomalies, and hence candidates for repair. If the system is written in a language for which monotonicity analysis is not available (e.g. Apache Storm components implemented in Java), Blazes relies on simple semantic *annotations* provided by the user that characterize the order-sensitivity and statefulness of each component.

Lineage-driven fault injection

The research described so far focused on the problem of *asynchrony* in distributed systems, identifying programs that are tolerant to message reordering and choosing efficient coordination strategies for programs that are not. If all nodes continue operating, and all messages are eventually delivered, monotonicity analysis and tools such as Blazes can provide programmers with strong assurances about program correctness. But how does a programmer ensure that all messages are eventually delivered? The lineage-driven fault injection (LDFI) project (described in Chapter 5) rounds out the suite of tools by ensuring that distributed programs are *fault-tolerant*—that is, that they produce correct outcomes even when a variety of failures (such as message loss, node failure and network partition) occur during their execution.

Like Blazes and Bloom consistency analysis, LDFI [21] takes advantage of the clean semantic core of Dedalus to reason about the effects of nondeterminism arising in a distributed execution. However, instead of using program syntax to reason statically about the space of possible executions, LDFI uses *data lineage* produced in concrete executions to directly connect system outcomes to the data and messages that led to them. Fine-grained lineage allows LDFI to reason *backwards* (from effects to causes) about whether a given correct outcome could have failed to occur due to some combination of faults. Rather than generating faults at random (or using application-specific heuristics), LDFI chooses only those failures that could have affected a known good outcome, exercising fault-tolerance code at increasing levels of complexity. Injecting failures in this targeted way allows LDFI to provide completeness guarantees like those achievable with formal methods such as model checking. When bugs are encountered, LDFI's top-down approach provides—in addition to a counterexample trace—fine-grained data lineage visualizations to help programmers understand the *root cause* of the bad outcome and consider possible remediation strategies.

Code examples

Over the course of this thesis, we ground our discussion in executable code examples, including program fragments, reusable modules and complete applications. As the thesis progresses, the presentation language is steadily refined.

All of the language variants are extensions of Datalog¬ [165], extended with scalar and aggregate functions. In Chapter 2, code examples are given in the Overlog language, which adds the ability to declaratively specify network communication. In Section 3.1, we develop the Dedalus language and present examples in increasingly rich variants of Dedalus; in Section 3.3 and Chapter 4, examples are given in the more refined Bloom language. In Chapter 5, in which the examples focus on protocol specification, we return to the succinct and germane DEDALUS language for examples.

Chapter 2

Disorderly programming: a hypothesis

It is very similar to the wedding ceremony in which the minister asks "Do you?" and the participants say "I do" (or "No way!") and then the minister says "I now pronounce you," or "The deal is off."
– Jim Gray, describing the two-phase commit protocol [73]

Until fairly recently, distributed programming was the domain of a small group of experts. However, a variety of recent technology trends have brought distributed programming to the mainstream of open source and commercial software. The challenges of distribution—concurrency and asynchrony, performance variability, and partial failure—often translate into tricky data management challenges regarding task coordination and data consistency. Given the growing need to wrestle with these challenges, there is increasing pressure on the data management community to help find solutions to the difficulty of distributed programming.

Although distributed programming remains hard today, one important subclass is relatively well-understood by programmers: data-parallel computations expressed using interfaces like MapReduce and SQL [57, 89, 1, 110]. These "disorderly" programming models substantially raise the level of abstraction for programmers: they mask the coordination of threads and events, and instead ask programmers to focus on applying functional or logical expressions to collections of data. These expressions are then auto-parallelized via a dataflow runtime that partitions and shuffles the data across machines in the network. Although easy to learn, these programming models have traditionally been restricted to batch-oriented computations and data analysis tasks—a rather specialized subset of distributed and parallel computing.

Inspired by the success of data-parallel programming, this thesis work began with a series of questions. Could we transform the notoriously difficult problem of distributed programming into the well-understood problem of data-parallel querying? Is it possible to build "real" distributed infrastructure—such as the data-parallel frameworks described above—using a query language? Does this approach generalize across the stack, from applications to infrastructures to low-level protocols? We hypothesized that with the right collection of programming abstractions, we could answer all of these questions in the affirmative. This chapter describes our successes confirming this hypothesis by using an existing relation logic language to implement both protocols (two-phase commit and Paxos) and a large-scale storage system (BOOM-FS).

Section 2.1 briefly describes Overlog, a Datalog-based language that we used to explore the disorderly programming hypothesis. Section 2.2 recounts our experiences using disorderly programming to implement distributed systems *in the small*: tackling safety-critical commit and consensus protocols. We observe that the abstractions provided by a rule-based query language such as Overlog map directly to the concepts described in protocol specifications, including state invariants and event dispatch rules. We also describe how the experience led us to think about protocol variants in terms of the composition of higher-level programming *idioms*, such as voting, sequences and choice. The desire to capture and reuse these idioms motivates the design of Bloom, presented in Section 3.3. Finally, Section 2.3 presents lessons from our experience applying the hypothesis to distributed systems *in the large*: a complete rewrite of a cluster filesystem. There, we show how combining a data-centric programming style with a disorderly language can yield systems that are substantially easier to implement, easier to understand, and easier to extend than the state of the art, without sacrificing performance.

2.1 Background: Overlog

Overlog is a language for *declarative networking* developed at UC Berkeley [119, 118, 116, 117]. The goal of the declarative networking project was to combine data-centric design with declarative programming to allow the concise, high-level expression of routing protocols. Because the transitive closure of a reachability relation—the basis of a majority of routing protocols—is easily expressed as a recursive query, Overlog was an excellent fit for the domain.

Overlog is a recursive query language based on Datalog [165] enhanced with communication primitives, integrity constraints, and aggregate and scalar functions. Datalog programs consist of rules that take the form:

```
head(A, C) :- clause1(A, B), clause2(B, C);
```

where `head`, `clause1`, and `clause2` are relations, "`:-`" denotes implication ($\Leftarrow$) and "`,`" denotes conjunction. A rule may have any number of clauses, but only a single head. Variables are denoted by identifiers that begin with an uppercase letter, or by the symbol "_", which indicates that the value of the variable will not be used in the rule. The example rule ensures that the `head` relation contains a tuple $\{A, C\}$ for each tuple $\{A, B\}$ in `clause1` and $\{B, C\}$ in `clause2` where the tuples have the same value for B. In practice, it does so by computing the join of `clause1` and `clause2` on B, and projecting A and C. A Datalog program begins with some base tuples, and derives new tuples by evaluating rules in a bottom-up fashion (substituting tuples in the clause relations to derive new tuples in the head relations) until no more derivations can be made. Such a computation is called a *fixpoint*. A set of rules essentially expresses the constraint that base facts and their transitive consequences will always be consistent at fixpoint.

Overlog computes a new fixpoint whenever new tuples arrive at a node. Overlog programs accept input from network events, timers, and native methods, each of which may produce new tuples. Because evaluation of an Overlog program proceeds in discrete time steps, rules may be interpreted as *invariants* over state: the consistency of the rule specifications will be true at every fixpoint.

Network communication is expressed using a simple extension to the Datalog syntax:

```
recv_msg(@A, Payload) :-
  send_msg(@B, Payload), peers(@B, A);
```

`@` denotes the *location specifier* field of a relation, which indicates that the associated variables *A* and *B* contain network addresses. Location specifiers in the body clauses are required to be identical, to capture the physical constraint that computation cannot occur on data stored at a remote network endpoint. A tuple moves between nodes if the location specifier in the head relation is distinct from those of the body [1]

It is often useful to compute an aggregate over a set of tuples, typically to choose an element of the set with a particular property (e.g. `min`, `max`) or to compute a summary statistic over the set (e.g. `count`, `sum`). For example:

```
least_msg(min<SeqNum>) :-
  queued_msgs(SeqNum, _);
```

defines an aggregate relation that contains the smallest sequence number among the queued messages, and

```
next_msg(Payload) :-
  queued_msgs(SeqNum, Payload),
  least_msg(SeqNum);
```

states that the content of `next_msg` is the payload of the queued message with the smallest sequence number. This pair of rules is equivalent to the SQL statement:

```
SELECT payload FROM queued_msgs
WHERE seqnum =
    (SELECT min(seqnum) FROM queued_msgs);
```

We encountered this pattern of selection over aggregation frequently when implementing protocols in Overlog.

Finally, Overlog allows special `timer` relations to be defined. The Overlog runtime inserts a tuple into each timer relation at a user-defined period, and the predicate holds only at these intervals. Thus, joining against a timer relation enforces periodic evaluation of a rule.

[1] Note however that rules of this kind do *not* establish state invariants: when the premises of such a rule hold, the conclusions hold at some unknown future time (if ever). Hence computation is local and communication is explicit. We return to this subject in Section 3.1

2.2 Commit and Consensus Protocols

Our first foray into using Overlog as a general-purpose programming language for distributed systems involved tackling a notoriously complicated domain: consensus protocols. Consensus protocols are a common building block for fault-tolerant distributed systems [62]. Paxos is a widely-used consensus protocol, first described by Lamport [106, 107]. While Paxos is conceptually simple, practical implementations are difficult to achieve, and typically require thousands of lines of carefully written code [39, 98, 130].

Much of this implementation difficulty arises because high-level protocol specifications must be translated into low-level imperative code, yielding a significant increase in program size and complexity. In practical implementations of Paxos, the simplicity of the consensus algorithm is obscured by common but often tricky details such as event loops, timer interrupts, explicit concurrency, and the serialization and persistence of data structures.

By contrast, consensus protocols such as two-phase commit and Paxos are specified in the literature at a high level, in terms of messages, invariants, and state machine transitions. Overlog supports all but the last of these concepts directly—we address the difficulty of modeling time-varying state in Section 2.5 and propose a solution in Section 3.1. By using a declarative language to implement consensus protocols, we hoped to achieve a more concise implementation that is conceptually closer to the original protocol specification. We discuss our Paxos implementation below, and describe how we mapped concepts from the Paxos literature into executable Overlog code. We reflect on the design patterns that we discovered while building this classical distributed service in a declarative language. The process of identifying these patterns helped us better understand why a declarative networking language is well-suited to programming distributed systems. It also clarified our thinking about the more general challenge of designing a language for distributed computing.

Two-phase commit

Before tackling Paxos, we used Overlog to build two-phase commit (2PC), a simple agreement protocol that decides on a series of Boolean values ("commit" or "abort"). Unlike Paxos, 2PC does not attempt to make progress in the face of node failures.

Both Paxos and 2PC are based on rounds of messaging and counting. In 2PC, the coordinator node communicates the intent to commit a transaction to the peer nodes. The peer nodes attempt execute the transaction without making its effects visible. If they succeed, they transition to the "prepare" phase and send a "yes" vote to the coordinator; otherwise, they send a "no" vote. The coordinator counts these responses; if all peers respond "yes" then the transaction commits. Otherwise it aborts. In terms of the Overlog primitives described above, this is just messaging, followed by a `count` aggregate, and a selection for the string "no" in the peers' responses.

The mechanism that implements this protocol follows directly from the specification (Listing 2.1). The `peer_cnt` table contains the coordinator address and the number of peers. When `vote` messages arrive, the second and fourth rules are considered. If the fourth rule is satisfied (with a single "no" vote), the transaction state is updated to "abort"; otherwise, `yes_cnt` is incre-

```
/* Count number of peers */
peer_cnt(Coordinator, count<Peer>) :-
  peers(Coordinator, Peer);

/* Count number of "yes" votes */
yes_cnt(Coordinator, TxnId, count<Peer>) :-
  vote(Coordinator, TxnId, Peer, Vote),
  Vote == "yes";

/* Prepare => Commit if unanimous */
transaction(Coordinator, TxnId, "commit") :-
  peer_cnt(Coordinator, NumPeers),
  yes_cnt(Coordinator, TxnId, NumYes),
  transaction(Coordinator, TxnId, State),
  NumPeers == NumYes, State == "prepare";

/* Prepare => Abort if any "no" votes */
transaction(Coordinator, TxnId, "abort") :-
  vote(Coordinator, TxnId, _, Vote),
  transaction(Coordinator, TxnId, State),
  Vote == "no", State == "prepare";

/* All peers know transaction state */
transaction(@Peer, TxnId, State) :-
  peers(@Coordinator, Peer),
  transaction(@Coordinator, TxnId, State);
```

Listing 2.1: 2PC coordinator protocol in Overlog. The DDL for `transaction` (not shown) specifies that the first two columns are a primary key.

```
/* Declare a timer that fires once per second */
timer(ticker, 1000ms);

/* Start counter when TxnId is in "prepare" state */
tick(Coordinator, TxnId, Count) :-
  transaction(Coordinator, TxnId, State),
  State == "prepare",
  Count := 0;

/* Increment counter every second */
tick(Coordinator, TxnId, NewCount) :-
  ticker(),
  tick(Coordinator, TxnId, Count),
  NewCount := Count + 1;

/* If not committed after 10 sec, abort TxnId */
transaction(Coordinator, TxnId, "abort") :-
  tick(Coordinator, TxnId, Count),
  transaction(Coordinator, TxnId, State),
  Count > 10, State == "prepare";
```

Listing 2.2: Timeout-based abort. The first two columns of `tick` are a primary key.

mented by the second rule to reflect another positive vote for this transaction. If `yes_cnt` equals `peer_cnt`, the vote is unanimous and the transaction moves to the "commit" state. The fifth rule continuously communicates changes to transaction state to every peer node.

A practical 2PC implementation must address two additional details: timeouts and persistence. Timeouts allow the coordinator to unilaterally choose to abort the transaction if the peers take too long to respond. This is straightforward to implement using timer relations (Listing 2.2).

The complete 2PC protocol is naturally specified in terms of aggregation, selection, messaging, timers, and persistence. Focusing on these details led to an implementation whose size and complexity resemble a specification.

Discussion

As we employed the primitives of messaging, timers and aggregation to implement 2PC, we found ourselves reasoning in terms of higher-level constructs that were more appropriate to the domain. We call these higher-level constructs "idioms", and denote them with italics.

Multicast, a frequently occurring pattern in consensus protocols, can be implemented by composing the messaging primitive described in Section 2.1 with a join against a relation containing the membership list. The last rule in Listing 2.1 implements a multicast.

The `tick` relation introduced in Listing 2.2 implements a *sequence*, a single-row relation whose attribute values change over time. A *sequence* is defined by a base rule that initializes the counter attribute of interest, and an inductive rule that increments this attribute. Combining this pattern with timer relations allows an Overlog program to count the number of clock ticks, and therefore the number of seconds, that have elapsed since some event. This is the basis of our *timeout* mechanism (Listing 2.2).

A coordinator and a set of peers can participate in a *roll call* to discover which peers are alive by combining a coordinator-side multicast with a peer-side unicast response. A `count` aggregate over a table containing network messages implements a *barrier*: the partial count of rows in the table increases with each received message, and synchronization is achieved when the count is high enough. A round of *voting* is a roll call with a selection at each peer (which vote to cast, probably implemented as selection over aggregation) and a *barrier* at the coordinator. The first three rules listed in Listing 2.1 are an example of the voting idiom.

Even with a simple protocol like 2PC, a variety of common distributed design patterns emerge quite naturally from the high-level Overlog specification. We now turn to Paxos, a more complicated protocol, to see if these patterns remain sufficient.

Paxos

Like 2PC, Paxos uses rounds of voting between a leader and a set of participating agents to decide on an update. Unlike 2PC, these roles are not fixed, but may change as a result of failures: this is central to Paxos' ability to make forward progress in the face of failures, even of the leader. We began by implementing the Synod protocol, which reaches consensus on a single update, and then extended it to make an unbounded series of consensus decisions ("Multipaxos"). In this section,

```
promise(@Master, View, OldView, OldUpdate, Agent) :-
  prepare(@Agent, View, Update, Master),
  prev_vote(@Agent, OldView, OldUpdate),
  View >= OldView;
```

Listing 2.3: An agent sends a constrained promise if it has voted for an update in a previous view.

```
agent_cnt(Master, count<Agent>) :-
  parliament(Master, Agent);

promise_cnt(Master, View, count<Agent>) :-
  promise(Master, View, Agent, _);

quorum(Master, View) :-
  agent_cnt(Master, NumAgents),
  promise_cnt(Master, View, NumVotes),
  NumVotes > (NumAgents / 2);
```

Listing 2.4: We have quorum if we have collected promises from more than half of the agents.

we describe our Paxos implementation in terms of the idioms we identified for 2PC, and detail additional constructs that we found necessary.

Prepare Phase

Paxos is bootstrapped by the selection of a leader and an accompanying view number: this is called a view change. To initiate a view change, a would-be leader *multicasts* a `prepare` message to all agents; this message includes a *sequence* number that is globally unique and monotonically increasing.

The Paxos protocol dictates that when an agent receives a `prepare` message, if it has already voted for a lower view number, it must send a constrained `promise` message containing the update associated with its previous vote. Otherwise, it must send an unconstrained `promise` message, indicating that it is willing to pass any update the leader proposes. This invariant couples requests with history, and is implemented with a query that joins the `prepare` stream with the local `prev_vote` relation (Listing 2.3). Finally, the prospective leader performs a `count` aggregate over the set of `promise` messages; if it has received responses from a majority of agents then the new view has quorum (Listing 2.4). In sum, the prepare phase employs the idioms of *sequences, multicast* and *barriers*.

Voting Phase

Once leadership has been established through a view change, the new leader performs a query to see if any responses constrain the update. If so, the leader chooses an update from one of the constraining responses (by convention, it uses a `max` aggregate over the view numbers in the `promise` messages). In the absence of constraining responses, it is free to choose any pending update.

The remainder of the voting phase is a generalization of 2PC. The leader multicasts a `vote` message, containing the current view number and the chosen update, to all agents in the view. Each agent joins this message against a local relation containing the agent's current view number. If the two agree, it responds with an `accept` message. An update is committed once it has been accepted by a quorum of agents; when the leader detects this, it responds to the client who initiated the update. This second phase of Lamport's original Paxos is a straightforward composition of *multicast* and *barriers*.

```
top_of_queue(Agent, min<Id>) :-
  stored_update_request(Agent, _, _, Id);

/* Select the enqueued update with the lowest Id */
begin_prepare(Agent, Update) :-
  stored_update_request(Agent, Update, _, Id),
  top_of_queue(Agent, Id);

/* Cleanup passed updates, causing top_of_queue
   to be refreshed */
delete
stored_update_request(Agent, Update, From, Id) :-
  stored_update_request(Agent, Update, From, Id),
  update_passed(Agent, _, _, Update, Id);
```

Listing 2.5: Choice and Atomic Dequeue.

Multipaxos

Multipaxos extends the algorithm described above to pass an ordered sequence of updates, and requires the introduction of additional state to capture the update history. A practical implementation performs the `prepare` phase once, and assuming a stable leader, carries out many instances of the voting phase.

Accommodating the notion of instances is a straightforward matter of schema modification. A `prepare` message now includes an instance number indicating the candidate position of the update in the globally ordered log. Each agent records the current instance number, and `promise` and `accept` message transmission is further constrained by joining against this relation: an agent only votes for a proposed update if its sequence number agrees with the current local high-water mark.

Leader Election

Leader election protocols choose Multipaxos leaders, typically in response to leader failure. Detection of leader failure is usually implemented with timeouts: if no progress has been made for a certain period of time, the current leader is presumed to be down and a new leader is chosen. We implemented the leader election protocol of Kirsch and Amir [98] in 19 Overlog rules (the original specification required 31 lines of pseudocode). Our implementation of leader election was based on aggregation, *multicast*, *sequences* and *timeouts*, and left the core of our Multipaxos code unchanged.

Discussion

Most of the logic of the basic Paxos algorithm is captured by combining *voting* with a *sequence* that allows us to distinguish new from expired views. Hence the idioms we described in our treatment of 2PC were nearly sufficient to express this significantly more complicated consensus protocol. As we reflected on our implementation, two new idioms emerged.

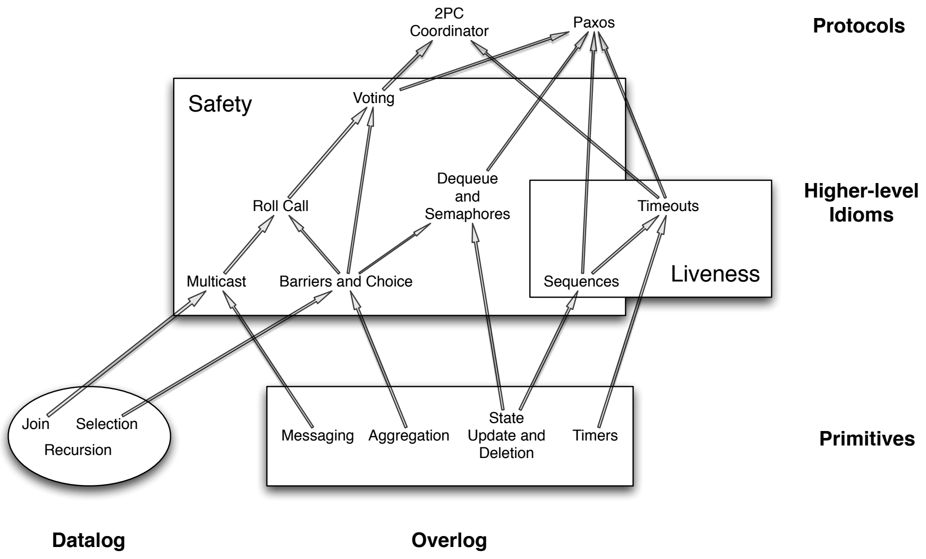


Figure 2.1: Distributed Logic Programming Idioms.

Rule Pattern	Idiom	Prepare	Propose	Election
All		13	13	19
Messages	Multicast	1	2	2
	Other	1	1	0
State Update	Sequence	2	2	3
	GC	1	3	2
	Other	0	1	6
Aggregation	Barrier	1	1	1
	Choice	2	1	0
	Other	5	2	3
Timer	Timeout	0	0	2

Table 2.1: The usage of Overlog primitives and idioms in our Paxos implementation.

Using an exemplary aggregate function like `min` in combination with selection implements a *choice* construct that selects a particular tuple from a set. In Paxos, this construct is necessary for the leader's choice of a constrained update during the `prepare` phase. Combining the choice pattern with a conditional delete rule against the base relation allows us to express an *atomic dequeue* operation, which is useful for implementing data structures such as FIFOs, stacks, and priority queues. We found this construct useful as a flow control mechanism, to ensure that at most one tuple enters the `prepare` phase dataflow at a time (Listing 2.5).

Figure 2.1 illustrates the composition of the distributed programming idioms we encountered while implementing 2PC and Paxos. To avoid clutter, not every connection is drawn; for example, selection and join occur in nearly every idiom. Instead, we draw connections to emphasize interesting relationships: join combines with messaging to implement *multicast*, and selection and aggregation combine to implement *choice*. Unanimity, the critical safety property of 2PC, is enforced via a *vote* construct, as is the quorum constraint in both phases of Paxos. The safety of Paxos also relies on the invariant that an update accepted by any agent must be accepted by all: this is maintained by the *choice* of a constrained update in the `prepare` phase. Figure 2.1 shows the breakdown of our Paxos implementation in terms of Overlog primitives and higher-level idioms. Idioms like *multicast, barriers* and *sequences* occur ubiquitously, and aggregation is the most common rule pattern in all modules. "GC" denotes garbage collection rules that explicitly delete tuples that are no longer needed. As we would expect, timer logic is relegated to the time-aware leader election module. When we set out to implement Paxos, our intent was to experiment with the generality of what was originally envisioned as a declarative networking language. As the P2 authors discovered for network protocols [120], we found that a few simple Overlog idioms cover an impressive amount of the design space for consensus protocols. The correspondence between Overlog idioms and consensus protocol specifications allowed us to directly reason about the correctness of our implementations.

In the course of our implementation, we saw significant reuse of higher-level constructs than those provided by Overlog. When comparing protocols, we found ourselves speaking in terms of *barriers*, *voting*, *choice* and *sequences* rather than select, project and join.

The use of these idioms made it easier to focus on the relatively small distinctions between protocol variants like 2PC, Synod and the Multipaxos variants, such as the conditions under which agents cast votes and when barriers may be passed. At a conceptual level, this highlights commonalities between these protocols that deserve more attention; at a constructive level, it motivates the need for a language in which key idioms can be expressed once and composed in a variety of ways. We present such a language—Bloom—in Section 3.3.

2.3 BOOM-FS

To further evaluate the feasibility of the BOOM vision beyond distributed protocols, we used Overlog to build an API-compliant reimplementation of the Hadoop distributed file system (HDFS) called BOOM-FS. We preserved the Java API "skin" of HDFS, but replaced its complex internals with Overlog. The Hadoop stack appealed to us for two reasons. First, it exercises the distributed power of a cluster. Unlike a farm of independent web service instances, the Hadoop and HDFS code entails coordination of large numbers of nodes toward common tasks. Second, Hadoop is a work in progress, still missing significant distributed features like availability and scalability of master nodes. The difficulty of adding these complex features could serve as a litmus test of the programmability of our approach.

The bulk of this section describes our experience implementing and evolving BOOM-FS, and running it on Amazon EC2. After two months of development, BOOM-FS performed as well as vanilla HDFS, and enabled us to easily develop complex new features including Paxos-supported replicated-master availability, and multi-master state-partitioned scalability. We describe how a data-centric programming style facilitated debugging of tricky protocols, and how by metaprogramming Overlog we were able to easily instrument our distributed system at runtime. Our experience implementing BOOM-FS in Overlog was gratifying both in its relative ease, and in the lessons learned along the way: lessons in how to quickly prototype and debug distributed software, and in understanding limitations of Overlog that may contribute to an even better programming environment for datacenter development.

This section presents the evolution of BOOM-FS from a straightforward reimplementation of HDFS to a significantly enhanced system. We describe how an initial prototype went through a series of major revisions ("revs") focused on *availability* (Section 2.3), *scalability* (Section 2.3), and *debugging and monitoring* (Section 2.3). In each case, the modifications involved were both simple and well-isolated from the earlier revisions. In each section we reflect on the ways that the use of a high-level, data-centric language affected our design process.

Implementation

HDFS is loosely based on GFS [69], and is targeted at storing large files for full-scan workloads. In HDFS, file system metadata is stored at a centralized *NameNode*, but file data is partitioned into 64MB chunks and distributed across a set of *DataNodes*. Each chunk is typically stored at three DataNodes to provide fault tolerance. DataNodes periodically send heartbeat messages to the NameNode that contain the set of chunks stored at the DataNode. The NameNode caches this information. If the NameNode has not seen a heartbeat from a DataNode for a certain period of time, it assumes that the DataNode has crashed and deletes it from the cache; it will also create additional copies of the chunks stored at the crashed DataNode to ensure fault tolerance.

Clients only contact the NameNode to perform metadata operations, such as obtaining the list of chunks in a file; all data operations involve only clients and DataNodes. HDFS only supports file read and append operations — chunks cannot be modified once they have been written.

BOOM-FS In Overlog

We built BOOM-FS from scratch in Overlog, limiting our Hadoop/Java compatibility task to implementing the HDFS client API in Java. We did this by creating a simple translation layer between Hadoop API operations and BOOM-FS protocol commands. The resulting BOOM-FS implementation works with either vanilla Hadoop MapReduce or with BOOM-MR [14], an Overlog implementation of Mapreduce that our research group built concurrently with this work.

Like GFS, HDFS maintains a clean separation of control and data protocols: metadata operations, chunk placement and DataNode liveness are cleanly decoupled from the code that performs bulk data transfers. This made our rewriting job substantially more attractive. We chose to implement the simple high-bandwidth data path "by hand" in Java, and used Overlog for the trickier but lower-bandwidth control path.

Name	*Description*	*Relevant attributes*
file	Files	fileid, parentfileid, name, isDir
fqpath	Fully-qualified pathnames	path, fileid
fchunk	Chunks per file	chunkid, fileid
datanode	DataNode heartbeats	nodeAddr, lastHeartbeatTime
hb_chunk	Chunk heartbeats	nodeAddr, chunkid, length

Table 2.2: BOOM-FS relations defining file system metadata.

File System State

The first step of our rewrite was to represent file system metadata as a collection of relations. We then implemented file system policy by writing queries over this schema. A simplified version of the relational file system metadata in BOOM-FS is shown in Table 2.2.

The *file* relation contains a row for each file or directory stored in BOOM-FS. The set of chunks in a file is identified by the corresponding rows in the *fchunk* relation.[2] The *datanode* and *hb_chunk* relations contain the set of live DataNodes and the chunks stored by each DataNode, respectively. The NameNode updates these relations as new heartbeats arrive; if the NameNode does not receive a heartbeat from a DataNode within a configurable amount of time, it assumes that the DataNode has failed and removes the corresponding rows from these tables.

The NameNode must ensure that the file system metadata is durable, and restored to a consistent state after a failure. This was easy to implement using Overlog, because of the natural atomicity boundaries provided by fixpoints. We used the Stasis storage library [150] to flush durable state changes to disk as an atomic transaction at the end of each fixpoint. Since Overlog allows durability to be specified on a per-table basis, the relations in Table 2.2 were marked durable, whereas "scratch tables" that are used to compute responses to file system requests were transient.

Since a file system is naturally hierarchical, a recursive query language like Overlog was a natural fit for expressing file system policy. For example, an attribute of the *file* table describes the parent-child relationship of files; by computing the transitive closure of this relation, we can infer the fully-qualified pathname of each file (*fqpath*). (The two Overlog rules that derive *fqpath* from *file* are listed in Listing 2.6.) Because this information is accessed frequently, we configured the *fqpath* relation to be cached after it is computed.

At each DataNode, chunks are stored as regular files on the file system. In addition, each DataNode maintains a relation describing the chunks stored at that node. This relation is populated by periodically invoking a table function defined in Java that walks the appropriate directory of the DataNode's local file system.

Communication Protocols

BOOM-FS uses three different protocols: the *metadata protocol* which clients and NameNodes use to exchange file metadata, the *heartbeat protocol* which DataNodes use to notify the NameNode

[2]The order of a file's chunks must also be specified, because relations are unordered. Currently, we assign chunk IDs in a monotonically increasing fashion and only support append operations, so clients can determine a file's chunk order by sorting chunk IDs.

```
// fqpath: Fully-qualified paths.
// Base case: root directory has null parent
fqpath(Path, FileId) :-
    file(FileId, FParentId, _, true),
    FParentId = null, Path = "/";

fqpath(Path, FileId) :-
    file(FileId, FParentId, FName, _),
    fqpath(ParentPath, FParentId),
    // Do not add extra slash if parent is root dir
    PathSep = (ParentPath = "/" ? "" : "/"),
    Path = ParentPath + PathSep + FName;
```

Listing 2.6: Example Overlog for computing fully-qualified pathnames from the base file system metadata in BOOM-FS.

```
// The set of nodes holding each chunk
compute_chunk_locs(ChunkId, set<NodeAddr>) :-
    hb_chunk(NodeAddr, ChunkId, _);

// Chunk exists => return success and set of nodes
response(@Src, RequestId, true, NodeSet) :-
    request(@Master, RequestId, Src,
            "ChunkLocations", ChunkId),
    compute_chunk_locs(ChunkId, NodeSet);

// Chunk does not exist => return failure
response(@Src, RequestId, false, null) :-
    request(@Master, RequestId, Src,
            "ChunkLocations", ChunkId),
    notin hb_chunk(_, ChunkId, _);
```

Listing 2.7: NameNode rules to return the set of DataNodes that hold a given chunk in BOOM-FS.

about chunk locations and DataNode liveness, and the *data protocol* which clients and DataNodes use to exchange chunks. We implemented the metadata and heartbeat protocols with a set of distributed Overlog rules in a similar style. The data protocol was implemented in Java because it is simple and performance critical. We proceed to describe the three protocols in order.

For each command in the metadata protocol, there is a single rule at the client (stating that a new request tuple should be "stored" at the NameNode). There are typically two corresponding rules at the NameNode: one to specify the result tuple that should be stored at the client, and another to handle errors by returning a failure message. An example of the NameNode rules for Chunk Location requests is shown in Listing 2.7.

Requests that modify metadata follow the same basic structure, except that in addition to deducing a new result tuple at the client, the NameNode rules also deduce changes to the file system metadata relations. Concurrent requests are serialized by the Overlog runtime at the NameNode.

DataNode heartbeats have a similar request/response pattern, but are not driven by the arrival of network events. Instead, they are "clocked" by joining with the built-in *periodic* relation [120], which produces new tuples at every tick of a wall-clock timer. In addition, control protocol mes-

System	*Lines of Java*	*Lines of Overlog*
HDFS	21,700	0
BOOM-FS	1,431	469

Table 2.3: Code size of two file system implementations.

sages from the NameNode to DataNodes are deduced when certain system invariants are unmet; for example, when the number of replicas for a chunk drops below the configured replication factor.

Finally, the data protocol is a straightforward mechanism for transferring the content of a chunk between clients and DataNodes. This protocol is orchestrated by Overlog rules but implemented in Java. When an Overlog rule deduces that a chunk must be transferred from host X to Y, an output event is triggered at X. A Java event handler at X listens for these output events, and uses a simple but efficient data transfer protocol to send the chunk to host Y. To implement this protocol, we wrote a simple multi-threaded server in Java that runs on the DataNodes.

Discussion

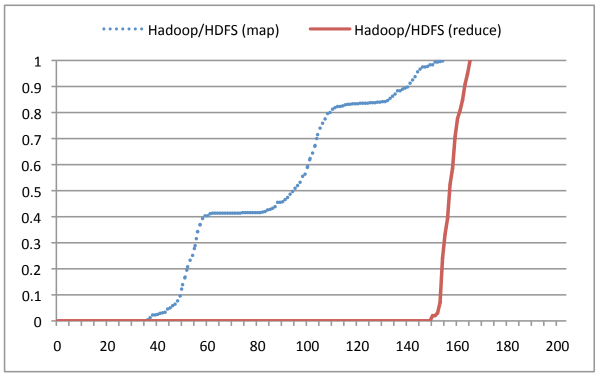
After two months of work, we had a working implementation of metadata handling strictly in Overlog, and it was straightforward to add Java code to store chunks in UNIX files. Adding the necessary Hadoop client APIs in Java took an additional week. Adding metadata durability took about a day. Table 2.3 compares statistics about the code bases of BOOM-FS and HDFS. The DataNode implementation accounts for 414 lines of the Java in BOOM-FS; the remainder is mostly devoted to system configuration, bootstrapping, and a client library. Adding support for accessing BOOM-FS to Hadoop required an additional 400 lines of Java.

BOOM-FS has performance, scaling and failure-handling properties similar to those of HDFS; again, we provide a simple performance validation in Figure 2.2. By following the HDFS architecture, BOOM-FS can tolerate DataNode failures but has a single point of failure and scalability bottleneck at the NameNode.

HDFS sidesteps many of the performance challenges of traditional file systems and databases by focusing nearly exclusively on scanning large files. It avoids most distributed systems challenges regarding replication and fault-tolerance by implementing coordination with a single centralized NameNode. As a result, most of our implementation consists of simple message handling and management of the hierarchical file system namespace.

Validation

While improved performance was not a goal of our work, we wanted to ensure that the performance of BOOM Analytics was competitive with Hadoop. To that end, we conducted a series of performance experiments using a 101-node cluster on Amazon EC2. One node executed the Hadoop JobTracker and the DFS NameNode, while the remaining 100 nodes served as slaves for running the Hadoop TaskTrackers and DFS DataNodes. The master node ran on an “high-CPU extra large” EC2 instance with 7.2 GB of memory and 8 virtual cores. Our slave nodes executed



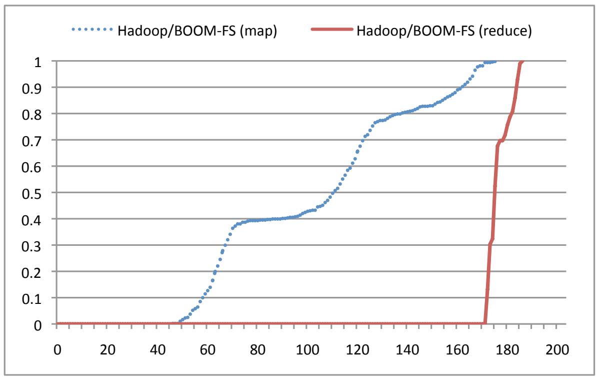


Figure 2.2: CDFs representing the elapsed time between job startup and task completion for both map and reduce tasks. In both graphs, the horizontal axis is elapsed time in seconds, and the vertical represents the fraction of tasks completed.

on "high-CPU medium" EC2 instances with 1.7 GB of memory and 2 virtual cores. Each virtual core is the equivalent of a 2007-era 2.5Ghz Intel Xeon processor.

For our experiments, we compared BOOM Analytics with Hadoop 18.1. Our workload was a wordcount job on a 30 GB file. The wordcount job consisted of 481 map tasks and 100 reduce tasks. Each of the 100 slave nodes hosted a single TaskTracker instance that can support the simultaneous execution of 2 map tasks and 2 reduce tasks.

Figure 2.2 compares the performance of BOOM-FS with HDFS. Both graphs report a cumulative distribution of map and reduce task completion times (in seconds). The map tasks complete in three distinct "waves". This is because only 2×100 map tasks can be scheduled at once. Although all 100 reduce tasks can be scheduled immediately, no reduce task can finish until all maps have been completed, because each reduce task requires the output of all map tasks.

The left graph describes the performance of Hadoop running on top of HDFS, and hence serves as a baseline. The right graphs details the performance of Hadoop MapReduce running on top of BOOM-FS. BOOM-FS performance is slightly worse than HDFS, but remains very competitive.

The Availability Rev

Having achieved a fairly faithful implementation of HDFS, we were ready to explore one of our main motivating hypotheses: that data-centric programming would make it easy to add complex distributed functionality to an existing system. We chose an ambitious goal: retrofitting BOOM-FS with high availability failover via "hot standby" NameNodes. A proposal for warm standby was posted to the Hadoop issue tracker in October of 2008 ([109] issue HADOOP-4539). When we started this work in 2009, we felt that a hot standby scheme would be more useful, and we deliberately wanted to pick a more challenging design to see how hard it would be to build in Overlog. A warm standby solution for a high-availability NameNode was incorporated into HDFS in 2012 [5]; a hot standby approach based on consensus was proposed in 2014 by a third party [3].

Paxos-replicated BOOM-FS

Correctly implementing efficient hot standby replication is tricky, since replica state must remain consistent in the face of node failures and lost messages. One solution to this problem is to implement a globally-consistent distributed log, which guarantees a total ordering over events affecting replicated state. The Paxos algorithm is the canonical mechanism for this feature [106]. Once we had Paxos implemented (Section 2.2), it was straightforward to support the replication of the distributed file system metadata. All state-altering actions are represented in the revised BOOM-FS as Paxos decrees, which are passed into the Paxos logic via a single Overlog rule that intercepts tentative actions and places them into a table that is joined with Paxos rules. Each action is considered complete at a given site when it is "read back" from the Paxos log, i.e. when it becomes visible in a join with a table representing the local copy of that log. A sequence number field in the Paxos log table captures the globally-accepted order of actions on all replicas.

We also had to ensure that Paxos state was durable in the face of crashes. The support for persistent tables in Overlog made this straightforward: Lamport's description of Paxos explicitly distinguishes between transient and durable state. Our implementation already divided this state into separate relations, so we simply marked the appropriate relations as durable.

We validated the performance of our implementation experimentally: in the absence of failure replication has negligible performance impact, but when the primary NameNode fails, a backup NameNode takes over reasonably quickly.

The Availability revision was our first foray into serious distributed systems programming, and we continued to benefit from the high-level abstractions provided by Overlog. Most of our attention was focused at the appropriate level of complexity: faithfully capturing the reasoning involved in distributed protocols.

Validation

After adding support for high availability to the BOOM-FS NameNode (Section 2.3), we wanted to evaluate two properties of our implementation. At a fine-grained level, we wanted to ensure that our complete Paxos implementation was operating according to the specifications in the literature. This required logging and analyzing network messages sent during the Paxos protocol. This was a natural fit for the metaprogrammed tracing tools we discussed in Section 2.3. We created unit tests to trace the message complexity of our Paxos code, both at steady state and under churn. When the message complexity of our implementation matched the specification, we had more confidence in the correctness of our code.

Second, we wanted to see the availability feature "in action", and to get a sense of how our implementation would perform in the face of master failures. Specifically, we evaluated the impact of the consensus protocol on BOOM Analytics system performance, and the effect of failures on overall completion time. We ran a Hadoop wordcount job on a 5GB input file with a cluster of 20 EC2 nodes, varying the number of master nodes and the failure condition. These results are summarized in Table 2.4. We then used the same workload to perform a set of simple fault-injection experiments to measure the effect of primary master failures on job completion rates at a

Number of NameNodes	Failure Condition	Avg. Completion Time (secs)	Standard Deviation
1	None	101.89	12.12
3	None	102.70	9.53
3	Backup	100.10	9.94
3	Primary	148.47	13.94

Table 2.4: Job completion times with a single NameNode, 3 Paxos-enabled NameNodes, backup NameNode failure, and primary NameNode failure.

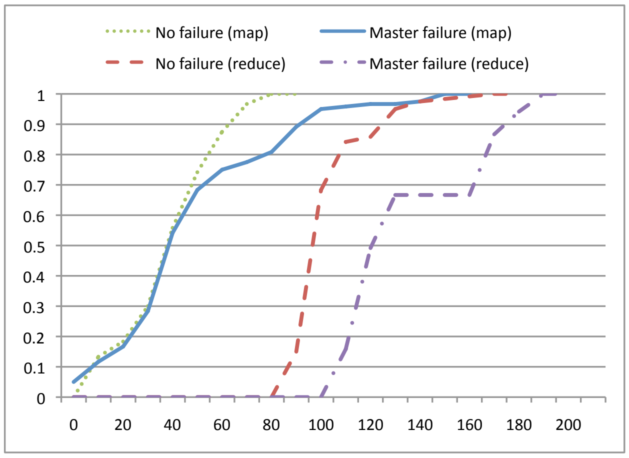


Figure 2.3: CDF of completed tasks over time (secs), with and without primary master failure.

finer grain, observing the progress of the map and reduce jobs involved in the wordcount program. Figure 2.3 shows the cumulative distribution of the percentage of completed map and reduce jobs over time, in normal operation and with a failure of the primary NameNode during the map phase. Note that Map tasks read from HDFS and write to local files, whereas Reduce tasks read from Mappers and write to HDFS. This explains why the CDF for Reduce tasks under failure goes so crisply flat in Figure 2.3: while failover is underway after an HDFS NameNode failure, some nearly-finished Map tasks may be able to complete, but no Reduce task can complete.

The Scalability Rev

HDFS NameNodes manage large amounts of file system metadata, which is kept in memory to ensure good performance. The original GFS paper acknowledged that this could cause significant memory pressure [69], and NameNode scaling was often an issue in practice at Yahoo! citeyahoo-hdfs. Given the data-centric nature of BOOM-FS, we hoped to very simply scale out the NameNode across multiple *NameNode-partitions*. From a database design perspective this seemed trivial — it involved adding a "partition" column to the Overlog tables containing NameNode state. The resulting code composes cleanly with our availability implementation: each NameNode-partition can be a single node, or a Paxos group.

There are many options for partitioning the files in a directory tree. We opted for a simple strategy based on the hash of the fully qualified pathname of each file. We also modified the client library to broadcast requests for directory listings and directory creation to each NameNode-partition. Although the resulting directory creation implementation is not atomic, it is idempotent; recreating a partially-created directory will restore the system to a consistent state, and will preserve any files in the partially-created directory.

For all other BOOM-FS operations, clients have enough information to determine the correct NameNode-partition. We do not support atomic "move" or "rename" across partitions. This feature is not exercised by Hadoop, and complicates distributed file system implementations considerably. In our case, it would involve the atomic transfer of state between otherwise-independent Paxos instances. We believe this would be relatively clean to implement — we have a two-phase commit protocol implemented in Overlog — but decided not to pursue this feature at present.

Discussion

By isolating the file system state into relations, it became a textbook exercise to partition that state across nodes. It took 8 hours of developer time to implement NameNode partitioning; two of these hours were spent adding partitioning and broadcast support to the Overlog code. This was a clear win for the data-centric approach.

The simplicity of file system scale-out made it easy to think through its integration with Paxos, a combination that might otherwise seem very complex. Our confidence in being able to compose techniques from the literature is a function of the compactness and resulting clarity of our code.

It is worth noting that the issue of scalability of NameNode state was addressed in 2011 when HDFS Federation [4] was released in 2011 (two years after the work reported here). Federation support multiple NameNode namespaces, as oppposed to the transparent partitioning in BOOM-FS.

The Monitoring Rev

As our BOOM Analytics prototype matured and we began to refine it, we started to suffer from a lack of performance monitoring and debugging tools. Singh et al. pointed out that Overlog is well-suited to writing distributed monitoring queries, and offers a naturally introspective approach: simple Overlog queries can monitor complex protocols [154]. Following that idea, we decided to develop a suite of debugging and monitoring tools for our own use.

Invariants

One advantage of a logic-oriented language like Overlog is that it encourages the specification of system invariants, including "watchdogs" that provide runtime checks of behavior induced by the program. For example, one can confirm that the number of messages sent by a protocol like Paxos matches the specification. Distributed Overlog rules induce asynchrony across nodes; such rules are only *attempts* to achieve invariants. An Overlog program needs to be enhanced with global coordination mechanisms like two-phase commit or distributed snapshots to convert distributed

Overlog rules into global invariants [42]. Singh et al. have shown how to implement Chandy-Lamport distributed snapshots in Overlog [154]; we did not go that far in our own implementation.

To simplify debugging, we wanted a mechanism to integrate Overlog invariant checks into Java exception handling. To this end, we added a relation called *die* to JOL; when tuples are inserted into the *die* relation, a Java event listener is triggered that throws an exception. This feature makes it easy to link invariant assertions in Overlog to Java exceptions: one writes an Overlog rule with an invariant check in the body, and the *die* relation in the head.

We made extensive use of these local-node invariants in our code and unit tests. Although these invariant rules increase the size of a program, they improve readability in addition to reliability. Assertions that we specified early in the implementation of Paxos aided our confidence in its correctness as we added features and optimizations.

Monitoring via Metaprogramming

Our initial prototypes of both BOOM-MR and BOOM-FS had significant performance problems. Unfortunately, Java-level performance tools were of little help. A poorly-tuned Overlog program spends most of its time in the same routines as a well-tuned Overlog program: in dataflow operators like Join and Aggregation. Java-level profiling lacks the semantics to determine which rules are causing the lion's share of the dataflow code invocations.

Fortunately, it is easy to do this kind of bookkeeping directly in Overlog. In the simplest approach, one can replicate the body of each rule in an Overlog program and send its output to a log table (which can be either local or remote). For example, the Paxos rule that tests whether a particular round of voting has reached quorum:

```
quorum(@Master, Round) :-
    priestCnt(@Master, Pcnt),
    lastPromiseCnt(@Master, Round, Vcnt),
    Vcnt > (Pcnt / 2);
```

might have an associated tracing rule:

```
trace_r1(@Master, Round, RuleHead, Tstamp) :-
    priestCnt(@Master, Pcnt),
    lastPromiseCnt(@Master, Round, Vcnt),
    Vcnt > (Pcnt / 2),
    RuleHead = "quorum",
    Tstamp = System.currentTimeMillis();
```

This approach captures per-rule dataflow in a trace relation that can be queried later. Finer levels of detail can be achieved by "tapping" each of the predicates in the rule body separately in a similar fashion. The resulting program passes no more than twice as much data through the system, with one copy of the data being "teed off" for tracing along the way. When profiling, this overhead is often acceptable. However, writing the trace rules by hand is tedious.

Using the metaprogramming approach of Evita Raced [50], we were able to automate this task via a *trace rewriting* program written in Overlog, involving the meta-tables of rules and terms. The trace rewriting expresses logically that for selected rules of some program, new rules should be added to the program containing the body terms of the original rule, and auto-generated head terms. Network traces fall out of this approach naturally: any dataflow transition that results in network communication is flagged in the generated head predicate during trace rewriting.

Using this idea, it took less than a day to create a general-purpose Overlog code coverage tool that traced the execution of our unit tests and reported statistics on the "firings" of rules in the JOL runtime, and the counts of tuples deduced into tables. Our metaprogram for code coverage and network tracing consists of 5 Overlog rules that are evaluated by every participating node, and 12 summary rules that are run at a centralized location. Several hundred lines of Java implement a rudimentary front end to the tool. We ran our regression tests through this tool, and immediately found both "dead code" rules in our programs, and code that we knew needed to be exercised by the tests but was as-yet uncovered.

2.4 Related Work

Related Work

Until recently, declarative and data-centric languages were considered useful only in narrow domains such as SQL databases and spreadsheets [108], but in the last decade this view has changed substantially. MapReduce [57] and its inheritors [173, 110] have popularized functional dataflow programming with new audiences in computing. And a surprising breadth of research projects have proposed and prototyped declarative languages in recent years, including overlay networks [120], three-tier web services [170], natural language processing [60], modular robotics [26], video games [167], file system metadata analysis [80], and compiler analysis [103].

Most of the languages cited above are declarative in the same sense as SQL: they are based in first-order logic. Some — notably MapReduce, but also SGL [167] — are algebraic or dataflow languages, used to describe the composition of operators that produce and consume sets or streams of data. Although arguably imperative, they are far closer to logic languages than to traditional imperative languages like Java or C, and are often amenable to set-oriented optimization techniques developed for declarative languages [167]. Declarative and dataflow languages can also share the same runtime, as demonstrated by recent integrations of MapReduce and SQL in Hive [161], DryadLINQ [172], HadoopDB [10], and products from vendors such as Greenplum and Aster.

Concurrent with our work, the Erlang language was used to implement a simple MapReduce framework called Disco [139], and a transactional DHT called Scalaris with Paxos support [149]. Philosophically, Erlang revolves around programming *concurrent processes*, rather than data. In Section 2.5 we reflect on some benefits of a data-centric language over Erlang's approach. For the moment, Disco is significantly less functional than BOOM Analytics, lacking a distributed file system, multiple scheduling policies, and high availability via consensus.

Distributed state machines are the traditional formal model for distributed system implementations, and can be expressed in languages like Input/Output Automata (IOA) and the Temporal Logic of Actions (TLA) [123]. By contrast, our approach is grounded in Datalog and its extensions. The pros and cons of starting with a database foundation are a recurring theme of this paper.

Our use of metaprogrammed Overlog was heavily influenced by the Evita Raced Overlog metacompiler [50], and the security and typechecking features of Logic Blox' LBTrust [128]. Some of our monitoring tools were inspired by Singh et al. [154], although our metaprogrammed implementation is much simpler and more elegant than that of P2.

2.5 Reflections

Our experience writing protocols and applications in Overlog was mostly positive. Using tables as a uniform data representation dramatically simplified the problem of state management. In the protocols and systems we built, *everything* is data, represented as tuples in tables. This includes traditional persistent information like file system metadata, runtime state like TaskTracker status, summary statistics like those used by the JobTracker's scheduling policy, in-flight messages, system events, execution state of the system, and even parsed code. With a few significant exceptions, it was also natural to express these systems and protocols with high-level declarative *queries*, describing continuous transformations over that state.

As we hypothesized, recasting the problem of manipulating distributed state into the problem of parallel data management simplified distributed programming. Features that should have been difficult were trivial to implement once the metadata was represented as relations accessible via queries. The benefits of this approach are perhaps best illustrated by the extreme simplicity with which we scaled out the NameNode via partitioning (Section 2.3): by having the relevant state stored as data, we were able to use standard data partitioning to achieve what would ordinarily be a significant rearchitecting of the system. Similarly, the ease with which we implemented system monitoring — via both system introspection tables and rule rewriting — arose because we could easily write rules that manipulated concepts as diverse as transient system state and program semantics (Section 2.3).

The uniformity of data-centric interfaces also enabled *interposition* [90] of components in a natural manner: the dataflow "pipe" between two system modules can be easily rerouted to go through a third module. Because dataflows can be routed across the network (via the location specifier in a rule's head), interposition can also involve distributed logic — this is how we easily added Paxos support into the BOOM-FS NameNode (Section 2.3).

The last data-centric programming benefit we observed related to the timestepped dataflow execution model, which we found to be simpler than traditional notions of concurrent programming. Traditional models for concurrency include event loops and multithreaded programming. Our concern regarding event loops — and the state machine programming models that often accompany them — is that one needs to reason about *combinations* of states and events. That would seem to put a quadratic reasoning task on the programmer. In principle our logic programming deals with the same issue, but we found that each composition of two tables (or tuple-streams) could be

```
counter(0).
counter(X+1) :- counter(X), request(_, _).
response(@From, X) :- counter(X), request(To, From).
```

Listing 2.8: An Overlog implementation of a counter.

thought through in isolation, much as one thinks about composing relational operators or piping Map and Reduce tasks. Given our prior experience writing multi-threaded code with locking, we were happy that the simple timestep model of Overlog obviated the need for this entirely — there is no explicit synchronization logic in any of the BOOM-FS code, and we view this as a clear victory for the programming model.

Despite these successes at a high level, Overlog's ambiguous temporal semantics were a stumbling block. Overlog was designed for routing protocols, which are *best-effort* systems with *soft state*. A simple language that exposed a single data type (the relation) and a single operator (logical implication) was sufficient to express protocols that accumulate more information over time. However, using implication ("given what I know, what else follows") to express both the *accumulation* of information and *state change* was problematic [137, 126]. Consider the following example, ubiquitous in distributed systems: a counter that is incremented on message receipt. When a *request* message is received, we increment the counter, and send a reply containing the (pre-increment) value of the counter.

An attempt to implement such a counter in Overlog is shown in Listing 2.8. There are a number of possible interpretations of this code. If there is a *request*, there should be a response message whose payload is:

1. the value in *counter* before the update.
2. the value in *counter* after the update.
3. all of the natural numbers.

In practice, a seasoned Overlog programmer can rule out #3 by placing a key constraint on *counter* to ensure that it only ever contains one row—this constraint implements a hidden "last writer wins" semantics, simulating "updates." Unfortunately, this approach blends declarative and imperative semantics, establishing a hidden ordering dependency in an ostensibly order-free language. Moreover, key constraints do not remove the ambiguity from the example above; it is still not clear whether the response message will contain 0 or 1.

Another weakness of Overlog is that its semantics does not model asynchronous communication. The last line of the example above states that if there is a request and a counter, then there will be a response *at* the location indicated by From, containing the counter payload. This is not a true implication, because its premises (RHS) may hold without its conclusion holding—a message is never received *immediately*, and indeed may never be received at all. Critically absent from Overlog is the ability to characterize *uncertainty* about when (and indeed whether) the conclusions of such an implication will hold.

In the next chapter, we will present DEDALUS, a Datalog-based distributed programming language that addresses these issues by incorporating an explicit representation of *time*.

Chapter 3

Disorderly languages

Time is a device that was invented to keep everything from happening at once.
– Graffiti on a wall at Cambridge University [9].

In this chapter, we present the disorderly programming languages Dedalus and Bloom. Dedalus, described in Section 3.1, provides a formal semantics that captures the salient details of distributed programming: namely, asynchronous communication and partial failure. Like Overlog, Dedalus's terse syntax is based on Datalog. Section 3.3 presents Bloom, which builds on Dedalus to provide a programmer-friendly syntax, a module system to enable encapsulation and reuse in software projects, and a collection of built-in analyses and tools to aid the programmer in the software development process.

Despite their differences in syntax and structuring conventions, Dedalus and Bloom share the same model-theoretic semantics (presented in Section 3.1), which capture the fundamental uncertainty in distributed *executions* and the relationship between that uncertainty and program *outcomes*. As we will see, this correspondence will provide a formal basis for confluence analysis (Section 3.2), coordination placement (Section 4.4) and lineage-driven fault injection (Chapter 5).

3.1 Dedalus

Despite its expressive limitations, Overlog made it easy to implement substantial systems functionality. To address its shortcomings, we designed the Dedalus language, whose syntax and semantics we present below.

In designing Dedalus, we sought to preserve the "disorderly" feel of Overlog—in which systems are implemented as continuous queries—as much as possible. Like Overlog, Dedalus has a concise syntax based on Datalog. It extends this core with a small set of *temporal* operators that resolve the semantic ambiguities discussed in the last section. Like Overlog, Dedalus blurs the distinction between various representations of program state and varieties of control flow in order to focus the programmer's attention on the transformation of data. Unlike Overlog, Dedalus also forces the programmer to reason explicitly about the role of time in state change and communication.

Instead of introducing an operational semantics to accomodate state change [45, 58] and asynchronous communication [76], Dedalus has a purely *declarative* semantics based on the reification of logical time into data. As we shall see, the "meaning" of a Dedalus program is precisely its set of *ultimate models*, or the facts (or equivalently, database records) that are *eventually always true*. The notion of ultimate models provides a useful finite characterization of infinite executions as well as a means of comparing executions that differ only superficially (e.g. only in the values of timestamps). Programs with a *unique* ultimate model have the valuable property of being *deterministic* despite pervasive nondeterminism in their distributed executions, a property that will become the basis for *confluence analysis*, described in Section 3.3. Dedalus provides a formal basis for studying exactly which programs have unique ultimate models.

We begin by defining Dedalus$_0$, a restricted sublanguage of Datalog (Section 3.1). In Section 3.1, we show how Dedalus$_0$ supports a variety of stateful programming patterns. Finally, we introduce Dedalus by adding support for asynchrony to Dedalus$_0$ in Section 3.1. Throughout this chapter, we demonstrate the expressivity and practical utility of our work with specific examples, including a number of building-block routines from classical distributed computing, such as sequences, queues, distributed clocks, and reliable broadcast. We also discuss the correspondence between Dedalus and our prior work implementing full-featured distributed services in Overlog.

Dedalus$_0$

We take as our foundation the language Datalog¬ [165]: Datalog enhanced with negated subgoals. We will be interested in the classes of syntactically stratifiable and locally stratifiable programs [141], which we revisit below. For conciseness, when we refer to "Datalog" below our intent is to admit negation—i.e., Datalog¬.

As a matter of notation, we refer to a countably infinite universe of constants C—in which $C_1, C_2, \ldots$ are representations of individual constants—and a countably infinite universe of variable symbols $\mathcal{A} = A_1, A_2, \ldots$. We will capture time in Dedalus$_0$ via an infinite relation `successor` isomorphic to the successor relation on the integers; for convenience we will in fact refer to the domain $\mathbb{N}$ when discussing time, though no specific interpretation of the symbols in $\mathbb{N}$ is intended beyond the fact that `successor(x,y)` is true exactly when $y = x + 1$.

Syntactic Restrictions

Dedalus$_0$ is a restricted sublanguage of Datalog. Specifically, we restrict the admissible schemata and the form of rules with the four constraints that follow.

Schema: We require that the final attribute of every Dedalus$_0$ predicate range over the domain $\mathbb{Z}$. In a typical interpretation, Dedalus$_0$ programs will use this final attribute to connote a "timestamp," so we refer to this attribute as the *time suffix* of the corresponding predicate.

Time Suffix: In a well-formed Dedalus$_0$ rule, every subgoal must use the same variable $\mathcal{T}$ as its time suffix. A well-formed Dedalus$_0$ rule must also have a time suffix $\mathcal{S}$ in its head predicate, which must be constrained in the rule body in exactly one of the following two ways:

1. The rule is *deductive* if $\mathcal{S}$ is bound to the value $\mathcal{T}$; that is, the body contains the subgoal $\mathcal{S}$ = $\mathcal{T}$.

2. The rule is *inductive* if $\mathcal{S}$ is the successor of $\mathcal{T}$; that is, the body contains the subgoal successor($\mathcal{T}$, $\mathcal{S}$).

In Section 3.1, we will define DEDALUS as a superset of DEDALUS$_0$ and introduce a third kind of rule to capture asynchrony.

Example 1 The following are examples of well-formed deductive and inductive rules, respectively.

Deductive:

```
p(A, B, S) ← e(A, B, 𝒯), 𝒮 = 𝒯;
```

Inductive:

```
q(A, B, S) ← e(A, B, 𝒯), successor(𝒯, 𝒮);
```

Abbreviated Syntax and Temporal Interpretation

We have been careful to define DEDALUS$_0$ as a subset of Datalog; this inclusion allows us to take advantage of Datalog's well-known semantics and the rich literature on the language.

DEDALUS$_0$ programs are intended to capture temporal semantics. For example, a fact p($C_1 \ldots C_n$, C_{n+1}), with some constant C_{n+1} in its time suffix can be thought of as a fact that is true "at time C_{n+1}." Deductive rules can be seen as *instantaneous* statements: their deductions hold for predicates agreeing in the time suffix and describe what is true "for an instant" given what is known at that instant. Inductive rules are *temporal*—their consequents are defined to be true "at a different time" than their antecedents.

To simplify DEDALUS$_0$ notation for this typical interpretation, we introduce some syntactic "sugar" as follows:

- *Implicit time-suffixes in body predicates:* Since each body predicate of a well-formed rule has an existential variable $\mathcal{T}$ in its time suffix, we optionally omit the time suffix from each body predicate and treat it as implicit.

- *Temporal head annotation:* Since the time suffix in a head predicate may be either equal to $\mathcal{T}$ or equal to $\mathcal{T}$'s successor, we omit the time suffix from the head—and its relevant constraints from the body—and instead attach an identifier to the head predicate of each temporal rule, to indicate the change in time suffix. A temporal head predicate is of the form:

  r(A_1,A_2,[...],A_n)@next

  The identifier @next stands in for successor($\mathcal{T}$,S) in the body.

- *Timestamped facts:* For notational consistency, we write the time suffix of facts (which must be given as a constant) outside the predicate. For example:

  `r(`A_1`,`A_2`,[...],`A_n`)@`C

  Example 2 The following are "sugared" versions of deductive and inductive rules from Example 1, and a temporal fact:

  Deductive:

  ```
  p(A, B) ← e(A, B);
  ```

  Inductive:

  ```
  q(A, B)@next ← e(A, B);
  ```

  Fact:

  ```
  e(1, 2)@10;
  ```

State in Logic

Given our definition of DEDALUS$_0$, we now address the persistence and mutability of data across time—one of the two signature features of distributed systems for which we provide a model-theoretic foundation.

The intuition behind DEDALUS$_0$'s `successor` relation is that it models the passage of (logical) time. In our discussion, we will say that ground atoms with lower time suffixes occur "before" atoms with higher ones. The constraints we imposed on DEDALUS$_0$ rules restrict how deductions may be made with respect to time. First, rules may only refer to a single time suffix variable in their body, and hence subgoals *cannot join across different "timesteps."* Second, rules may specify deductions that occur concurrently with their ground facts or in the next timestep—in DEDALUS$_0$, we rule out induction "backwards" in time or "skipping" into the future.

This notion of time allows us to consider the contents of the EDB—as well as models of the IDB—with respect to an "instant in time". Intuitively, we can imagine the successor relation gradually being revealed over time: this produces a sequence of models (one per timestep), providing an unambiguous way to declaratively express persistence and state changes across time. In this section, we give examples of language constructs that capture state-oriented motifs such as persistent relations, deletion and update, sequences, and queues. Later we will consider the model-theoretic formal semantics of DEDALUS.

Simple Persistence

A fact in predicate p at time $\mathcal{T}$ may provide ground for deductive rules at time $\mathcal{T}$, as well as ground for deductive rules in timesteps greater than $\mathcal{T}$, provided there exists a *simple persistence rule* of the form:

```
p(A_1,A_2,...,A_n)@next ← p(A_1,A_2,...,A_n);
```

A simple persistence rule of this form ensures that a p fact true at time i will be true $\forall j \in \mathbb{Z} : j \geq i$.

Mutable State

To model deletions and updates of a fact, it is necessary to break the induction in a simple persistence rule. Adding a $\mathtt{p}_{neg}$ subgoal to the body of a simple persistence rule accomplishes this:

```
p(A_1,A_2,...,A_n)@next ←, p(A_1,A_2,[...],A_n),¬ p_neg(A_1,A_2,...,A_n);
```

If, at any time k, we have a fact $\mathtt{p}_{neg}(C_1,C_2,[...],C_n)\mathtt{@k}$, then we do not deduce a $\mathtt{p}(C_1,C_2,[...],C_n)\mathtt{@k+1}$ fact. Furthermore, we do not deduce a $\mathtt{p}(C_1,C_2,[...],C_n)\mathtt{@j}$ fact for any $j > k$, unless this `p` fact is re-derived at some timestep $j > k$ by another rule. That is, a persistent fact, once stated, remains true until it is retracted; once retracted, a fact remains false until it is re-asserted.

Example 3 Consider the following DEDALUS$_0$ program and ground facts:

```
p(A, B)@next ← p(A, B), ¬p_neg(A, B);

p(1,2)@101;
p(1,3)@102;
p_neg(1,2)@300;
```

The following facts are true: `p(1,2)@200`, `p(1,3)@200`, `p(1,3)@300`. However, `p(1,2)@301` is false because `p(1,2)` was "deleted" at timestep `300`.

Since mutable persistence occurs frequently in practice, we provide the *persist* macro, which takes three arguments: a predicate name, the name of another predicate to hold "deleted" facts, and the (matching) arity of the two predicates. The macro expands to the corresponding mutable persistence rule. For example, the above `p` persistence rule may be equivalently specified as `persist[p]`.

Sequences

One may represent a database sequence—an object that retains and monotonically increases a counter value—with a pair of inductive rules. One rule increments the current counter value when some condition is true, while the other persists the value of the sequence when the condition is false. We can capture the increase of the sequence value without using arithmetic, because the infinite series of `successor` has the monotonicity property we require:

```
seq(B)@next ← seq(A), successor(A,B), event(_);
seq(A)@next ← seq(A), ¬event(_);
seq(0);
```

Note that these three rules produce only a single value of `seq` at each timestep, but they do so in a manner slightly different than our standard persistence template.

Queues

While sequences are useful for assigning integers to (and hence imposing a total ordering on) tuples, programs will in some cases require that tuples are processed in a particular (partial) order—that is, they require that tuples are assigned timestamps in an order consistent with their contents. To this end, we introduce a queue template, which employs inductive persistence and aggregate functions in rule heads to "reveal" tuples according to a data-dependent order, rather than as a set.

Aggregate functions simplify our discussion of queues. Mumick and Shmueli observe correspondences in the expressivity of Datalog with stratified negation and stratified aggregation functions [133]. Adding aggregation to our language does not affect its expressive power, but is useful for writing natural constructs for distributed computing including queues and ordering.

In Dedalus$_0$ we allow aggregate functions to appear in the heads of deductive rules in the form: $\mathrm{p}(A_1,\ldots,A_n,\rho_1(A_{n+1}),\ldots,\rho_m(A_{n+m}))$

In such a rule (whose body must bind $A_1,\ldots,A_{n+m}$), the predicate p contains one row for each satisfying assignment of $A_1,\ldots,A_n$—akin to the distinct "groups" of SQL's "GROUP BY" notation.

Consider a predicate priority_queue that represents a series of tasks to be performed in some predefined order. Its attributes are a string representing a user, a job, and an integer indicating the priority of the job in the queue:

```
priority_queue('bob', 'bash', 200)@123;
priority_queue('eve', 'ls', 1)@123;
priority_queue('alice', 'ssh', 204)@123;
priority_queue('bob', 'ssh', 205)@123;
```

Observe that all the time suffixes are the same. Given this schema, we note that a program would likely want to process priority_queue events individually in a data-dependent order, in spite of their coincidence in logical time.

In the program below, we define a table m_priority_queue that serves as a queue to feed priority_queue. The queue must persist across timesteps because it may take multiple timesteps to drain it. At each timestep, for each value of **A**, a single tuple is projected into priority_queue and deleted (atomic with the projection) from m_priority_queue, changing the value of the aggregate calculated at the subsequent step:

```
persist[m_priority_queue]

omin(A, min<C>) ← // identify the lowest priority value
  m_priority_queue(A, _, C);

priority_queue(A, B, C)@next ← // dequeue tuple with lowest priority
  m_priority_queue(A, B, C),
  omin(A, C);

m_priority_queue_neg(A, B, C) ← // ... and concurrently remove it
  m_priority_queue(A, B, C),
  omin(A, C);
```

Under such a queueing discipline, deductive rules that depend on priority_queue are constrained to consider only min-priority tuples at each timestep per value of the variable **A**, thus implementing a per-user FIFO discipline. To enforce a global FIFO ordering over `priority_queue`, we may redefine `omin` and any dependent rules to exclude the **A** attribute.

A queue establishes a mapping between DEDALUS$_0$'s timesteps and the priority-ordering attribute of the input relation. By doing so, we take advantage of the monotonic property of timestamps to enforce an ordering property over our input that is otherwise difficult to express in a logic language. We return to this idea in our discussion of temporal "entanglement" in Section 3.1.

Asynchrony

In this section we introduce DEDALUS, a superset of DEDALUS$_0$ that also admits the *choice* construct [77] to bind time suffixes. Choice allows us to model the inherent non-determinism in communication over unreliable networks that may delay, lose or reorder the results of logical deductions. We then describe a syntactic convention for rules that model communication between agents, and show how DEDALUS can be used to implement common distributed computing idioms like Lamport clocks and reliable broadcast.

Choice

An important property of distributed systems is that individual computers cannot control or observe the temporal interleaving of their computations with other computers. One aspect of this uncertainty is captured in network delays: the arrival "time" of messages cannot be directly controlled by either sender or receiver. In this section, we enhance our language with a traditional model of non-determinism from the literature to capture these issues: the *choice* construct as defined by Greco and Zaniolo [77].

The subgoal `choose(`(X_1)`, `(X_2)`)` may appear in the body of a rule, where X_1 and X_2 are vectors whose constituent variables occur elsewhere in the body. Such a subgoal enforces the functional dependency $X_1 \rightarrow X_2$, "choosing" a single assignment of values to the variables in X_2 for each variable in X_1.

The choice construct is meant to capture non-determinism in program executions. In a model-theoretic interpretation of logic programming, an unstratifiable program must have a multiplicity of *stable models*: one for each possible choice of variable assignments. Greco and Zaniolo define choice in precisely this fashion: the choice construct is expanded into an unstratifiable strongly-connected component of rules, and each possible choice is associated with a different model. Each such model has a unique assignment that respects the given functional dependencies. In our discussion, it may be helpful to think of one such model chosen non-deterministically—a non-deterministic "assignment of timestamps to tuples."

Distribution Model

The choice construct captures the non-determinism of communicating agents in a distributed system: we use it in a stylized way to model typical notions of distribution. To this end, DEDALUS adopts the "horizontal partitioning" convention introduced by Loo et al. [121]. To represent a distributed system, we consider some number of agents, each running a copy of the same program against a disjoint subset (*horizontal partition*) of each predicate's contents. We require one attribute in each predicate to be used to identify the partitioning for tuples in that predicate. We call such an attribute a *location specifier*, and prefix it with a # symbol in Dedalus.

Finally, we constrain Dedalus rules in such a way that the location specifier variable in each body predicate is the same—i.e., the body contains tuples from exactly one partition of the database, logically colocated (on a single "machine"). If the head of the rule has the same location specifier variable as the body, we call the rule "local," since its results can remain on the machine where they are computed. If the head has a different variable in its location specifier, we call the rule a *communication rule*. We now proceed to our model of the asynchrony of this communication, which is captured in a syntactic constraint on the heads of communication rules.

Asynchronous Rules

In order to represent the non-determinism introduced by distribution, we admit a third type of rule, called an *asynchronous* rule. A rule is asynchronous if the relationship between the head time suffix $\mathcal{S}$ and the body time suffix $\mathcal{T}$ is unknown. Furthermore, $\mathcal{S}$ (but not $\mathcal{T}$) may take on the special value $\top$ which means "never." Derivation at $\top$ indicates that the deduction is "lost," as time suffixes in rule bodies do not range over $\top$.

We model network non-determinism using the choice construct to choose from a value in the special `time` predicate, which is defined using the following Datalog rules:

```
time(⊤);
time(S) ← successor(S, _);
```

Each asynchronous rule with head predicate $\texttt{p}(A_1,\ldots,A_n)$ has the following additional subgoals in its body:

$\texttt{time}(\mathcal{S})\texttt{, choose((}A_1,\ldots,A_n,\mathcal{T}\texttt{), (}\mathcal{S}\texttt{)),}$

where $\mathcal{S}$ is the timestamp of the rule head. Note that our use of `choose` incorporates all variables of each head predicate tuple, which allows a unique choice of $\mathcal{S}$ for each head tuple. We further require that communication rules include the location specifier appearing in the rule body among the functionally-determining attributes of the `choose` predicate, even if it does not occur in the head.

Example 4 A well-formed asynchronous Dedalus rule:

```
r(A, B, 𝒮) ←
  e(A, B, 𝒯),
  time(𝒮),
  choose((A, B, 𝒯), (𝒮));
```

We admit a new temporal head annotation to sugar the rule above. The identifier `async` implies that the rule is asynchronous, and stands in for the additional body predicates. The example above expressed using `async` is:

Example 5 A sugared asynchronous Dedalus rule:

	DEDALUS code	Sugared DEDALUS code
deductive	`p(A, B, S) ← e(A, B, T), S = T;`	`p(A, B) ← e(A, B);`
inductive	`q(A, B, S) ← e(A, B, T), successor(T, S);`	`q(A, B)@next ← e(A, B);`
async	`r(A, B, S) ← e(A, B, T), time(S), choose((A,B,T), (S));`	`r(A, B)@async ← e(A, B);`
fact	`e(1, 2, 3)`	`e(1, 2)@3`

Table 3.1: Well-formed deductive, inductive, and asynchronous rules and a fact, and sugared versions.

```
r(A, B)@async ← e(A, B);
```

The complete DEDALUS syntax, with and without syntactic sugar, is shown in Table 3.1.

Asynchrony and Distribution in DEDALUS

As noted above, communication rules must be asynchronous. This restricts our model of communication between agents in two important ways. First, by restricting all rule bodies to a single agent, the only communication modeled in DEDALUS occurs via communication rules. Second, because all communication rules are asynchronous, agents may only learn about time values at another agent by receiving messages (with unbounded delay) from that agent. Note that this model says nothing about the relationship between the agents' clocks.

Temporal Monotonicity

Nothing in our definition of asynchronous rules prevents tuples in the head of a rule from having a timestamp that precedes the timestamp in the rule's body. This is a significant departure from DEDALUS$_0$, since it violates the monotonicity assumptions upon which we based our proof of temporal stratification. On an intuitive level, it may also trouble us that rules can derive head tuples that exist "before" the body tuples on which they are grounded; this situation violates intuitive notions of causality and admits the possibility of temporal paradoxes.

We have avoided restricting DEDALUS to rule out such issues, as doing so would reduce its expressiveness. Recall that simple monotonic Datalog (without negation) is insensitive to the values in any particular attribute. As we will discuss in detail in Section 3.2, DEDALUS programs without negation are also well-defined regardless of any "temporal ordering" of deductions: in monotonic programs, even if tuples with timestamps "in the future" are used to derive tuples "from the past," there is an unambiguous unique minimal model.

Practical Implications Given this discussion, in practice we are interested in three asynchronous scenarios: (a) monotonic programs (even with non-monotonicity in time), (b) non-monotonic programs whose semantics guarantee monotonicity of time suffixes and (c) non-monotonic programs where we have domain knowledge guaranteeing monotonicity of time suffixes. Each represents practical scenarios of interest.

The first category captures the spirit of many simple distributed implementations that are built atop unreliable asynchronous substrates. For example, in some Internet publishing applications (weblogs, online fora), it is possible due to caching or failure that a "thread" of discussion arrives out of order, with responses appearing before the comments they reference. In many cases a monotonic "bag semantics" for the comment program is considered a reasonable interface for readers, and the ability to tolerate temporal anomalies simplifies the challenge of scaling a system through distribution.

The second scenario is achieved in Dedalus$_0$ via the use of `successor` for the time suffix. The asynchronous rules of Dedalus require additional program logic to guarantee monotonic increases in time for predicates with dependencies. In the literature of distributed computing, this constraint is known as a *causal ordering* and is enforced by distributed clock protocols. We review one classic protocol in the Dedalus context in Section 3.1; including this protocol into Dedalus programs ensures temporal monotonicity.

Finally, certain computational substrates guarantee monotonicity in both timestamps and message ordering—for example, some multiprocessor cache coherency protocols provide this property.

Counters

```
1 counter(0).
2 counter(X+1)@next :- counter(X), request(_, _).
3 counter(X)@next :- counter(X), notin request(_, _).
4 response(@From, X)@async :- counter(X), request(To, From).
5 response(From, X)@next :- response(From, X).
```

Above we present the Dedalus implementation of the counter-incrementing program shown in Overlog in Listing 2.8. Note that the ambiguities have disappeared; upon receipt of a message, this program replies with the *current* counter value and performs a *deferred* deduction that "updates" the sequence. The temporal annotations allow us to be precise about when deductions represent composite operations across time. Line 2 describes how the value of *counter* changes from one timestep to the next when a *request* message is received. Line 3 describes how *counter* values *persist* across timesteps when there are no *request* messages. Line 5 defines the simplest sort of frame rule: a *basic persistence* rule that establishes the immutability of *response* facts. If a *response* record ever exists, it exists henceforth. Finally, line 4 indicates a communication rule; its consequences hold at another physical location at some unknown future time.

Entanglement

Consider the asynchronous rule below:

```
p(A, B, N)@async ← q(A, B)@N;
```

Due to the `async` keyword in the rule head, each p tuple will take some unspecified time suffix value. Note however that the time suffix N of the rule body appears also in an attribute of p other than the time suffix, recording a binding of the time value of the deduction. We call such a binding an *entanglement*. Note that in order to write the rule it was necessary to not sugar away the time suffix in the rule body.

Entanglement is a powerful construct [22]. It allows a rule to reference the logical clock time of the deduction that produced one (or more) of its subgoals; this capability supports protocols that reason about partial ordering of time across machines. More generally, it exposes the infinite `successor` relation to attributes other than the time suffix, allowing us to express concepts such as infinite sequences.

Lamport Clocks Recall that DEDALUS allows program executions to order message timestamps arbitrarily, violating intuitive notions of causality by allowing deductions to "affect the past." This section explains how to implement Lamport clocks [104] atop DEDALUS, which allows programs to ensure temporal monotonicity by reestablishing a causal order despite derivations that flow backwards through time.

Consider a rule `p(A,B)@async ← q(A,B)`. By rewriting it to:

```
persist[p]
p_wait(A, B, N)@async ← q(A, B)@N;
p_wait(A, B, N)@next ← p_wait(A, B, N)@M, N ≥ M;
p(A, B)@next ← p_wait(A, B, N)@M, N < M;
```

we place the derived tuple in a new relation `p_wait` that stores any tuples that were "sent from the future" with their sending time "entangled"; these tuples stay in the `p_wait` predicate until the point in time at which they were derived. Conceptually, this causes the system to evaluate a potentially large number of timesteps (if N is significantly less than the timestamp of the system when the tuple arrives). However, if the runtime is able to efficiently evaluate timesteps when the database is quiescent, then instead of "waiting" by evaluating timesteps, it will simply increase its logical clock to match that of the sender. In contrast, if the tuple is "sent into the future," then it is processed using the timestep that receives it.

This manipulation of timesteps and clock values is equivalent to conventional descriptions of Lamport clocks, except that our Lamport clock implementation effectively "advances the clock" by preventing derivations until the clock is sufficiently advanced, by temporarily storing incoming tuples in the `p_wait` relation.

Reliable Broadcast

Distributed systems cope with unreliable networks by using mechanisms such as broadcast and consensus protocols, timeouts and retries, and often hide the nondeterminism behind these abstractions. DEDALUS supports these notions, achieving encapsulation of nondeterminism while dealing explicitly with the uncertainty in the model. Consider the simple broadcast protocol below:

```
sbcast(@Member, Sender, Message)@async ← // forward a message to every member
  smessage(@Agent, Sender, Message),
  members(@Agent, Member);

sdeliver(@Member, Sender, Message) ←    // when a message is received, deliver it immediately
  sbcast(@Member, Sender, Message);
```

Assume that `members` is a persistent relation that contains the broadcast membership list. The protocol is straightforward: if a tuple appears in `smessage` (an EDB predicate), then it will be sent to all members (a multicast). The interpretation of the non-deterministic choice implied by the `@async` rule indicates that messaging order and delivery (i.e., finite delay) are not guaranteed.

The program shown below makes use of the multicast primitive provided by the previous program and uses it to implement a basic reliable broadcast using a textbook mechanism [132] that assumes any node that fails to receive a message sent to it has failed. When the broadcast completes, all nodes that have not failed have received the message.

Our simple two-rule broadcast program is augmented with the following rules, so that if a node receives a message, it also multicasts it to every member *before* delivering the message locally:

```
smessage(Agent, Sender, Message)  ← // given a message, use the simple protocol to broadcast it
  rmessage(Agent, Sender, Message);

buf_bcast(Sender, Me, Message)  ←   // buffer messages delivered by the simple protocol.
  sdeliver(Me, Sender, Message);

smessage(Me, Sender, Message)  ←    // and relay the message, using the simple protocol
  buf_bcast(Sender, Me, Message);

rdeliver(Me, Sender, Message)@next  ← // and then locally deliver the message
  buf_bcast(Sender, Me, Message);
```

Note that all network communication is initiated by the `@async` rule from the original simple broadcast. The `@next` is required in the `rdeliver` definition in order to prevent nodes from taking actions based upon the broadcast before it is guaranteed to meet the reliability guarantee.

Semantics

Now that we have established a complete syntax and semantic intuition for Dedalus, we may present its formal, model-theoretic semantics.

Preliminary Definitions

We assume an infinite universe $\mathcal{U}$ of values. We assume $\mathbb{N} = \{0, 1, 2, \ldots\} \subset \mathcal{U}$.

A *relation schema* is a pair $R^{(k)}$ where R is a relation name and k its arity. A *database schema* $\mathcal{S}$ is a set of relation schemas. Any relation name occurs at most once in a database schema.

A *fact* over a relation schema $R^{(n)}$ is a pair consisting of the relation name R and an n-tuple $(c_1, \ldots, c_n)$, where each $c_i \in \mathcal{U}$. We denote a fact over $R^{(n)}$ by $\mathtt{R(c_1,\ \ldots,\ c_n)}$. A *relation instance* for relation schema $R^{(n)}$ is a set of facts whose relation name is R. A *database instance* maps each relation schema $R^{(n)}$ to a corresponding relation instance for $R^{(n)}$.

A *rule* φ consists of several distinct components: a head atom $head(\varphi)$, a set $pos(\varphi)$ of *positive body atoms*, a set $neg(\varphi)$ of *negative body atoms*, a set of inequalities $ineq(\varphi)$ of the form $x < y$, and a set of choice operators $cho(\varphi)$ applied to the variables. The conventional syntax for a rule is:

$$\mathtt{head}(\varphi) \longleftarrow f_1, \ldots, f_n, \neg g_1, \ldots, \neg g_m,\ \mathtt{ineq}(\varphi),\ \mathtt{cho}(\varphi).$$

where $f_i \in pos(\varphi)$ for $i = 1, \ldots, n$ and $g_i \in neg(\varphi)$ for $i = 1, \ldots, m$.

Spatial and Temporal Extensions

Given a database schema $\mathcal{S}$, we use $\mathcal{S}^+$ to denote the schema obtained as follows. For each relation schema $\texttt{r}^{(n)} \in (\mathcal{S} \setminus \{<\})$, we include a relation schema $\texttt{r}^{n+1}$ in $\mathcal{S}^+$. The additional column added to each relation schema is the *location specifier*. By convention, the location specifier is the first column of the relation. $\mathcal{S}^+$ also includes $<^{(2)}$, and a relation schema $\texttt{node}^{(1)}$: the finite set of node identifiers that represents all of the nodes in the distributed system. We call $\mathcal{S}^+$ a *spatial schema*.

A *spatial fact* over a relation schema $R^{(n)}$ is a pair consisting of the relation name R and an $(n+1)$-tuple $(d, c_1, \ldots, c_n)$ where each $c_i \in \mathcal{U}$, $d \in \mathcal{U}$, and `node(d)`. We denote a spatial fact over $R^{(n)}$ by $\texttt{R(d, c}_1\texttt{, ..., c}_n\texttt{)}$. A *spatial relation instance* for a relation schema $\texttt{r}^{(n)}$ is a set of spatial facts for $\texttt{r}^{(n+1)}$. A *spatial database instance* is defined similarly to a database instance.

Given a database schema $\mathcal{S}$, we use $\mathcal{S}^*$ to denote the schema obtained as follows. For each relation schema $\texttt{r}^{(n)} \in (\mathcal{S} \setminus \{<\})$ we include a relation schema $\texttt{r}^{(n+2)}$ in $\mathcal{S}^*$. The first additional column added is the location specifier, and the second is the *timestamp*. By convention, the location specifier is the first column of every relation in $\mathcal{S}^*$, and the timestamp is the last. $\mathcal{S}^*$ also includes $<^{(2)}$ (finite), $\texttt{node}^{(1)}$ (finite), $\texttt{time}^{(1)}$ (infinite) and $\texttt{timeSucc}^{(2)}$ (infinite), We call $\mathcal{S}^*$ a *spatio-temporal schema*.

A *spatio-temporal fact* over a relation schema $R^{(n)}$ is a pair consisting of the relation name R and an $(n+2)$-tuple $(d, t, c_1, \ldots, c_n)$ where each $c_i \in \mathcal{U}$, $d \in \mathcal{U}, t \in \mathcal{U}$, `node(d)`, and `time(t)`. We denote a spatial fact over $R^{(n)}$ by $\texttt{R(d, t, c}_1\texttt{, ..., c}_n\texttt{)}$.

A *spatio-temporal relation instance* for relation schema $\texttt{r}^{(n)}$ is a set of spatio-temporal facts for $\texttt{r}^{(n+2)}$. A *spatio-temporal database instance* is defined similarly to a database instance; in any spatio-temporal database instance, $\texttt{time}^{(1)}$ is mapped to the set containing a `time(t)` fact for all $\texttt{t} \in \mathbb{N}$, and $\texttt{timeSucc}^{(2)}$ to the set containing a `timeSucc(x,y)` fact for all $y = x + 1$, $(x, y \in \mathbb{N})$.

We will use the notation `f@t` to mean the spatio-temporal fact obtained from the spatial fact `f` by adding a timestamp column with the constant `t`.

A *spatio-temporal rule* over a spatio-temporal schema $\mathcal{S}^*$ is a rule of one of the following three forms:

$$\texttt{p(L,}\overline{\texttt{W}}\texttt{, T)} \longleftarrow \texttt{b}_1\texttt{(L,}\overline{\texttt{X}_1}\texttt{, T), ..., b}_l\texttt{(L,}\overline{\texttt{X}_l}\texttt{, T), }\neg\texttt{c}_1\texttt{(L,}\overline{\texttt{Y}_1}\texttt{, T), ..., }\neg\texttt{c}_m\texttt{(L,}\overline{\texttt{Y}_m}\texttt{, T), node(L), time(T), } \mathit{ineq}(\varphi).$$

$$\texttt{p(L,}\overline{\texttt{W}}\texttt{), S} \longleftarrow \texttt{b}_1\texttt{(L,}\overline{\texttt{X}_1}\texttt{, T), ..., b}_l\texttt{(L,}\overline{\texttt{X}_l}\texttt{, T), }\neg\texttt{c}_1\texttt{(L,}\overline{\texttt{Y}_1}\texttt{, T), ..., }\neg\texttt{c}_m\texttt{(L,}\overline{\texttt{Y}_m}\texttt{, T), node(L), time(T),}$$
$$\texttt{timeSucc(T,S), } \mathit{ineq}(\varphi).$$

$$\texttt{p(D,}\overline{\texttt{W}}\texttt{, S)} \longleftarrow \texttt{b}_1\texttt{(L,}\overline{\texttt{X}_1}\texttt{, T), ..., b}_l\texttt{(L,}\overline{\texttt{X}_l}\texttt{, T), }\neg\texttt{c}_1\texttt{(L,}\overline{\texttt{Y}_1}\texttt{, T), ..., }\neg\texttt{c}_m\texttt{(L, }\overline{\texttt{Y}_m}\texttt{, T), node(L), time(T), time(S),}$$
$$\texttt{choice((L, }\overline{\texttt{B}}\texttt{, T),(S)), node(D), } \mathit{ineq}(\varphi).$$

The first corresponds to *deductive* rules, the second to *inductive* rules, and the last to *asynchronous* rules (as presented in Sections 3.1 and 3.1). The last two kinds of rules are collectively called *temporal* rules.

In the rules above, $\overline{\texttt{B}}$ is a tuple containing all the distinct variable symbols in $\overline{\texttt{X}_1}, \ldots, \overline{\texttt{X}_l}, \overline{\texttt{Y}_1}, \ldots, \overline{\texttt{Y}_m}$. The variable symbols `D` and `L` may appear in any of $\overline{\texttt{W}}, \overline{\texttt{X}_1}, \ldots, \overline{\texttt{X}_l}, \overline{\texttt{Y}_1}, \ldots, \overline{\texttt{Y}_m}$, whereas `T` and `S` may not.[1] Head relation name `p` may not be `time`, `timeSucc`, or `node`. Relations $\texttt{b}_1\texttt{, ..., b}_l\texttt{, c}_1\texttt{, ..., c}_m$

[1]Note that in this formalization we rule out temporal *entanglement*, described in Section 3.1

may not be `timeSucc`, `time`, or `<`.

The use of a single location specifier and timestamp in rule bodies corresponds to considering deductions that can be evaluated at a single node at a single point in time.

The `choice` construct—due to Saccà and Zaniolo [147]—is used in this context to model the fact that the network may arbitrarily delay, re-order, and batch messages. We further add the causality rewrite [22], which restricts `choice` in the following way: a message sent by a node x at local timestamp s cannot cause another message to arrive in the past of node x (i.e., at a time before local timestamp s). Intuitively, the causality constraint rules out models corresponding to impossible executions, in which effects are perceived before their causes. Full details about `choice` and the causality constraint are described by Ameloot et al [22].

A Dedalus *program* is a finite set of causally rewritten spatio-temporal rules over some spatio-temporal schema $\mathcal{S}^*$.

Syntactic Sugar

The restrictions on timestamps and location specifiers suggest the natural syntactic sugar that omits boilerplate usage of timestamp attributes and location specifiers, as well as the use of `node`, `time`, `timeSucc`, and `choice` in rule bodies. Example deductive, inductive, and asynchronous rules are shown below.

$\mathtt{p}(\overline{\mathtt{W}}) \leftarrow \mathtt{b}_1(\overline{\mathtt{X}_1}),\ \ldots,\ \mathtt{b}_l(\overline{\mathtt{X}_l}),\ \neg\mathtt{c}_1(\overline{\mathtt{Y}_1}),\ \ldots,\ \neg\mathtt{c}_m(\overline{\mathtt{Y}_m}).$

$\mathtt{p}(\overline{\mathtt{W}})\mathtt{@next} \leftarrow \mathtt{b}_1(\overline{\mathtt{X}_1}),\ \ldots,\ \mathtt{b}_l(\overline{\mathtt{X}_l}),\ \neg\mathtt{c}_1(\overline{\mathtt{Y}_1}),\ \ldots,\ \neg\mathtt{c}_m(\overline{\mathtt{Y}_m}).$

$\mathtt{p}(\overline{\mathtt{W}})\mathtt{@async} \leftarrow \mathtt{b}_1(\overline{\mathtt{X}_1}),\ \ldots,\ \mathtt{b}_l(\overline{\mathtt{X}_l}),\ \neg\mathtt{c}_1(\overline{\mathtt{Y}_1}),\ \ldots,\ \neg\mathtt{c}_m(\overline{\mathtt{Y}_m}).$

In any rule, the body location specifier can be accessed by including a variable symbol or constant prefixed with `@` as the first argument of any body atom. In asynchronous rules, the head location specifier can be accessed in a similar manner in the head atom, as shown in the following rule.

```
p(#D,L,W)@async ← b(#L,D,W), ¬c(#L,L).
```

The head and body location specifiers are `D` and `L` respectively. `D` may appear in the body, `L` may appear in the head, and `L` may appear duplicated in the body.

Model-theoretic semantics

To allow us to reason about the meaning of programs without explicitly reasoning about their executions or about details of a given interpreter, we devised a *declarative* semantics for Dedalus.

These semantics are based on the model-theoretic semantics of Datalog, in which the "meanings" of a program are its minimal models. A *model* of a Datalog program is an assignment of records to relations (essentially, a database) that *satisfies* all the program statements. A minimal model is one that is a subset of all other models A model M is *minimal* if and only if there does not exist a model M' of the program such that $M' \subset M$. Minimal models allow us to consider only facts that are *supported* by the program and its inputs. Pure (i.e., negation-free) Datalog programs have a unique minimal model or *least model*—this model is the program's "meaning." Programs with negation may have multiple least models, or no models.

We only consider Dedalus programs whose deductive rules are syntactically stratified.

An *input schema* $\mathcal{S}^I$ for a Dedalus program P with spatio-temporal schema $\mathcal{S}^*$ is a subset of P's spatial schema $\mathcal{S}^+$. Every input schema contains the `node` relation; we will not explicitly mention the presence of `node` when detailing an input schema. A relation in $\mathcal{S}^I$ is called an *EDB relation*. EDB relations may not appear in the heads of rules. All other relations are called *IDB*.

An *EDB instance* $\mathcal{E}$ is a spatial database instance that maps each EDB relation `r` to a finite spatial relation instance for `r`. The *active domain* of an EDB instance $\mathcal{E}$ and a program P is the set of constants appearing in $\mathcal{E}$ and P. Every EDB instance maps the $<$ relation to a total order over its active domain. We can view an EDB instance as a spatio-temporal database instance $\mathcal{K}$. For every $\mathtt{r(d,c_1,\ldots,c_n)} \in \mathcal{E}$, the fact $\mathtt{r(d,t,c_1,\ldots,c_n)} \in \mathcal{K}$ for all $\mathtt{t} \in \mathbb{N}$. Intuitively, EDB facts "exist at all timesteps."

We refer to a Dedalus program together with an EDB instance as a Dedalus *instance*. The behavior of a Dedalus program can be viewed as a mapping from EDB instances to spatio-temporal database instances.

The principal difficulty in finding an appropriate model-theoretic semantics for Dedalus that described this mappingis how to represent the nondeterminism of the *choice* construct. An asynchronous rule chooses an arbitrary receipt timestamp—hence every legal choice corresponds to a different execution. Because the *time* relation is infinite, there are infinitely many such choices. Hence one natural model-theoretic semantics associates a different model with every possible execution—these are the program's *stable models* [77, 22].

Example 6 Take the following Dedalus program with input schema {`q`}. Assume the EDB instance is {`node(n1), q(n1)`}.

```
p(#L)@async ⟵ q(#L).
```

Let the power set of X be denoted $\mathcal{P}(X)$. For each $S \in \mathcal{P}(\mathbb{N} \setminus \{0\})$, where $|S| = |\mathbb{N}|$, the following are exactly the stable models: $\{\mathtt{node(n1)}\} \cup \{\mathtt{p(n1,i)} \mid \mathtt{i} \in S\} \cup \{\mathtt{q(n1,i)} \mid \mathtt{i} \in \mathbb{N}\}$.

Since `q` is part of the input schema, it is true at every time. Every time involves a separate choice of time for `p`, which must be later than time 0. The causality constraint rules out elements of the power set with finite cardinality [22].

Ultimate Models

Stable models highlight uninteresting temporal differences that may not be "eventually" significant. Intuitively, there would be different stable models for different message orderings, even when those orderings were not meaningful because they represented some commutative operation. In order to rule out such behaviors from the output, we will define the concept of an *ultimate model* to represent a program's "output."

An *output schema* for a Dedalus program P with spatio-temporal schema $\mathcal{S}^*$ is a subset of P's spatial schema $\mathcal{S}^+$. We denote the output schema as $\mathcal{S}^O$.

Let $\Diamond\Box$ map spatio-temporal database instances $\mathcal{T}$ to spatial database instances. For every spatio-temporal fact $\mathtt{r(p,t,c_1,\ldots,c_n)} \in \mathcal{T}$, the spatial fact $\mathtt{r(p,c_1,\ldots,c_n)} \in \Diamond\Box\mathcal{T}$ if relation `r` is in

$\mathcal{S}^O$ and $\forall \texttt{s}\,.\,(\texttt{s} \in \mathbb{N} \wedge \texttt{t} < \texttt{s}) \Rightarrow (\texttt{r(p,s,c}_1\texttt{,...,c}_n\texttt{)} \in \mathcal{T})$. The set of *ultimate models* of a Dedalus instance I is $\{\Diamond\Box(\mathcal{T}) \,|\, \mathcal{T}$ is a stable model of I$\}$. Intuitively, an ultimate model contains exactly the facts in relations in the output schema that are eventually always true in a stable model.

Note that an ultimate model is always finite because of the finiteness of the EDB, the safety conditions on rules, the restrictions on the use of `timeSucc` and `time`, and the prohibition on entanglement. A Dedalus program only has a finite number of ultimate models for the same reason.

Example 7 For Example 6 with $\mathcal{S}^O = \{\texttt{p}\}$, there are two ultimate models: $\{\}$ and $\{\texttt{p(n1)}\}$. The latter corresponds to an element of the power set S such that $\exists x\,.\,\forall y\,.\,(y > x) \Rightarrow (y \in S)$. The former corresponds to an element S that does not have this property.

Discussion

The notion of ultimate models provides a useful finite characterization of the *outcomes* of distributed computations, abstracting away the wide variety of *behaviors* that can arise due to asynchronous communication. A multiplicity of ultimate models implies that a program has a variety of *alternative* outcomes—in a particular execution, details outside the programmer's control (namely, the choice of timestamps assigned by asynchronous rules) decide the outcome. Hence the existence of a *unique* ultimate model for any EDB is a desirable property: such programs are *deterministic* despite the fundamental nondetermism arising in their executions due to asynchrony.

Unfortunately, uniqueness of an ultimate model (called *confluence*) is not decidable in Dedalus [127]. Nevertheless, as we will see in Section 3.2, we can devise practical, conservative tests for confluence based on monotonicity tests, which are well-understood in Datalog-like languages.s.

3.2 Consistency and Logical Monotonicity: the CALM Theorem

In this section we present the connection between distributed consistency and logical monotonicity. This discussion informs the language and analysis tools we develop in subsequent sections.

To guard against partial failure, distributed programs typically employ some form of *redundancy*, such as *replicating* data and computation across multiple nodes, or *replaying* source data when failures are detected. A key problem in distributed computing is reasoning about the consistency of redundancy mechanisms in the face of *temporal non-determinism*: uncertainty about the ordering and timing of message delivery. Different delivery orders may lead to different replica states or non-reproducibility across replays, ultimately defeating the purpose of these mechanisms. To prevent such anomalies, distributed programmers employ *coordination* mechanisms such as distributed locks and ordered broadcast and commit protocols. These mechanisms—which cause local computation to *wait* for a remote signal that it is safe to proceed—incur well-known latency, availability and throughput penalties in large-scale systems [35].

Temporal non-determinism may be unavoidable in distributed systems, but distributed consistency issues tend to arise only when execution non-determinism leaks into program *outcomes*. For example, a message race leads to a concurrency bug only when the program logic can *witness* the outcome of the race, such that it is possible to determine the outcome of the race by inspecting the program's output. The ultimate model semantics of Dedalus (Section 3.1) provided a finite characterization of the outcomes of distributed programs by abstracting away execution details such as concrete timestamps. Each ultimate model of a given program and input represents an equivalence class of execution traces (of which there may be infinitely many) that lead to the same final state. A Dedalus program that witnesses a message race will have at least two ultimate models—one for each race outcome.

A particularly interesting class of Dedalus programs are those that, for any set of inputs, produce a *unique ultimate model* (UUM). These programs are *deterministic* (i.e., they represent a function from inputs to final states) despite widespread non-determinism in their executions. One useful consequence is that these "eventually consistent" programs may be replicated without the use of *coordination*, since replica agreement is guaranteed by the determinism of the program.

Which programs have unique ultimate models? Informally, a sufficient condition is order independence: a program whose outcomes are insensitive to delivery order is surely eventually consistent. Unfortunately, a whole-program analysis for order independence seems elusive. The CALM Theorem [83, 22] provides a more precise and practical answer to this question: *monotonic* Dedalus programs are guaranteed to produce UUMs, just as monotonic Datalog programs generate a unique minimal model [165].

This insight behind the CALM Theorem generalizes beyond Dedalus. Monotonic programs—e.g., programs expressible via selection, projection and join (even with recursion)—can be implemented by streaming algorithms that incrementally produce output elements as they receive input elements. The final order or contents of the input will never cause any earlier output to be revoked once it has been generated. Non-monotonic programs—e.g., those that contain aggregation or negation operations—seem to require blocking algorithms that do not produce any output until they have received all tuples in logical partitions of an input set. For example, aggregation queries need to receive entire "groups" before producing aggregates, which in general requires receiving the entire input set. As a result, even simple non-monotonic tasks like counting are difficult in distributed systems.

We can use the CALM principle to develop checks for distributed consistency in logic languages, where conservative tests for monotonicity are well understood. A simple syntactic check is often sufficient: if the program does not contain any of the symbols in the language that correspond to non-monotonic operators (e.g., `NOT IN` or aggregate symbols), then it is monotonic and can be implemented without coordination, regardless of any read-write dataflow dependencies in the code. As students of the logic programming literature will recognize [141, 145, 146], these conservative checks can be refined further to consider semantics of predicates in the language. For example, the expression "`MIN(x) < 100`" is monotonic despite containing an aggregate, by virtue of the semantics of `MIN` and $<$: once a subset S satisfies this test, any superset of S will also satisfy it. Many refinements along these lines exist, increasing the ability of program analyses to verify monotonicity.

In cases where an analysis cannot guarantee monotonicity of a whole program, it can instead provide a conservative assessment of the points in the program where coordination may be required to ensure consistency. For example, a shallow syntactic analysis can flag all non-monotonic predicates in a program (e.g., `NOT IN` tests or predicates with aggregate values as input). The loci produced by a non-monotonicity analysis are the program's *points of order*. A program with non-monotonicity can be made consistent by including coordination logic at its points of order.

Because analyses based on the CALM principle operate with information about program semantics, they can avoid coordination logic in cases where traditional read/write analysis would require it. Perhaps more importantly, as we will see in our discussion of shopping carts (Section 3.3), logic languages and the analysis of points of order can help programmers redesign code to reduce coordination requirements.

3.3 Bloom

We have illustrated the benefits of data-centric programming with a disorderly language. Like Overlog, Dedalus made it easy to write succinct, intuitive "executable specifications." Unlike Overlog, Dedalus provides a precise semantics that captures time-varying state and uncertain communication, allowing us to reason about program correctness without resorting to operational reasoning (e.g., "the program means whatever it is that this interpreter I've written does" [165]).

Protocols written in Dedalus often fit on a single page, and can be compared line-by-line with pseudocode specifications. However, what happens when programs grow far beyond what can fit on a page? Rule-based languages offer no structuring or scoping mechanisms: as the complexity of a program increases, it becomes harder to understand how different rules interact.

In this section, we introduce Bloom, a practical language for programming distributed systems. Below the surface, Bloom is almost indistinguishable from Dedalus: it shares the same first-order data model and model-theoretic semantics. Bloom differs from Dedalus in the following ways:

1. *Bloom provides tools that exploit the CALM Theorem to provide guarantees that submitted programs are deterministic or, if such a guarantee cannot be made, to identify program locations where coordination may be required to ensure determinism.* In this respect Bloom is a complete disorderly programming solution. It not only makes it easy and natural to express order-independent distributed programs, but precisely identifies those programs for which order has been *underspecified*.

2. *Bloom provides a module system that supports structuring, hiding and reuse.* As we will see, Bloom's module system overcomes many of the software engineering limitations of rule-based languages such as Datalog, which (despite their brevity) become increasingly difficult to reason about as program size (and hence the possible interactions among rules) grows. Bloom allows programmers to define components with external interfaces that hide the details of internal implementations, and to construct larger components out of smaller ones using mixin composition. Bloom modules may be defined at a variety of granularities, from the scale of reusable protocol design idioms such as those presented in Section 2.2 to that of distributed services in a service-oriented architecture.

3. *Bloom provides a programmer-friendly syntax in which rules are expressed as comprehensions over relations.* Anecdotally, a great many programmers find Datalog-style rules difficult to read. Much of this difficulty is due to shallow issues with Prolog-style syntax: positional attribute notation and implicit, unification-based joins are burdensome when predicates have high arity. To improve readability of rules, Bloom collections have explicit names provided when relations are defined. The right-hand side of rules are then any collection-valued expression matching the schema of the left-hand side.

Bud: Bloom Under Development

Like Overlog and Dedalus, Bloom is designed in the tradition of programming styles that are "disorderly" by nature. State is captured in unordered relations. Computation is expressed in logic: an unordered set of declarative rules, each consisting of an unordered conjunction of predicates. As we discuss below, mechanisms for imposing order are available when needed, but the programmer is provided with tools to evaluate the need for these mechanisms as special-case behaviors, rather than a default model. The result is code that runs naturally on distributed machines with a minimum of coordination overhead.

The prototype version of Bloom we describe here is embodied in an implementation we call *Bud* (Bloom Under Development). Bud is a domain-specific subset of the popular Ruby scripting language and is evaluated by a stock Ruby interpreter via a `Bud` Ruby class. Bud uses a Ruby-flavored syntax, but this is not fundamental; we have experimented with analogous Bloom embeddings in other languages including Python, Erlang and Scala, and they look similar in structure.

Bloom Basics

Bloom programs are bundles of declarative statements about collections of "facts" or tuples, similar to SQL views or Datalog rules. A statement can only reference data that is local to a node. Bloom statements are defined with respect to atomic "timesteps," which can be implemented via successive rounds of evaluation. In each timestep, certain "ground facts" exist in collections due to persistence or the arrival of messages from outside agents (e.g., the network or system clock). The statements in a Bloom program specify the derivation of additional facts, which can be declared to exist either in the current timestep, at the very next timestep, or at some non-deterministic time in the future at a remote node.

A Bloom program also specifies the way that facts persist (or do not persist) across consecutive timesteps on a single node. Bloom is a side-effect free language with no "mutable state": if a fact is defined at a given timestep, its existence at that timestep cannot be refuted by any expression in the language. This technicality is key to avoiding many of the complexities involved in reasoning about earlier "stateful" rule languages.

Type	Behavior
table	A collection whose contents persist across timesteps.
scratch	A collection whose contents persist for only one timestep.
channel	A scratch collection with one attribute designated as the *location specifier*. Tuples "appear" at the network address stored in their location specifier.
periodic	A scratch collection of key-value pairs (`id`, `timestamp`). The definition of a periodic collection is parameterized by a `period` in seconds; the runtime system arranges (in a best-effort manner) for tuples to "appear" in this collection approximately every `period` seconds, with a unique `id` and the current wall-clock time.
interface	A scratch collection specially designated as an interface point between modules.

Op	Valid lhs types	Meaning
<=	**table**, **scratch**	lhs includes the content of the rhs in the current timestep.
<+	**table**, **scratch**	lhs will include the content of the rhs in the next timestep.
<-	**table**	tuples in the rhs will be absent from the lhs at the start of the next timestep.
<~	**channel**	tuples in the rhs will appear in the (remote) lhs at some non-deterministic future time.

Figure 3.1: Bloom collection types and operators.

State in Bloom

Bloom programs manage state using five collection types described in the top of Figure 3.1. A collection is defined with a relational-style schema of named columns, including an optional subset of those columns that forms a primary key. Line 16 in Listing 3.1 defines a collection named `send_buf` with four columns `dst`, `src`, `ident`, and `payload`; the primary key is `(dst, src, ident)`. Column values are just Ruby objects, so it is possible to have a column based on any Ruby class the programmer cares to define or import (including nested Bud collections). In Bud, a tuple in a collection is simply a Ruby array containing as many elements as the columns of the collection's schema. As in other object-relational ADT schemes like Postgres [157], column values can be manipulated using their own (non-destructive) methods. Bloom also provides for nesting and unnesting of collections using standard Ruby constructs like `reduce` and `flat_map`. Note that collections in Bloom provide set semantics—collections do not contain duplicates.

In contrast to DEDALUS, in which all tuples are ephemeral and notions such as persistence are programmatic constructs, in Bloom the persistence of a tuple is determined by the type of the collection that contains it. Ephemeral **scratch** collections are useful for transient data like intermediate results and "macro" definitions that enable code reuse. The contents of a **table** persist across consecutive timesteps (until that persistence is interrupted via a Bloom statement containing the <- operator described below). Bloom tables correspond to DEDALUS relations for which there exists a persistence rule, as described in Section 3.1. It is also convenient to think operationally as follows: scratch collections are "emptied" before each timestep begins, tables are "stored" collections (similar to tables in SQL), and the <- operator represents batch deletion before the beginning of

```
 1  module DeliveryProtocol
 2    def state
 3      interface input, :pipe_in,
 4        [:dst, :src, :ident] => [:payload]
 5      interface output, :pipe_sent,
 6        [:dst, :src, :ident] => [:payload]
 7    end
 8  end

10  module ReliableDelivery
11    include DeliveryProtocol

13    state do
14      channel :data_chan, [@:dst, :src, :ident] => [:payload]
15      channel :ack_chan, [@:src, :dst, :ident]
16      table :send_buf, [:dst, :src, :ident] => [:payload]
17      periodic :timer, 10
18    end

20    bloom :send_packet do
21      send_buf <= pipe_in
22      data_chan <~ pipe_in
23    end

25    bloom :timer_retry do
26      data_chan <~ (send_buf * timer).lefts
27    end

29    bloom :send_ack do
30      ack_chan <~ data_chan{|p| [p.src, p.dst, p.ident]}
31    end

33    bloom :recv_ack do
34      temp :got_ack = (ack_chan * send_buf).rights(:ident => :ident)
35      pipe_sent <= got_ack
36      send_buf <- got_ack
37    end
38  end
```

Listing 3.1: Reliable unicast messaging in Bloom.

the next timestep.

The facts of the "real world," including network messages and the passage of wall-clock time, are captured via **channel** and **periodic** collections; these are scratch collections whose contents "appear" at non-deterministically-chosen timesteps. Note that failure of nodes or communication is captured here: it can be thought of as the repeated "non-appearance" of a fact at every timestep. Again, it is convenient to think operationally as follows: the facts in a channel are sent to a remote node via an unreliable transport protocol like UDP; the address of the remote node is indicated by a distinguished column in the channel called the *location specifier* (denoted by the symbol `@`). The definition of a periodic collection instructs the runtime to "inject" facts at regular wall-clock intervals to "drive" further derivations. Lines 14 and 17 in Listing 3.1 contain examples of channel and periodic definitions, respectively.

The final type of collection is an **interface**, a scratch collection which specifies a connection point between Bloom modules. Interfaces are described in Section 3.3.

Bloom Statements

Bloom statements are declarative relational expressions that define the contents of derived collections. They can be viewed operationally as specifying the insertion or accumulation of expression results into collections. The syntax is:

<collection-variable> <op> <collection-expression>

The bottom of Figure 3.1 describes the five operators that can be used to define the contents of the left-hand side (lhs) in terms of the right-hand side (rhs). As in Datalog, the lhs of a statement may be referenced recursively in its rhs, or recursion can be expressed mutually across statements.

In the Bud prototype, both the lhs and rhs are instances of (a descendant of) a Ruby class called `BudCollection`, which supports methods for manipulating collections inherited from the standard Ruby `Enumerable` mixin. The rhs of a statement typically invokes `BudCollection` methods on one or more collection objects to produce a derived collection. The most commonly used method is `map`, which applies a scalar operation to every tuple in a collection; this can be used to implement relational selection and projection. For example, line 30 of Listing 3.1 projects the `data_chan` collection to its `src`, `dst`, and `ident` fields. Multiway joins are specified using the `join` method, which takes a list of input collections and an optional list of join conditions. Line 34 of Listing 3.1 shows a join between `ack_chan` and `send_buf`. Syntax sugar for natural joins and outer joins is also provided. `BudCollection` also defines a `group` method similar to SQL's `GROUP BY`, supporting the standard SQL aggregates; for example, lines 16–18 of Listing 3.9 compute the sum of `reqid` values for every combination of values for `item` and `session`.

Bloom statements are specified within method definitions that are flagged with the `bloom` keyword (e.g., line 20 of Listing 3.1). The semantics of a Bloom program are defined by the union of its `bloom` blocks; the order of statements is immaterial. Dividing statements into multiple blocks improves the readability of the program and also allows the use of Ruby's method overriding and inheritance features: because a Bloom class is just a stylized Ruby class, any of the methods in a Bloom class can be overridden by a subclass. We expand upon this idea next.

Modules and Interfaces

Conventional wisdom in certain quarters says that rule-based languages are untenable for large programs that evolve over time, since the interactions among rules become too difficult to understand. Bloom addresses this concern in two different ways. First, unlike many prior rule-based languages, Bloom is purely declarative; this avoids forcing the programmer to reason about the interaction between declarative statements and imperative constructs. Second, Bloom borrows object-oriented features from Ruby to enable programs to be broken into small modules and to allow modules to interact with one another by exposing narrow interfaces. This aids program comprehension, because it reduces the amount of code a programmer needs to read to understand the behavior of a module.

A Bloom module is a bundle of collections and statements. Like modules in Ruby, a Bloom module can "mixin" one or more other modules via the `include` statement; mixing-in a module imports its collections and statements. A common pattern is to specify an abstract interface in one module and then use the mixin feature to specify several concrete realizations in separate modules. To support this idiom, Bloom provides a special type of collection called an **interface**. An input interface defines a relation from which a module accepts stimuli from the outside world (e.g., other Bloom modules). Typically, inserting a fact into an input interface results in a corresponding fact appearing (perhaps after a delay) in one of the module's output interfaces.

For example, the DeliveryProtocol module in Listing 3.1 defines an abstract interface for sending messages to a remote address. Clients use an implementation of this interface by inserting a fact into `pipe_in`; this represents a new message to be delivered. A corresponding fact will eventually appear in the `pipe_sent` output interface; this indicates that the delivery operation has been completed. The ReliableDelivery module of Listing 3.1 is one possible implementation of the abstract DeliveryProtocol interface—it uses a buffer and acknowledgment messages to delay emitting a `pipe_sent` fact until the corresponding message has been acknowledged by the remote node. Figure 3.2 contains a different implementation of the abstract DeliveryProtocol. A client program that is indifferent to the details of message delivery can simply interact with the abstract DeliveryProtocol; the particular implementation of this protocol can be chosen independently.

```
module BestEffortDelivery
  include DeliveryProtocol

  state do
    channel :pipe_chan,
      [:@dst, :src, :ident] => [:payload]
  end

  bloom :snd do
    pipe_chan <~ pipe_in
  end

  bloom :done do
    pipe_sent <= pipe_in
  end
end
```

Figure 3.2: Best-effort unicast messaging in Bloom.

A common requirement is for one module to "override" some of the statements in a module that it mixes in. For example, an OrderedDelivery module might want to reuse the functionality provided by ReliableDelivery but prevent a message with sequence number x from being delivered until all messages with sequence numbers $< x$ have been acknowledged. To support this pattern, Bloom allows an interface defined in another module to be overridden simply by redeclaring it. Internally, both of these redundantly-named interfaces exist in the namespace of the module that declared them, but they only need to be referenced by a fully qualified name if their use is otherwise ambiguous. If an input interface appears in the lhs of a statement in a module that declared the interface, it is rewritten to reference the interface with the same name in a mixed-in class, because a module cannot insert into its own input interface. The same is the case for output interfaces appearing in the rhs of statements. This feature allows programmers to reuse existing modules and interpose additional logic in a style reminiscent of superclass invocation in object-oriented languages. We provide an example of interface overriding in Section 3.3.

Bud Implementation

Bud is intended to be a lightweight rapid prototype of Bloom: a first effort at embodying the Dedalus logic in a syntax familiar to programmers. The initial implementation of Bud, developed in 2010, consisted of less than 2400 lines of Ruby code, developed as a part-time effort over the course of a semester.

A Bud program is just a Ruby class definition. To make it operational, a small amount of imperative Ruby code is needed to create an instance of the class and invoke the Bud `run` method. This imperative code can then be launched on as many nodes as desired. As an alternative to the `run` method, the Bud class also provides a `tick` method that can be used to force evaluation of a single timestep; this is useful for debugging Bloom code with standard Ruby debugging tools or for executing a Bud program that is intended as a "one-shot" query.

Because Bud is pure Ruby, some programmers may choose to embed it as a domain-specific language (DSL) within traditional imperative Ruby code. In fact, nothing prevents a subclass of Bud from having both Bloom code in `declare` methods and imperative code in traditional Ruby methods. This is a fairly common usage model for many DSLs. A mixture of declarative Bloom methods and imperative Ruby allows the full range of existing Ruby code—including the extensive RubyGems repositories—to be combined with checkable distributed Bloom programs. The analyses we describe in the remaining sections still apply in these cases; the imperative Ruby code interacts with the Bloom logic in the same way as any external agent sending and receiving network messages.

Case Study: Key-Value Store

In this section, we present two variants of a key-value store (KVS) implemented using Bloom. We begin with an abstract protocol that any key-value store will satisfy, and then provide both single-node and replicated implementations of this protocol. We then introduce a graphical visualization of the dataflow in a Bloom program and use this visualization to reason about the *points of order* in our programs: places where additional coordination may be required to guarantee consistent results.

Abstract Key-Value Store Protocol

Listing 3.2 specifies a protocol for interacting with an abstract key-value store. The protocol comprises two input interfaces (representing attempts to insert and fetch items from the store) and a single output interface (which represents the outcome of a fetch operation). To use an implementation of this protocol, a Bloom program can store key-value pairs by inserting facts into `kvput`. To

```
module KVSProtocol
  state do
    interface input, :kvput,
      [:client, :key, :reqid] => [:value]
    interface input, :kvget, [:reqid] => [:key]
    interface output, :kvget_response =>
      [:reqid], [:key, :value]
  end
end
```

Listing 3.2: Abstract key-value store protocol.

```
 1 module BasicKVS
 2   include KVSProtocol

 4   state do
 5     table :kvstate, [:key] => [:value]
 6   end

 8   bloom :do_put do
 9     kvstate <+ kvput{|p| [p.key, p.value]}
10     kvstate <- (kvstate * kvput).lefts(:key => :key)
11   end

13   bloom :do_get do
14     temp :getj <= (kvget * kvstate).pairs(:key => :key)
15     kvget_response <= getj do |g, t|
16       [g.reqid, t.key, t.value]
17     end
18   end
19 end
```

Listing 3.3: Single-node key-value store implementation.

retrieve the value associated with a key, the client program inserts a fact into `kvget` and looks for a corresponding response tuple in `kvget_response`. For both put and get operations, the client must supply a unique request identifier (`reqid`) to differentiate tuples in the event of multiple concurrent requests.

A module which uses a key-value store but is indifferent to the specifics of the implementation may simply mixin the abstract protocol and postpone committing to a particular implementation until runtime. As we will see shortly, an implementation of the KVSProtocol is a collection of Bloom statements that read tuples from the protocol's input interfaces and send results to the output interface.

Single-Node Key-Value Store

Figure 3.3 contains a single-node implementation of the abstract key-value store protocol. Key-value pairs are stored in a persistent table called `kvstate` (line 5). When a `kvput` tuple is received, its key-value pair is stored in `kvstate` at the next timestep (line 9). If the given key already exists in `kvstate`, we want to replace the key's old value. This is done by joining `kvput` against the current

```
1  module ReplicatedKVS
2    include BasicKVS
3    include MulticastProtocol

5    state do
6      interface input, :kvput,
7        [:client, :key, :reqid] => [:value]
8    end

10   bloom :replicate do
11     temp :mcasts =  kvput.notin(members, :client => :peer)
12     send_mcast <= mcasts{|m| [m.reqid, [@local_addr, k.key, k.reqid, k.value]]}
13   end

15   bloom :apply_put do
16     kvput <= mcast_done{|m| m.payload}

18     kvput <= pipe_chan do |d|
19       if d.payload.fetch(1) != @local_addr
20         d.payload
21       end
22     end
23   end
24 end
```

Listing 3.4: Replicated key-value store implementation.

version of `kvstate` (line 10). If a matching tuple is found, the old key-value pair is removed from `kvstate` at the beginning of the next timestep (line 10). Note that we also insert the new key-value pair into `kvstate` in the next timestep (line 9); hence, an overwriting update is implemented as an atomic deletion and insertion.

Replicated Key-Value Store

Next, we extend the basic key-value store implementation to support replication (Listing 3.4). To communicate between replicas, we use a simple multicast library implemented in Bloom, shown in Figure 3.3. To send a multicast, a program inserts a fact into `send_mcast`; a corresponding fact appears in `mcast_done` when the multicast is complete. The multicast library also exports the membership of the multicast group in a table called `members`.

Our replicated key-value store is implemented on top of the single-node key-value store described in the previous section. When a new key is inserted by a client, we multicast the insertion to the other replicas (lines 11–12). We apply an update to our local `kvstate` table in two cases: (1) if a multicast succeeds at the node that originated it (line 16) (2) whenever a multicast is received by a peer replica (lines 18–22). Note that `@local_addr` is a Ruby instance variable defined by Bud that contains the network address of the current Bud instance.

In Listing 3.4 ReplicatedKVS "intercepts" `kvput` events from clients, and only applies them to the underlying BasicKVS module when certain conditions are met. To achieve this, we "override" the declaration of the `kvput` input interface as discussed in Section 3.3 (lines 6–7). In ReplicatedKVS, references to `kvput` appearing in the lhs of statements are resolved to the `kvput` provided

```
module MulticastProtocol
  state do
    table :members, [:peer]
    interface input, :send_mcast, [:ident] => [:payload]
    interface output, :mcast_done, [:ident] => [:payload]
  end
end

module SimpleMulticast
  include MulticastProtocol
  include DeliveryProtocol

  bloom :snd_mcast do
    pipe_in <= (send_mcast * members).pairs do |s, m|
      [m.peer, @local_addr, s.ident, s.payload]
    end
  end

  bloom :done_mcast dp
    mcast_done <= pipe_sent{|p| [p.ident, p.payload]}
  end
end
```

Figure 3.3: A simple unreliable multicast library in Bloom.

```
1 class RealizedReplicatedKVS < Bud
2   include ReplicatedKVS
3   include SimpleMulticast
4   include BestEffortDelivery
5 end

7 kvs = RealizedReplicatedKVS.new("localhost", 12345)
8 kvs.run
```

Listing 3.5: A fully specified key-value store program.

by BasicKVS, while references in the rhs of statements resolve to the local `kvput`. As described in Section 3.3, this is unambiguous because a module cannot insert into its own input or read from its own output interfaces.

Listing 3.5 combines ReplicatedKVS with a concrete implementation of MulticastProtocol and DeliveryProtocol. The resulting class, a subclass of Bud, may be instantiated and run as shown in lines 7 and 8.

Predicate Dependency Graphs

Now that we have introduced two concrete implementations of the abstract key-value store protocol, we turn to analyzing the properties of these programs. We begin by describing the graphical dataflow representation used by our analysis. In the following section, we discuss the dataflow graphs generated for the two key-value store implementations.

A Bloom program may be viewed as a dataflow graph with external input interfaces as sources, external output interfaces as sinks, collections as internal nodes, and rules as edges. This graph

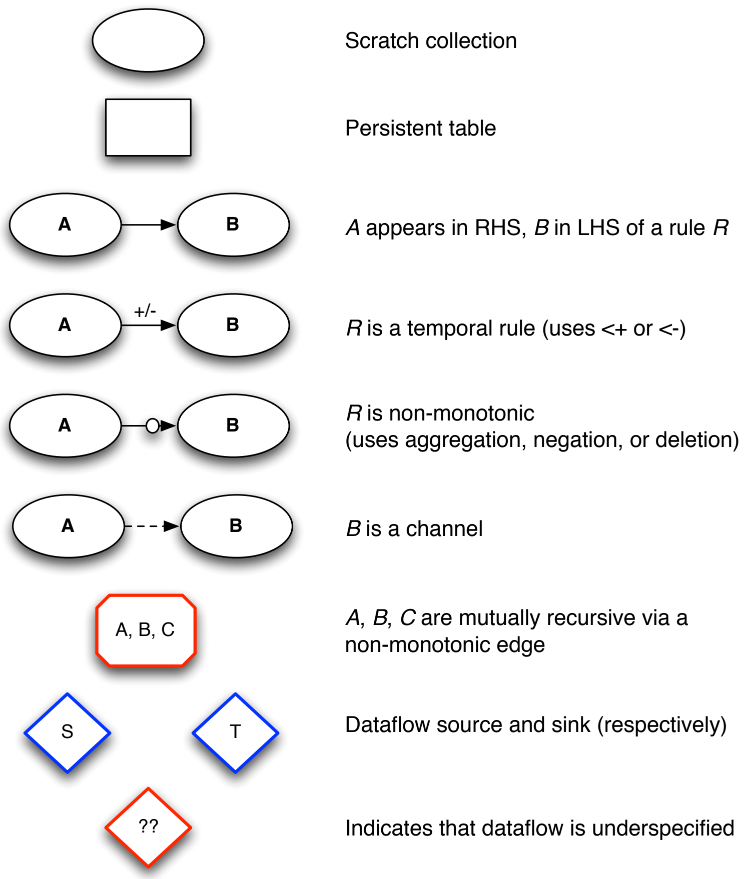

Scratch collection
Persistent table
A
B
A appears in RHS, *B* in LHS of a rule *R*
+/-
R is a temporal rule (uses <+ or <-)
R is non-monotonic
(uses aggregation, negation, or deletion)
B is a channel
A, B, C
A, *B*, *C* are mutually recursive via a non-monotonic edge
S
T
Dataflow source and sink (respectively)
??
Indicates that dataflow is underspecified

Figure 3.4: Visual analysis legend.

represents the dependencies between the collections in a program and is generated automatically by the Bud interpreter. Figure 3.4 contains a list of the different symbols and annotations in the graphical visualization; we provide a brief summary below.

Each node in the graph is either a collection or a cluster of collections; tables are shown as rectangles, ephemeral collections (scratch, periodic and channel) are depicted as ovals, and clusters (described below) as octagons. A directed edge from node A to node B indicates that B appears in the lhs of a Bloom statement that references A in the rhs, either directly or through a join expression. An edge is annotated based on the operator symbol in the statement. If the statement uses the <+ or <− operators, the edge is marked with "+/−". This indicates that facts traversing the edge "spend" a timestep to move from the rhs to the lhs. Similarly, if the statement uses the <∼ operator, the edge is a dashed line—this indicates that facts from the rhs appear at the lhs at a non-deterministic future time. If the statement involves a non-monotonic operation (aggregation, negation, or deletion via the <− operator), then the edge is marked with a white circle. To make the visualizations more readable, any strongly connected component marked with both a circle and a +/− edge is collapsed into an octagonal "temporal cluster," which can be viewed abstractly as a single, non-monotonic node in the dataflow. Any non-monotonic edge in the graph is a *point of*

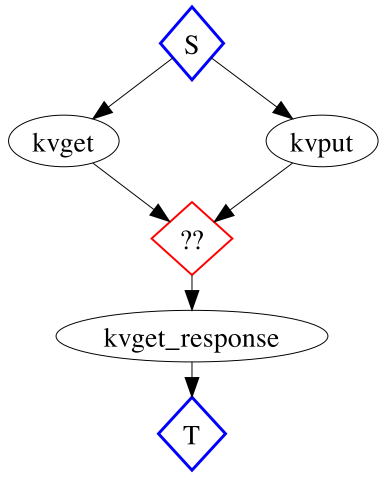


Figure 3.5: Visualization of the abstract key-value store protocol.

order, as are all edges incident to a temporal cluster, including their implicit self-edge.

Analysis

Figure 3.5 presents a visual representation of the abstract key-value store protocol. Naturally, the abstract protocol does not specify a connection between the input and output events; this is indicated in the diagram by the red diamond labeled with "??", denoting an underspecified dataflow. A concrete realization of the key-value store protocol must, at minimum, supply a dataflow that connects an input interface to an output interface.

Figure 3.6 shows the visual analysis of the single-node KVS implementation, which supplies a concrete dataflow for the unspecified component in the previous graph. `kvstate` and `prev` are collapsed into a red octagon because they are part of a strongly connected component in the graph with both negative and temporal edges. Any data flowing from `kvput` to the sink must cross at least one non-monotonic point of order (at ingress to the octagon) and possibly an arbitrary number of them (by traversing the dependency cycle collapsed into the octagon), and any path from `kvget` to the sink must join state potentially affected by non-monotonicity (because `kvstate` is used to derive `kvget_response`).

Reviewing the code in Listing 3.3, we see the source of the non-monotonicity. The contents of `kvstate` may be defined via a "destructive" update that combines the previous state and the current input from `kvput` (lines 9–10 of Listing 3.3). Hence the contents of `kvstate` may depend on the order of arrival of `kvput` tuples.

Case Study: Shopping Cart

In this section, we develop two different designs for a distributed shopping-cart service in Bloom. In a shopping cart system, clients add and remove items from their shopping cart. To provide fault

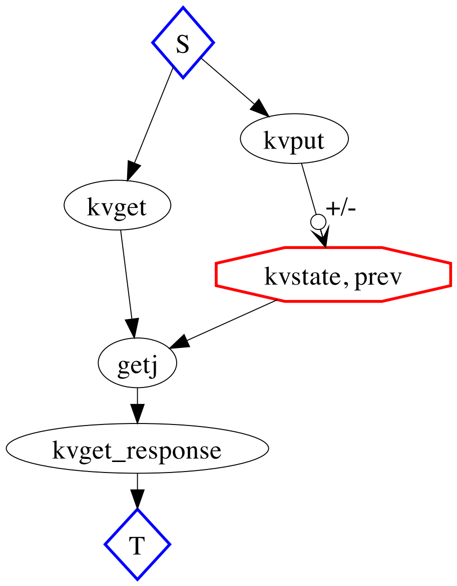


Figure 3.6: Visualization of the single-node key-value store.

tolerance and persistence, the content of the cart is stored by a collection of server replicas. Once a client has finished shopping, they perform a “checkout” request, which returns the final state of their cart.

After presenting the abstract shopping cart protocol and a simple client program, we implement a “destructive,” state-modifying shopping cart service that uses the key-value store introduced in Section 3.3. Second, we illustrate a “disorderly” cart that accumulates updates in a set-wise fashion, summarizing updates at checkout into a final result. These two different designs illustrate our analysis tools and the way they inform design decisions for distributed programming.

Shopping Cart Client

An abstract shopping cart protocol is presented in Listing 3.6. Listing 3.7 contains a simple shopping cart client program: it takes client operations (represented as `client_action` and `client_checkout` facts) and sends them to the shopping cart service using the CartProtocol. We omit logic for clients to choose a cart server replica; this can be based on simple policies like round-robin or random selection, or via more explicit load balancing.

“Destructive” Shopping Cart Service

We begin with a shopping cart service built on a key-value store. Each cart is a `(key,value)` pair, where `key` is a unique session identifier and `value` is an object containing the session’s state: in this example, a Ruby hash that holds the items currently in the cart and their counts. Adding or deleting items from the cart result in “destructive” updates: the value associated with the key is

```
module CartProtocol
  state do
    channel :action_msg,
      [:@server, :client, :session, :reqid] => [:item, :cnt]
    channel :checkout_msg, [:@server, :client, :session, :reqid]
    channel :response_msg,
      [:@client, :server, :session] => [:items]
  end
end

module CartClientProtocol
  state do
    interface input, :client_action, [:server, :session, :reqid] => [:item, :cnt]
    interface input, :client_checkout, [:server, :session, :reqid]
    interface output, :client_response, [:client, :server, :session] => [:items]
  end
end
```

Listing 3.6: Abstract shopping cart protocol.

```
module CartClient
  include CartProtocol
  include CartClientProtocol

  bloom :client do
    action_msg <~ client_action {|a| [a.server, ip_port, a.session, a.reqid, a.item, a.cnt]}
    checkout_msg <~ client_checkout {|a| [a.server, ip_port, a.session, a.reqid]}
    client_response <= response_msg
  end
end
```

Listing 3.7: Shopping cart client implementation.

```
module DestructiveCart
  include CartProtocol
  include KVSProtocol

  bloom :on_action do
    kvget <= action_msg {|a| [a.reqid, a.session] }
    kvput <= (action_msg * kvget_response).outer(:reqid => :reqid) do |a,r|
      val = r.value || {}
      [a.client, a.session, a.reqid, val.merge({a.item => a.cnt}) {|k,old,new| old + new}]
    end
  end

  bloom :on_checkout do
    kvget <= checkout_msg {|c| [c.reqid, c.session] }
    response_msg <~ (kvget_response * checkout_msg).pairs(:reqid => :reqid) do |r,c|
      [c.client, c.server, r.key, r.value.select {|k,v| v > 0}.sort]
    end
  end
end
```

Listing 3.8: Destructive cart implementation.

```
 1  module DisorderlyCart
 2    include CartProtocol

 4    state do
 5      table :action_log, [:session, :reqid] => [:item, :cnt]
 6      scratch :item_sum, [:session, :item] => [:num]
 7      scratch :session_final, [:session] => [:items, :counts]
 8    end

10    bloom :on_action do
11      action_log <= action_msg {|c| [c.session, c.reqid, c.item, c.cnt] }
12    end

14    bloom :on_checkout do
15      temp :checkout_log <= (checkout_msg * action_log).rights(:session => :session)
16      item_sum <= checkout_log.group([:session, :item], sum(:cnt)) do |s|
17        s if s.last > 0
18      end
19      session_final <= item_sum.group([:session], accum_pair(:item, :num))
20      response_msg <~ (session_final * checkout_msg).pairs(:session => :session) do |c,m|
21        [m.client, m.server, m.session, c.items.sort]
22      end
23    end
24  end
```

Listing 3.9: Disorderly cart implementation.

replaced by a new value that reflects the effect of the update. Deletion requests are ignored if the item they refer to does not exist in the cart.

Listing 3.8 shows the Bloom code for this design. The `kvget` collection is provided by the abstract KVSProtocol described in Section 3.3. Our shopping cart service would work with any concrete realization of the KVSProtocol; we will choose to use the replicated key-value store (Section 3.3) to provide fault-tolerance.

When client actions arrive from the CartClient, the cart service checks to see if there is a record in the key-value store associated with the client's session by inserting a record into the `kvput` interface(Line 6). It then performs an outer join between the client action record and the response record from the KVS (Line 7). f no record is found (i.e., this is the first operation for a new session), then the service initializes `val` to an empty Ruby hash ; otherwise, it assigns the existing session state (a hash mapping items to counts) to `val` (Line 8). Finally, it inserts a new record (implicitly replacing the old one according to the semantics of the KVS) containing, for each item, a revised sum (Line 9).

"Disorderly" Shopping Cart Service

Listing 3.9 shows an alternative shopping cart implementation, in which updates are monotonically accumulated in a set, and summed up only at checkout. Line 11 insert client updates into the persistent table `action_log`. When a checkout message arrives, Line 15 looks up the appropriate records in `action_log` for that session. Lines 16-18 compute, for each item, the total number of occurrences in the cart by summing its additions and deletions (if the final total is negative, it is

```
class ReplicatedDisorderlyCart < Bud
  include DisorderlyCart
  include SimpleMulticast
  include BestEffortDelivery

  bloom :replicate do
    send_mcast <= action_msg do |a|
      [a.reqid, [a.session, a.item, a.action, a.reqid]]
    end
    cart_action <= pipe_chan{|c| c.payload }
  end
end
```

Figure 3.7: The complete disorderly cart program.

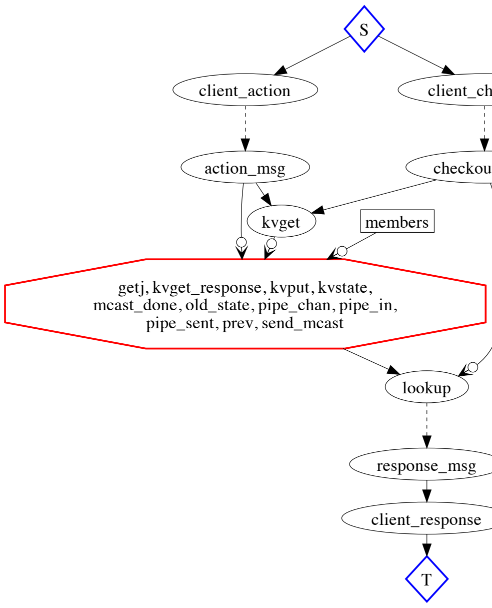

S
client_action
client_checkout
action_msg
checkout_msg
kvget
members
getj, kvget_response, kvput, kvstate, mcast_done, old_state, pipe_chan, pipe_in, pipe_sent, prev, send_mcast
lookup
response_msg
client_response
T

Figure 3.8: Visualization of the destructive cart program.

discarded). Finally, it assembles a new list of `item, num` pairs for each session (Line 19) and sends these to the client in a `response_msg` (Lines 20-22).

Analysis

Figure 3.8 presents the analysis of the "destructive" shopping cart variant. Note that because all dependencies are analyzed, collections defined in mixins but not referenced in the code sample (e.g., `pipe_chan`, `member`) also appear in the graph. Although there is no syntactic non-monotonicity in Listing 3.8, the underlying key-value store uses the non-monotonic `<-` operator to model updateable state. Thus, while the details of the implementation are encapsulated by the key-value store's abstract interface, its points of order resurface in the full-program analysis. Figure 3.8 indicates that there are points of order between `action_msg`, `member`, and the temporal cluster. This figure

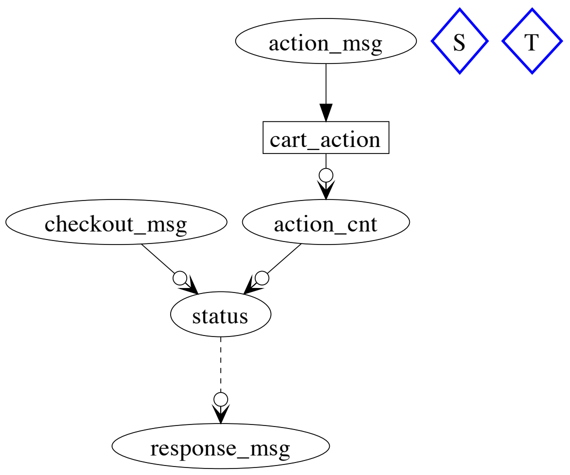


Figure 3.9: Visualization of the core logic for the disorderly cart.

also tells the (sad!) story of how we could ensure consistency of the destructive cart implementation: introduce coordination between client and server—and between the chosen server and all its replicas—for *every client action or kvput update*. The programmer can achieve this coordination by supplying a "reliable" implementation of multicast that awaits acknowledgments from all replicas before reporting completion: this fine-grained coordination is akin to "eager replication" [75]. Unfortunately, it would incur the latency of a round of messages per server per client update, decrease throughput, and reduce availability in the face of replica failures.

Because we only care about the *set* of elements contained in the value array and not its order, we might be tempted to argue that the shopping cart application is eventually consistent when asynchronously updated and forego the coordination logic. Unfortunately, such informal reasoning can hide serious bugs. For example, consider what would happen if a delete action for an item arrived at some replica before any addition of that item: the delete would be ignored, leading to inconsistencies between replicas.

A happier story emerges from our analysis of the disorderly cart service. Figure 3.9 shows a visualization of the core logic of the disorderly cart module presented in Listing 3.9. This program is not complete: its inputs and outputs are channels rather than interfaces, so the dataflow from source to sink is not completed. To complete this program, we must mixin code that connects input and output interfaces to `action_msg`, `checkout_msg`, and `response_msg`, as the CartClient does (Listing 3.7). Note that the disorderly cart has points of order on all paths but there are no cycles.

Figure 3.10 shows the analysis for a complete implementation that mixes in both the client code and logic to replicate the `cart_action` table via best-effort multicast (see Listing 3.7 for the corresponding source code). Note that communication (via `action_msg`) between client and server—and among server replicas—crosses no points of order, so all the communication related to shopping actions converges to the same final state without coordination. However, there are

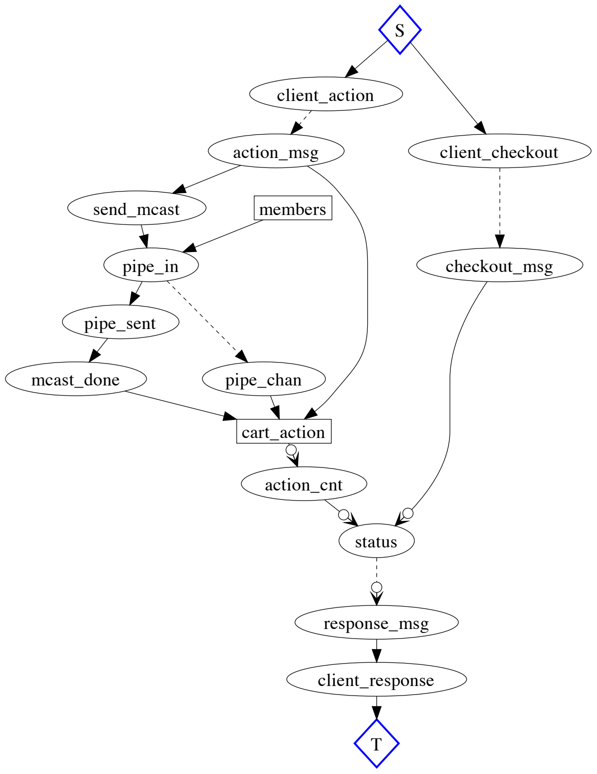


Figure 3.10: Visualization of the complete disorderly cart program.

points of order upon the appearance of `checkout_msg` messages, which must be joined with an `action_cnt` aggregate over the set of updates. Additionally, using the `accum` aggregate adds a point of order to the end of the dataflow, between `status` and `response_msg`. Although the accumulation of shopping actions is monotonic, summarization of the cart state requires us to ensure that there will be no further cart actions.

Comparing Figure 3.8 and Figure 3.10, we can see that the disorderly cart requires less coordination than the destructive cart: to ensure that the response to the client is deterministic and consistently replicated, we need to coordinate once per *session* (at checkout), rather than once per shopping action. This is analogous to the desired behavior in practice [82].

Discussion

Not all programs can be expressed using only monotonic logic, so adding some amount of coordination is often required to ensure consistency. In this running example we studied two candidate implementations of a simple distributed application with the aid of our program analysis. Both programs have points of order, but the analysis tool helped us reason about their relative coordination costs. Deciding that the disorderly approach is “better” required us to apply domain knowledge: checkout is a coarser-grained coordination point than cart actions and their replication. In Section 4.5, we will show how richer static analysis can convert the destructive cart (which is arguably easier to write and understand) into an implementation with coordination requirements equivalent to those of the disorderly cart.

By providing the programmer with a set of abstractions that are predominantly order-independent, Bloom encourages a style of programming that minimizes coordination requirements. But as we see in the destructive cart program, it is nonetheless possible to use Bloom to write code in an imperative, order-sensitive style. Our analysis tools provide assistance in this regard. Given a particular implementation with points of order, Bloom's dataflow analysis can help a developer iteratively refine their program—either to "push back" the points to as late as possible in the dataflow, as we did in this example, or to "localize" points of order by moving them to locations in the program's dataflow where the coordination can be implemented on individual nodes without communication.

3.4 Related Work

Dedalus

Deductive Databases and Updateable State

Many deductive database systems, including LDL [45] and Glue-Nail [58], admit procedural semantics to deal with updates using an assignment primitive. In contrast, languages proposed by Cleary and Liu [49, 114, 122] retain a purely logical interpretation by admitting temporal extensions into their syntax and interpreting assignment or update as a composite operation across timesteps [114] rather than as a primitive. We follow the approach of Datalog/UT [114] in that we use explicit time suffixes to enforce a stratification condition, but differ in several significant ways. First, we model persistence explicitly in our language, so that like updates, it is specified as a composite operation across timesteps. Partly as a result of this, we are able to enforce stricter constraints on the allowable time suffixes in rules: a program may only specify what deductions are visible in the current timestep, the immediate next timestep, and *some* future timestep, as opposed to the free use of intervals allowed in rules in Liu et al.

Our temporal approach to representing state change most closely resembles the Statelog language [68]. By contrast, our motivation is the logical specification and implementation of distributed systems, and our principal contribution is the use of time to model both local state change and communication over unreliable networks.

Lamport's TLA+ [105] is a language for specifying concurrent systems in terms of constraints over valuations of state and temporal logic that describes admissible transitions. While DEDALUS may be used as a specification language for the purposes of verifification, as we will see in Chapter 5, it differs from TLA+ in several key ways. While TLA+ is a full-blown temporal logic, DEDALUS—which provides only the small temporal vocabulary of *next* and *async*—is an executable programming language. DEDALUS's unified treatment of temporal and other attributes of facts; this enables the full literature of Datalog to be applied to both temporal and instantaneous properties of programs.

Distributed Systems

Significant recent work ([14, 32, 46, 119], etc.) has focused on applying deductive database languages to the problem of specifying and implementing network protocols and distributed systems. Loo et al. [119] proved that classes of programs with certain monotonicity properties (i.e., programs without negation or fact deletion) are equivalent (specifically, eventually consistent) when evaluated globally (via a single fixpoint computation) or in a distributed setting in which the *chain of fixpoints* interpretation is applied at each participating node, and no messages are lost. Navarro et al. [137] proposed an alternate syntax that addressed key ambiguities in Overlog, including the *event creation vs. effect* ambiguity. Their solution solves the problem by introducing procedural semantics to the interpretation of the augmentog programs. A similar analysis was offered by Mao [126].

Bloom

Systems with loose consistency requirements have been explored in depth by both the systems and database management communities (e.g., [66, 71, 75, 160]); we do not attempt to provide an exhaustive survey here. The shopping cart case study in Section 3.3 was motivated by the Amazon Dynamo paper [81], as well as the related discussion by Helland and Campbell [82].

The Bloom language is inspired by earlier work that attempts to integrate databases and programming languages. This includes early research such as Gem [176] and more recent object-relational mapping layers such as Ruby on Rails. Unlike these efforts, Bloom is targeted at the development of both distributed infrastructure and distributed applications, so it does not make any assumptions about the presence of a database system "underneath". Given our prototype implementation in Ruby, it is tempting to integrate Bud with Rails; we have left this for future work.

There is a long history of attempts to design programming languages more suitable to parallel and distributed systems; for example, Argus [113] and Linda [67]. Again, we do not hope to survey that literature here. More pragmatically, Erlang is an oft-cited choice for distributed programming in recent years. Erlang's features and design style encourage the use of asynchronous lightweight "actors." As mentioned previously, we did a simple Bloom prototype DSL in Erlang (which we cannot help but call "Bloomerlang"), and there is a natural correspondence between Bloom-style distributed rules and Erlang actors. However there is no requirement for Erlang programs to be written in the disorderly style of Bloom. It is not obvious that typical Erlang programs are significantly more amenable to a useful points-of-order analysis than programs written in any other functional language. For example, ordered lists are basic constructs in functional languages, and without program annotation or deeper analysis than we need to do in Bloom, any code that modifies lists would need be marked as a point of order, much like our destructive shopping cart. We believe that Bloom's "disorderly by default" style encourages order-independent programming, and we know that its roots in database theory helped produce a simple but useful program analysis technique. While we would be happy to see the analysis "ported" to other distributed programming environments, it may be that design patterns using Bloom-esque disorderly programming are the natural way to achieve this.

Our work on Bloom bears a resemblance to the Reactor language [61]. Both languages target distributed programming and are grounded in Datalog. Like many other rule languages including our earlier work on Overlog, Reactor updates mutable state in an operational step "outside Datalog" after each fixpoint computation. By contrast, Bloom is purely declarative: following Dedalus, it models updates as the logical derivation of immutable "versions" of collections over time. While Bloom uses a syntax inspired by object-oriented languages, Reactor takes a more explicitly agent-oriented approach. Reactor also includes synchronous coupling between agents as a primitive; we have opted to only include asynchronous communication as a language primitive and to provide synchronous coordination between nodes as a library.

Another recent language related to our work is Coherence [59], which also embraces "disorderly" programming. Unlike Bloom, Coherence is not targeted at distributed computing and is not based on logic programming.

3.5 Discussion

Dedalus is a disorderly language; like Overlog, it encourages a systems-as-queries programming style in which high-level rules describe the required relationships among data elements without specifying order of computation or data. This makes it easy to express programs that are insensitive to delivery and scheduling order—a valuable property in distributed environments, in which controlling these orders can incur prohibitive costs. Dedalus overcomes the expressivity issues of query languages like Overlog by allowing programmers to also express temporal constraints using inductive and asynchronous rules. Inductive rules make it possible to declaratively express *change* over time, and hence capture key patterns such as mutable state. Asynchronous rules capture the fundamental uncertainty of distributed systems, in which the effects of deductions can be lost or arbitrarily delayed. The principal contribution of Dedalus is the reification of logical time, not just into the syntax of the language but into its model-theoretic semantics, enabling a variety of program analyses that can guarantee program correctness despite widespread nondeterminism in executions.

Bloom showed that the clean semantic core of Dedalus can be exposed in a programmer-friendly language supporting familiar patterns for structuring and reuse. More importantly, it delivers on the promise of domain-specific program analyses for distributed systems by providing tools that check whether submitted programs are deterministic, leveraging the model-theoretic semantics of Dedalus. Monotonicity analysis provides concrete *assurances* of determinism; when it cannot do so, it provides *guidance* to assist programmers in repairing programs to ensure consistent outcomes.

Bloom's monotonicity analysis only scratches the surface of what can be built on this foundation. In the remainder of this thesis, we develop the complementary analysis tools Blazes and LDFI. Blazes, presented in Chapter 4, extends the guidance provided by monotonicity analysis into an automated framework for coordination selection and placement. LDFI (Chapter 5) exploits data lineage to find bugs caused by partial failures or to provide assurances that Dedalus programs are fault-tolerant.

Chapter 4

Blazes

The first principle of successful scalability is to batter the consistency mechanisms down to a minimum.
– James Hamilton, as transcribed in [35].

When your map or guidebook indicates one route, and the blazes show another, follow the blazes.
– Appalachian trail conservancy [2].

In the previous chapter, we saw how the monotonicity analysis enabled by Bloom allows programmers to identify whether their distributed programs are guaranteed to produce deterministic outcomes; when Bloom cannot provide this guarantee, it identifies nonmonotonic program statements as potential causes of non-determinism. It is tempting to ask whether appropriate tooling can take this a step further: if we can identify precisely *why* programs can exhibit bad behaviors that lead to inconsistent outcomes, can we automatically *repair* them to prevent those behaviors? Specifically, can we rewrite programs in order to guard potentially order-sensitive operations with order-preserving or order-restoring coordination mechanisms such as ordered delivery mechanisms and barriers?

In this chapter we present Blazes, a cross-language analysis framework that provides developers of distributed programs with judiciously chosen, application-specific coordination code. First, Blazes *analyzes* applications to identify code that may cause consistency anomalies. Blazes' analysis is based on a pattern of properties and composition: it begins with key properties of individual software components, including order-sensitivity, statefulness, and replication; it then reasons transitively about compositions of these properties across dataflows that span components. Second, Blazes automatically *generates* application-aware coordination code to prevent consistency anomalies with a minimum of coordination. The key intuition exploited by Blazes is that even when components are order-sensitive, it is often possible to avoid the cost of global ordering without sacrificing consistency. In many cases, Blazes can ensure consistent outcomes via a more efficient and manageable protocol of asynchronous point-to-point communication between producers and consumers—called *sealing*—that indicates when partitions of a stream have stopped changing. These partitions are identified and "chased" through a dataflow via techniques from functional dependency analysis, another surprising application of database theory to distributed consistency.

The Blazes architecture is depicted in Figure 4.1. Blazes can be directly applied to existing programming platforms based on distributed stream or dataflow processing, including Twitter Storm [110], Apache S4 [138], and Spark Streaming [174]. Programmers of stream processing engines interact with Blazes in a "grey box" manner: they provide simple semantic *annotations* to the black-box components in their dataflows, and Blazes performs the analysis of all dataflow paths through the program. Blazes can also take advantage of the richer analyzability of declarative languages like Bloom. Bloom programmers are freed from the need to supply annotations, since Bloom's language semantics allow Blazes to infer component properties automatically.

We make the following contributions in this chapter:

- **Consistency Anomalies and Properties.** We present a spectrum of consistency anomalies that arise in distributed dataflows. We identify key properties of both streams and components that affect consistency.

- **Composition of Properties.** We show how to analyze the composition of consistency properties in complex programs via a term-rewriting technique over dataflow paths, which translates local component properties into end-to-end stream properties.

- **Custom Coordination Code.** We distinguish two alternative coordination strategies, *ordering* and *sealing*, and show how we can automatically generate application-aware coordination code that uses the cheaper sealing technique in many cases.

We conclude by evaluating the performance benefits offered by using Blazes as an alternative to generic, order-based coordination mechanisms available in both Storm and Bloom.

Running Examples

We consider two running examples: a streaming analytic query implemented using the Storm stream processing system and an ad-tracking network implemented using the Bloom distributed programming language.

Streaming analytics with Storm: Figure 4.2 shows the architecture of a Storm topology that computes a windowed word count over the Twitter stream. Each "tweet" is associated with a numbered batch (the unit of replay for replay-based fault-tolerance, which we describe below) and is sent to exactly one `Splitter` component via random partitioning. The `Splitter` component divides tweets into their constituent words. These are hash partitioned to the `Count` component, which tallies the number of occurrences of each word in the current batch. When a batch ends, the `Commit` component records the batch number and frequency for each word in the batch in a backing store.

For this use case, we are not concerned with the consistency of replicated state, but with ensuring that accurate counts are committed to the store despite the at-least-once delivery semantics.

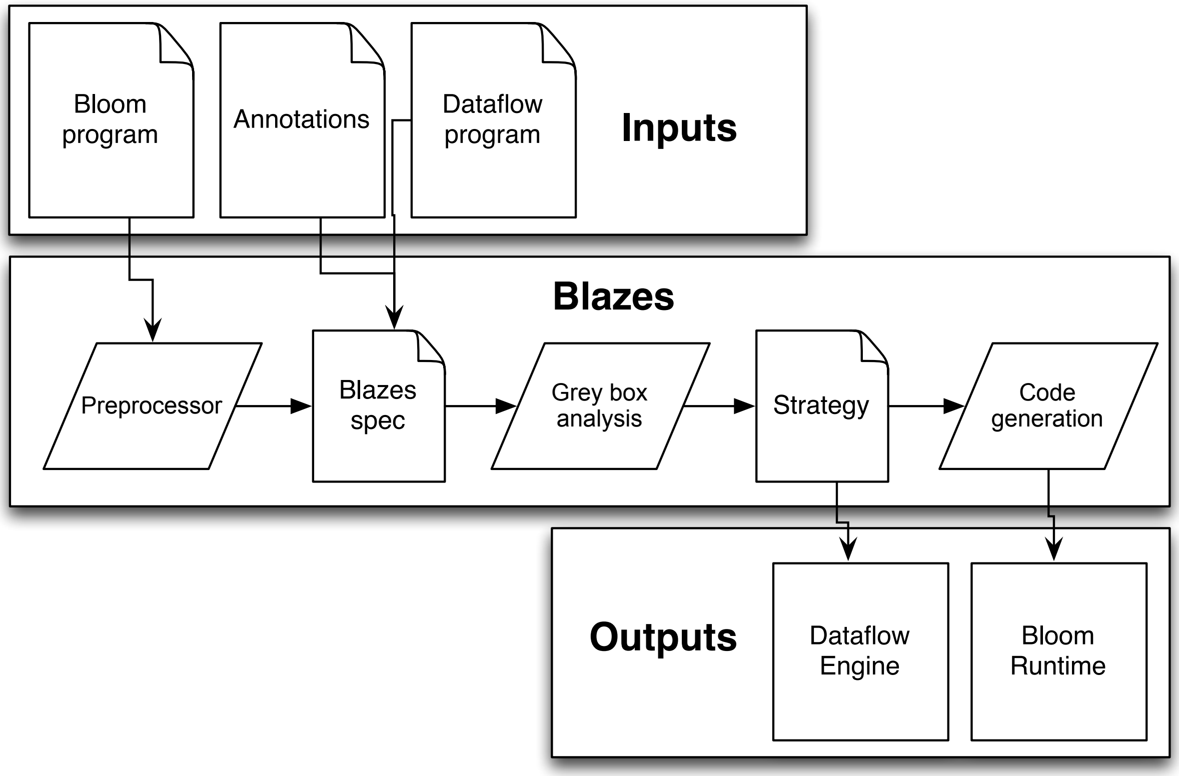


Figure 4.1: The BLAZES framework. In the "grey box" system, programmers supply a configuration file representing an annotated dataflow. In the "white box" system, this file is automatically generated via static analysis.

Storm ensures fault-tolerance via replay: if component instances fail or time out, stream sources redeliver their inputs. As a result, messages may be delivered more than once. When implementing a Storm topology, the programmer must decide whether to make it *transactional*: a transactional topology processes tuples in atomic batches, ensuring that certain components (called *committers*) emit the batches in a total order. The Storm documentation describes how programmers may, by recording the last successfully processed batch identifier at the time of each commit, ensure exactly-once processing in the face of possible replay by incurring the extra overhead of synchronized batch processing [110].

Note that batches are independent; because the streaming query groups outputs by batch id, there is no need to order batches with respect to each other. As we shall see, BLAZES can aid a topology designer in avoiding unnecessary ordering constraints, which (as we will see in Section 4.7) can result in a 3× improvement in throughput.

Ad-tracking with Bloom: Figure 4.3 depicts an ad-tracking network, in which a collection of *ad servers* deliver ads to users (not shown) and send click logs (edges labeled "*c*") to a set of

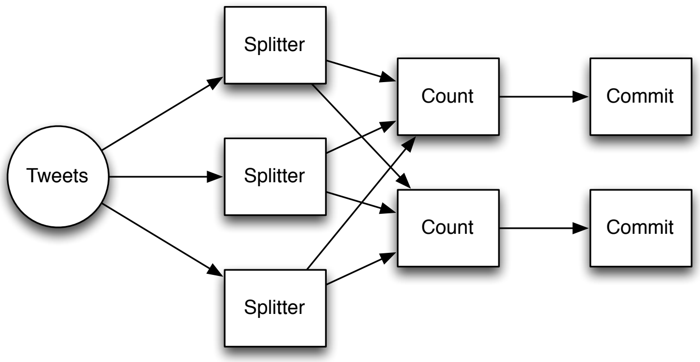


Figure 4.2: Physical architecture of a Storm word count topology

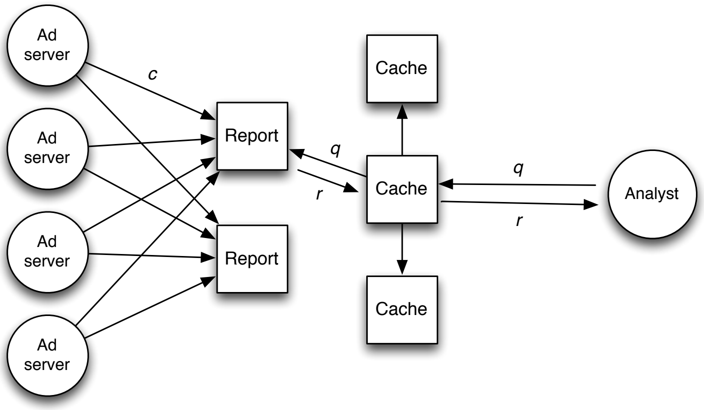


Figure 4.3: Physical architecture of an ad-tracking network

reporting servers. Reporting servers compute a continuous query; analysts make requests ("*q*") for subsets of the query answer (e.g., by visiting a "dashboard") and receive results via the stream labeled "*r*". To improve response times for frequently-posed queries, a caching tier is interposed between analysts and reporting servers. An analyst poses a request about a particular ad to a cache server. If the cache contains an answer for the query, it returns the answer directly. Otherwise, it forwards the request to a reporting server; when a response is received, the cache updates its local state and returns a response to the analyst. Asynchronously, it also sends the response to the other caches. The clickstream *c* is sent to all reporting servers; this improves fault tolerance and reduces query latency, because caches can contact any reporting server. Due to failure, retry and the interleaving of messages from multiple sources, network delivery order is non-deterministic. As we shall see, different continuous queries have different sensitivities to network non-determinism. Blazes will help determine exactly when coordination is required to ensure that network behavior does not cause inconsistent results, and when possible will choose a coordination mechanism that minimizes latency, throughput and availability penalties.

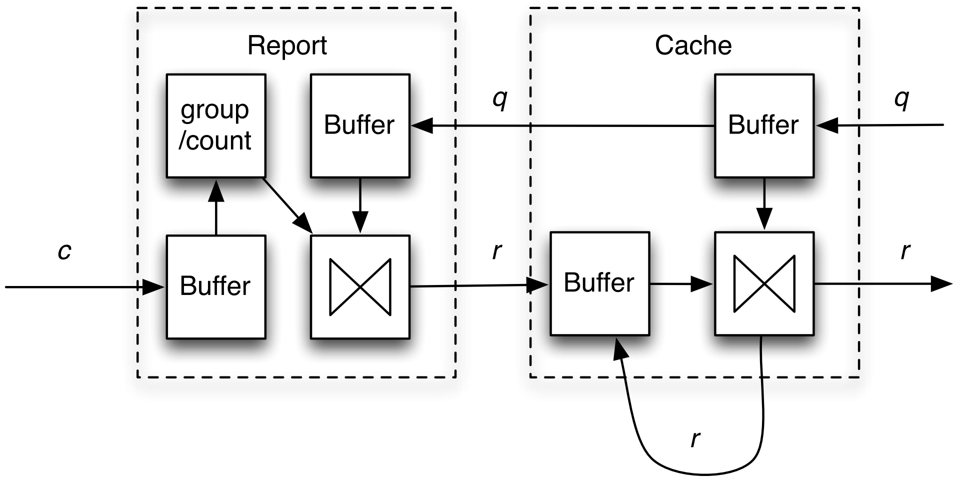


Figure 4.4: Dataflow representations of the ad-tracking network's `Report` and `Cache` components.

4.1 System Model

The Blazes API is based on a simple "black box" model of component-based distributed services. We use dataflow graphs [92] to represent distributed services: nodes in the graph correspond to service components, which expose input and output *interfaces* that correspond to service calls or other message events. While we focus our discussion on streaming analytics systems, we can represent both the data- and control-flow of arbitrary distributed systems using this dataflow model. Dataflow makes an attractive model of distributed computation not only because of its generality, but because of its *scale-invariance*. For example, Figure 4.4 represents the ad-tracking system at two levels of granularity: a dataflow of components and a dataflow of relational operators.

A *logical dataflow* (e.g., the representation of the ad tracking network depicted in Figure 4.4) captures a *software architecture*, describing how components interact via API calls. By contrast, a *physical dataflow* (like the one shown in Figure 4.3) extends a software architecture into a *system architecture*, mapping software components to the physical resources on which they will execute. We choose to focus our analysis on logical dataflows, which abstract away details like the multiplicity of physical resources but are sufficient—when properly annotated—to characterize the consistency semantics of distributed services.

Components and Streams

A *component* is a logical unit of computation and storage, processing streams of inputs and producing streams of outputs over time. Components are connected by *streams*, which are unbounded, unordered [112] collections of messages. A stream associates an output interface of one component with an input interface of another. To reason about the behavior of a component, we consider all the *paths* that connect its inputs and outputs. For example, the reporting server (`Report` in

Figure 4.4) has two input streams, and hence defines two possible dataflow paths (from *c* to *r* and from *q* to *r*). We assume that components are deterministic: two instances of a given component that receive the same inputs in the same order produce the same outputs and reach the same state.

A *component instance* is a binding between a component and a physical resource—with its own clock and (potentially mutable) state—on which the component executes. In the ad system, the reporting server is a single logical component in Figure 4.4, but corresponds to two distinct (replicated) component instances in Figure 4.3. Similarly, we differentiate between logical streams (which characterize the types of the messages that flow between components) and *stream instances*, which correspond to physical channels between component instances. Individual components may execute on different machines as separate component instances, consuming stream instances with potentially different contents and orderings.

While streams are unbounded, in practice they are often divided into batches [110, 174, 30] to enable replay-based fault-tolerance. *Runs* are (possibly repeated) executions over finite stream batches.

A stream producer can optionally embed *punctuations* [164] into the stream. A punctuation guarantees that the producer will generate no more messages within a particular logical partition of the stream. For example, in Figure 4.3, an ad server might indicate that it will henceforth produce no new records for a particular time window or advertising campaign via the *c* stream. In Section 4.4, we show how punctuations can enable efficient, localized coordination strategies based on *sealing*.

4.2 Dataflow Consistency

In this section, we develop consistency criteria and mechanisms appropriate to distributed, fault-tolerant dataflows. We begin by describing undesirable behaviors that can arise due to the interaction between nondeterministic message orders and fault-tolerance mechanisms. We review common strategies for preventing such anomalies by exploiting semantic properties of components (Section 4.2) or by enforcing constraints on message delivery (Section 4.2). We then generalize delivery mechanisms into two classes: message *ordering* and partition *sealing*. To provide intuition, we consider a collection of queries that we could install at the reporting server in the ad tracking example presented in Section 4. We show how slight differences in the queries can lead to different distributed anomalies, and how practical variants of the ordering and sealing strategies can be used to prevent these anomalies.

Anomalies

Nondeterministic messaging interacts with fault-tolerance mechanisms in subtle ways. Two standard schemes exist for fault-tolerant dataflows: *replication* (used in the ad reporting system described in Section 4) and *replay* (employed by Storm and Spark) [30]. Both mechanisms allow systems to recover from the failure of components or the loss of messages via *redundancy* of state and computation. Unfortunately, redundancy brings with it a need to consider issues of *consis-*

Strategy	**Prevents**				**Required Coordination**
	Nondeterministic order	*Cross-run nondeterminism*	*Cross-instance nondeterminism*	*Replica divergence*	
Delivery mechanisms					
Sequencing	**X**	**X**	**X**	**X**	Global
Dynamic ordering			**X**	**X**	Global
Component Properties					
Confluent		**X**	**X**	**X**	None
Convergent				**X**	None
Hybrid					
Sealing		**X**	**X**	**X**	Local

Table 4.1: The relationship between potential stream anomalies and remediation strategies based on component properties and delivery mechanisms. For each strategy, we enumerate the stream anomalies that it prevents, and the level of coordination it requires. For example, convergent components prevent replica divergence but not cross-instance nondeterminism, while a dynamic message ordering mechanism prevents all replication anomalies (but requires coordination to establish a global ordering among operations).

tency, because nondeterministic message orders can lead to disagreement regarding stream contents among replicas or across replays. This disagreement undermines the transparency that fault tolerance mechanisms are meant to achieve, giving rise to to stream *anomalies* that are difficult to debug.

Because it is difficult to control the order in which a component's inputs appear, the first (and least severe) anomaly is nondeterministic orderings of (otherwise deterministic) output contents. In this paper, we focus on the three remaining classes of anomalies, all of which have direct consequences on the fault-tolerance mechanism:

1. *Cross-run nondeterminism*, in which a component produces different output stream *contents* in different runs over the same inputs. Systems that exhibit cross-run nondeterminism do not support replay-based fault-tolerance. For obvious reasons, nondeterminism across runs makes such systems difficult to test and debug.

2. *Cross-instance nondeterminism* , in which replicated instances of the same components produce different output contents in the *same* execution over the same inputs. Cross-instance nondeterminism can lead to inconsistencies across queries.

3. *Replica divergence*, in which the state of multiple replicas becomes permanently inconsistent. Some services may tolerate transient disagreement between streams (e.g., for streams corresponding to the results of read-only queries), but permanent replica divergence is never desirable.

Name	*Continuous Query*
THRESH	`select id from clicks group by id having count(*) > 1000`
POOR	`select id from clicks group by id having count(*) < 100`
WINDOW	`select window, id from clicks group by window, id having count(*) < 100`
CAMPAIGN	`select campaign, id from clicks group by campaign, id having count(*) < 100`

Figure 4.5: Reporting server queries (shown in SQL syntax for familiarity).

Table 4.1 captures the relationship between these anomalies and the various delivery mechanisms and component properties that can be exploited to prevent them.

Component properties: confluence and convergence

Distributed programs that produce the same outcome for all message delivery orders exhibit none of the anomalies listed in Section 4.2, regardless of the choice of fault-tolerance or delivery mechanisms. The CALM Theorem, discussed in Section 3.2, establishes that programs expressed in *monotonic* logic produce deterministic results despite nondeterminism in delivery orders [18, 83, 23]. Intuitively, monotonic programs compute a continually growing result, never retracting an earlier output given new inputs. Hence replicas running monotonic code always eventually agree, and replaying monotonic code produces the same result in every run.

We call a dataflow component *confluent* if it produces the same *set* of outputs for all *orderings* of its inputs. At any time, the output of a confluent component (and any redundant copies of that component) is a subset of the unique, "final" output. Confluent components never exhibit any of the three dataflow anomalies listed above. Confluence is a property of the behavior of components—monotonicity (a property of program logic) is a sufficient condition for confluence.

Confluence is similar to the notion of replica *convergence* common in distributed systems. A system is convergent or "eventually consistent" if, when all messages have been delivered, all replicas agree on the set of stored values [166]. Convergent components never exhibit replica divergence. Convergence is a local guarantee about component *state*; by contrast, confluence provides guarantees about component *outputs*, which (because they become the inputs to downstream components) compose into global guarantees about dataflows.

Confluence implies convergence but the converse does not hold. Convergent replicated components are guaranteed to eventually reach the same state, but this final state may not be uniquely determined by component inputs. As Figure 4.1 indicates, convergent components allow cross-instance nondeterminism, which can occur when reading "snapshots" of the convergent state while it is still changing. Consider what happens when the read-only outputs of a convergent component (e.g., GETs posed to a key/value store) flow into a replicated stateful component (e.g., a replicated cache). If the caches record different stream contents, the result is replica divergence.

Coordination Strategies: ordering and sealing

Confluent components produce deterministic outputs and convergent replicated state. How can we achieve these properties for components that are not confluent? We assume that components are deterministic, so we can prevent inconsistent outputs within or across program runs simply by removing the nondeterminism from component input orderings. Two extreme approaches include (a) establishing a single total order in which all instances of a given component receive messages (a *sequencing* strategy) and (b) disallowing components from producing outputs until all of their inputs have arrived (a *sealing* strategy). The former—which enforces a total order of inputs—resembles state machine replication from the distributed systems literature [148], a technique for implementing consistent replicated services. The latter—which instead controls the order of evaluation at a coarse grain—resembles stratified evaluation of logic programs [165] in the database literature, as well as barrier synchronization mechanisms used in systems like MapReduce [57].

Both strategies lead to "eventually consistent" program outcomes—if we wait long enough, we get a unique output for a given input. Unfortunately, neither leads directly to a practical coordination implementation. We cannot in general preordain a total order over all messages to be respected in all executions. Nor can we wait for streams to stop producing inputs, as streams are unbounded.

Fortunately, both coordination strategies have a dynamic variant that allows systems to make incremental progress over time. To prevent replica divergence, it is sufficient to use a dynamic ordering service (e.g., Paxos) that decides a global order of messages *within a particular run*. As Figure 4.1 shows, a nondeterministic choice of message ordering can prevent cross-instance nondeterminism but not cross-run nondeterminism since the choice is dependent on arrival orders at the coordination service. Similarly, strategies based on sealing inputs can be applied to infinite streams as long as the streams can be partitioned into finite partitions that exhibit temporal locality, like windows with "slack" [8]. Sealing strategies—applicable when input stream partitioning is *compatible* with component semantics—can rule out all nondeterminism anomalies. The notion of compatibility is defined precisely in Section 4.4. Informally, a punctuated input stream is compatible with a sealable component when the partitioning of the stream (e.g., the variant of the ad-tracking network in which click logs are partitioned by *campaign_id*) corresponds with the partitioning of the component (e.g., the reporting query `CAMPAIGN`, which groups by *campaign_id*). Note that sealing is significantly less constrained than ordering: it enforces an output barrier per partition, but allows asynchrony both in the arrival of a batch's inputs and in interleaving across batches.

Example Queries

The ad reporting system presented in Section 4 involves a collection of components interacting in a dataflow graph. In this section, we focus on the `Report` component, which accumulates click logs and continually evaluates a standing query against them. Figure 4.5 presents a variety of simple queries that we might install at the reporting server; perhaps surprisingly, these queries have substantially different coordination requirements if we demand that they return deterministic answers.

We consider first a threshold query *THRESH*, which computes the unique identifiers of any ads that have at least 1000 impressions. *THRESH* is confluent: we expect it to produce a deterministic result set without need for coordination, since the value of the count monotonically increases in a manner insensitive to message arrival order [52].

By contrast, consider a "poor performers" query: *POOR* returns the IDs of ads that have fewer than one hundred clicks (this might be used to recommend such ads for removal from subsequent campaigns). *POOR* is nonmonotonic: as more clicks are observed, the set of poorly performing ads might shrink—and because it ranges over the entire clickstream, we would have to wait until there were no more log messages to ensure a unique query answer. Allowing *POOR* to emit results "early" based on a nondeterministic event, like a timer or request arrival, is potentially dangerous; multiple reporting server replicas could report different answers in the same execution. To avoid such anomalies, replicas could remain in sync by coordinating to enforce a global message delivery order. Unfortunately, this approach incurs significant latency and availability costs.

In practice, streaming query systems often address the problem of blocking operators via *windowing*, which constrains blocking queries to operate over bounded inputs [8, 24, 41]. If the poor performers threshold test is *scoped* to apply only to individual windows (e.g., by including the window name in the grouping clause), then ensuring deterministic results is simply a matter of blocking until there are no more log messages *for that window*. Query *WINDOW* returns, for each one hour window, those advertisement identifiers that have fewer than 100 clicks within that window.

The windowing strategy is a special case of the more general technique of *sealing*, which may also be applied to partitions that are not explicitly temporal. For example, it is common practice to associate a collection of ads with a "campaign," or a grouping of advertisements with a similar theme. Campaigns may have different lengths, and may overlap or contain other campaigns. Nevertheless, given a punctuation indicating the termination of a campaign, the nonmonotonic query *CAMPAIGN* can produce deterministic outputs.

4.3 Annotated Dataflow Graphs

So far, we have focused on the consistency anomalies that can affect the outputs of individual "black box" components. In this section, we extend our discussion in two ways. First, we propose a *grey box* model in which programmers provide simple annotations about the semantic properties of components. Second, we show how BLAZES can use these annotations to automatically derive the consistency properties of entire dataflow graphs.

Annotations and Labels

In this section, we describe a language of *annotations* and *labels* that enriches the "black box" model (Section 4.1) with additional semantic information. Programmers supply annotations about

Severity	*Label*	*Confluent*	*Stateless*
1	CR	**X**	**X**
2	CW	**X**	
3	OR_{gate}		**X**
4	OW_{gate}		

Table 4.2: The **C.R.O.W.** component annotations. A component path is either **C**onfluent or **O**rder-sensitive, and either changes component state (a **W**rite path) or does not (a **R**ead-only path). Component paths with higher *severity* annotations can produce more stream anomalies.

Severity	*Label*	*Nondeterministic order*	*Cross-run nondeterminism*	*Cross-instance nondeterminism*	*Replica divergence*
0	**NDRead**$_{gate}$	**X**	**X**		
0	**Taint**	**X**	**X**		
1	**Seal**$_{key}$	**X**			
2	**Async**	**X**			
3	**Run**	**X**	**X**		
4	**Inst**	**X**	**X**	**X**	
5	**Split**	**X**	**X**	**X**	**X**

Table 4.3: Stream labels, ranked by severity. **NDRead**$_{gate}$ and **Taint** are internal labels, used by the analysis system but never output. **Run, Inst** and **Split** correspond to the stream anomalies enumerated in Section 4.2: cross-run non-determinism, cross-instance non-determinism and split brain, respectively.

paths through components and about input streams; using this information, Blazes derives labels for each component's output streams.

Component Annotations

Blazes provides a small, intuitive set of annotations that capture component properties relevant to stream consistency. A review of the implementation or analysis of a component's input/output behavior should be sufficient to choose an appropriate annotation. Figure 4.2 lists the component annotations supported by Blazes. Each annotation applies to a path from an input interface to an output interface; if a component has multiple input or output interfaces, each path can have a different annotation.

The CR annotation indicates that a path through a component is confluent and stateless; that is, it produces deterministic output regardless of its input order, and calls to the input interface do not modify the component's state. CW denotes a path that is confluent and stateful.

The annotations OR_{gate} and OW_{gate} denote non-confluent paths that are stateless or stateful, respectively. The *gate* subscript is a set of attribute names that indicates the partitions of the input streams over which the non-confluent component operates. This annotation allows BLAZES to determine whether an input stream containing end-of-partition punctuations can produce deterministic executions without using global coordination. Supplying *gate* is optional; if the programmer does not know the partitions over which the component path operates, the annotations OR_* and OW_* indicate that each record belongs to a different partition.

Consider a reporting server component implementing the query *WINDOW*. When it receives a request referencing a particular advertisement and window, it returns a response if the advertisement has fewer than 1000 clicks *within that window*. We would label the path from request inputs to outputs as $OR_{ad,window}$—a stateless non-confluent path operating over partitions with composite key *ad,window*. Requests do not affect the internal state of the component, but they do return potentially non-deterministic results that depend on the outcomes of races between queries and click records (assuming the inputs are non-deterministically ordered). Note however that if we were to delay the results of queries until we were certain that there would be no new records for a particular advertisement *or* a particular window,[1] the output would be deterministic. Hence *WINDOW* is "compatible" with click streams partitioned (and emitting appropriate punctuations) on *ad* or *window*—this notion of compatibility will be made precise in Section 4.4.

Stream Annotations

Programmers can also supply optional annotations to describe the semantics of streams. The $\textbf{Seal}_{key}$ annotation means that the stream is *punctuated* on the subset *key* of the stream's attributes—that is, the stream contains punctuations on *key*, and there is at least one punctuation corresponding to every stream record. For example, a stream representing messages between a client and server might have the label $\textbf{Seal}_{client,session}$, to indicate that clients will send messages indicating that sessions are complete. To ensure progress, there must be a punctuation for every session identifier; because punctuations may arrive before all of the messages that it seals, punctuations must contain a manifest of the partition contents.

Programmers can use the **Rep** annotation to indicate that a stream is *replicated*. Replicated streams have the following properties:

1. A replicated stream connects a producer component instance (or instances) to more than one consumer component instance.

2. A replicated stream produces the same contents for all stream instances (unlike, for example, a partitioned stream).

The **Rep** annotation carries semantic information both about expected execution *topology* and *programmer intent*, which BLAZES uses to determine when non-deterministic stream contents can lead to replica disagreement. **Rep** is an optional boolean flag that may be combined with other annotations and labels.

[1]This rules out races by ensuring that the query comes *after* all relevant click records.

Derived Stream Labels

Given an annotated component with labeled input streams, Blazes derives a label for each of its output streams. Figure 4.3 lists the derived stream labels, each of which corresponds to a class of anomalies that may occur in a given stream instance. As we saw in Figure 4.3, anomalies represent a natural hierarchy in which each class contains those of lower severity.

The label **Async** corresponds to streams with deterministic contents whose order may differ in different executions or on different stream instances. **Async** is conservatively applied as the default label; in general, we assume that communication between components is asynchronous.

Streams labeled **Run** may exhibit cross-run non-determinism, having different contents in different runs. Those labeled **Inst** may also exhibit cross-instance non-determinism on different replicas within a single run. Finally, streams labeled **Split** may have split-brain behaviors (persistent replica divergence).

4.4 Coordination Analysis and Synthesis

Blazes uses component and stream annotations to determine if a given dataflow is guaranteed to produce deterministic outcomes; if it cannot make this guarantee, it augments the program with coordination code. In this section, we describe the program analysis and synthesis process.

Analysis

To derive labels for the output streams in a dataflow graph, Blazes starts by enumerating all paths between pairs of sources and sinks. To rule out infinite paths, it reduces each cycle in the graph to a single node with a collapsed label by selecting the label of highest severity among the cycle members. Note that in the ad-tracking network dataflow shown in Figure 4.4, `Cache` participates in a cycle (the self-edge, corresponding to communication with other cache instances), but `Cache` and `Report` form no cycle, because `Cache` provides no path from r to q.

For each component whose input streams are defined (beginning with the components with unconnected inputs), Blazes first performs an *inference* step, shown in Figure 4.6, for every path through the component. When inference is complete, each of the output interfaces of the component is associated with a set of derived stream labels. Each output interface will have at least one label for each an input interface from which the output interface is reachable, and may have other labels introduced by the inference rules.

Blazes then performs the second analysis step, the *reconciliation* procedure (described in Figure 4.7), which may add additional labels. Finally, the labels for each output interface are merged into a single label. This output stream becomes an input stream of the next component in the dataflow, and so on.

Transitivity of seals

When input streams are sealed, the inference and reconciliation procedures test whether the seal keys are compatible with the annotations of the component paths into which they flow.

Sealed streams can enable efficient, localized coordination strategies when the sealed partitions are *independent*—a property not just of the streams themselves but of the components that process them. To recognize when sealed input streams are compatible with the component paths into which they flow, we need to compare seal keys with the partitions over which non-confluent operations range.

Intuitively, the stream partitioning matches the component partitioning if at least one of the attributes in *gate* is *injectively* determined by all of the attributes in *key*. For example, a company name may functionally determine their stock symbol and the location of their headquarters; when the company name Yahoo! is "sealed" (a promise is given that there will be no more records with that company name) their stock symbol `YHOO` is implicitly sealed as well, but the city of Sunnyvale is not, since other companies may have their headquarters there. A trivial (and ubiquitous) example of an injective function between input and output attributes is the identity function, which is applied whenever we project an attribute without transformation—we will focus our discussion on this example.

We define the predicate *injectivefd*(A, B), which holds for attribute sets A and B if $A \mapsto B$ (A functionally determines B) via some injective (distinctness-preserving) function. Such functions preserve the property of sealing: if we have seen all of the As, then we have also seen all the $f(A)$ for some injective f.

We may now define the predicate *compatible*:

$$\text{compatible(partition, seal)} \equiv \exists\, attr \subseteq \text{partition} \;.\; \text{injectivefd}(\text{seal}, attr)$$

The compatible predicate will allow the *inference* and *reconciliation* procedures to test whether a sealed input stream matches the implicit partitioning of a component path annotated OW_{gate} or OR_{gate}.

For example, given the queries in Figure 4.5, an input stream sealed on *campaign* is compatible with the query *CAMPAIGN*. All other queries combine the results from multiple campaigns into their answer, and may produce different outputs given different message and punctuation orderings.

In the remainder of this section we describe the *inference* and *reconciliation* procedures in detail.

Inference

At each inference step, we apply the rules in Figure 4.6 to derive additional intermediate stream labels for a component path. An intermediate stream label may be any of the labels in Figure 4.3.

Rules 1 and 2 of Figure 4.6 reflect the consequences of providing non-deterministically ordered inputs to order-sensitive components. **Taint** indicates that the internal state of the component may become corrupted by unordered inputs. **NDRead**$_{gate}$ indicates that the output stream may have transient non-deterministic contents. Rule 3 captures the interaction between cross-instance

$$(1)\ \frac{\{\textbf{Async, Run}\} \qquad \mathrm{OR}_{gate}}{\textbf{NDRead}_{gate}}$$

$$(2)\ \frac{\{\textbf{Async, Run}\} \qquad \mathrm{OW}_{gate}}{\textbf{Taint}} \qquad\qquad (3)\ \frac{\textbf{Inst} \qquad \mathrm{CW}, \mathrm{OW}_{gate}}{\textbf{Taint}}$$

$$(4)\ \frac{\textbf{Seal}_{key} \qquad \mathrm{OW}_{gate} \qquad \neg\ \mathrm{compatible}(gate, key)}{\textbf{Taint}}$$

Figure 4.6: Inference rules for component paths. Each rule takes an input stream label and a component annotation, and produces a new (internal) stream label. Rules may be read as implications: if the premises (expressions above the line) hold, then the conclusion (below) should be added to the `Labels` list.

non-determinism and split brain: transient disagreement among replicated streams can lead to permanent replica divergence if the streams modify component state downstream. Rules 4 tests whether the predicate *compatible* (defined in the previous section) holds, in order to determine when sealed input streams are compatible with stateful, non-confluent components.

When *inference* completes, each output interface of the component is associated with a set `Labels` of stream labels, containing all input stream labels as well as any intermediate labels derived by inference rules.

Reconciliation

Given an output interface associated with a set of labels, BLAZES derives additional labels using the *reconciliation* procedure shown in Figure 4.7, removing any intermediate labels that were produced by the inference step.

If the *inference* procedure has already determined that component state is tainted, then the output stream may exhibit split brain (if the component is replicated) and cross-run non-determinism. If $NDRead_{gate}$ (for some partition key *gate*) is among the stream labels, the output interface may have non-deterministic contents given non-deterministic input orders or interleavings with other component inputs, unless *all* streams with which it can "rendezvous" are sealed on a compatible key. If the component is replicated, non-deterministic outputs can lead to cross-instance non-determinism.

Once the internal labels have been dealt with, BLAZES returns the label in `Labels` of highest severity.

Notation

When describing trees of inferences and reconciliations used to derive output stream labels, we will use the following notation:

$$\begin{aligned}\text{protected}(\mathbf{gate}) \equiv \forall l \in \texttt{Labels}\ \ (l = \mathbf{NDRead}_{gate} \vee \\ \exists key\ (l = \mathbf{Seal}_{key} \wedge \text{compatible}(gate, key)))\end{aligned}$$

$$\frac{\mathbf{Taint} \in \texttt{Labels}}{Rep\ ?\ \mathbf{Split} : \mathbf{Run}}$$

$$\frac{\exists gate(\mathbf{NDRead}_{gate} \in \texttt{Labels} \qquad \neg\text{protected}(\mathbf{gate}))}{Rep\ ?\ \mathbf{Inst} : \mathbf{Run}}$$

Figure 4.7: The *reconciliation* procedure applies the rules above to the set `Labels`, possibly adding additional labels. "*Rep* ? **A** : **B**" means 'if *Rep*, add **A** to `Labels`, otherwise add **B**.' A set of keys *gate* is "protected" (with respect to a set *Labels* of stream labels) if and only if all labels are either read-only or are sealed on compatible keys. After *reconciliation*, output interfaces are associated with the element in `Labels` with highest severity.

$$\mathrm{CN}_1\ \frac{(\mathrm{R}_1)\,\dfrac{\mathrm{SL}_1 \qquad \mathrm{CA}_1}{\mathrm{SL}_2} \qquad (\mathrm{R}_2)\,\dfrac{\mathrm{SL}_3 \qquad \mathrm{CA}_2}{\mathrm{SL}_4} \qquad [\ldots]}{\mathrm{SL}_5}$$

Here the *SL* are stream labels, the *CA* are component annotations, *R* is the inference rule applied, and *CN* is the component name whose outputs are combined by the reconciliation procedure.[2] SL_2 and SL_4 are different labels for the same output interface. If no inference rules apply, we show the preservation of input stream labels by applying a default rule labeled "(p)."

Coordination Selection

In this section we describe practical coordination strategies for dataflows that are not confluent or convergent. Blazes will automatically repair such dataflows by constraining how messages are delivered to individual components. When possible, Blazes will recognize the compatibility between sealed streams and component semantics, synthesizing a seal-based strategy that avoids global coordination. Otherwise, it will enforce a total order on message delivery.

Sealing Strategies

If the programmer has provided a seal annotation $\mathbf{Seal}_{key}$ that is compatible with the (non-confluent) component annotation, we may use a synchronization strategy that avoids global coordination. Consider a component representing a reporting server executing the query *WINDOW* from Section 4. Its label is $OR_{id,window}$. We know that *WINDOW* will produce deterministic output contents

[2]For ease of exposition, we only consider cases where a component has a single output interface (as do all of our example components).

if we delay its execution until we have accumulated a complete, immutable partition to process (for each value of the *window* attribute). Thus a satisfactory protocol must allow stream producers to communicate when a stream partition is sealed and what it contains, so that consumers can determine when the complete contents of a partition are known.

To determine that the complete partition contents are available, the consumer must a) participate in a protocol with each producer to ensure that the local per-producer partition is complete, and b) perform a unanimous voting protocol to ensure that it has received partition data from each producer. Note that the voting protocol is a local form of coordination, limited to the "stakeholders" contributing to individual partitions. When there is only one producer instance per partition, Blazes need not synthesize a voting protocol.

Once the consumer has determined that the contents of a partition are immutable, it may process the partition without any further synchronization.

Ordering Strategies

If sealing strategies are not available, Blazes achieves convergence for replicated, non-confluent components by using an ordering service to ensure that all replicas process state-modifying events in the same order. Our current prototype uses a totally ordered messaging service based on Zookeeper for Bloom programs; for Storm, we use Storm's built-in support for "transactional" topologies, which enforces a total order over commits.

4.5 Case studies

In this section, we apply Blazes to the examples introduced in Section 4. We describe how programmers can manually annotate dataflow components. We then discuss how Blazes identifies the coordination requirements and, where relevant, the appropriate locations in these programs for coordination placement. In the next section we will consider how Blazes chooses safe yet relatively inexpensive coordination protocols. Finally, in Section 4.7 we show concrete performance benefits of the Blazes coordination choices as compared to a conservative use of a coordination service like Zookeeper.

We implemented the Storm wordcount dataflow, which consists of three "bolts" (components) and two distinct "spouts" (stream sources, which differ for the coordinated and uncoordinated implementations) in roughly 400 lines of Java. We extracted the dataflow metadata from Storm into Blazes via a reusable adapter; we describe below the output that Blazes produced and the annotations we added manually.

We implemented the ad reporting system entirely in Bloom, in roughly 125 LOC. The "destructive" Bloom shopping cart is presented in its entirety in Section 3.3. As we discuss in Section 4.6, Blazes automatically extracts all the relevant annotations for programs written in Bloom.

For each dataflow, we present excerpts from the Blazes configuration file, containing the programmer-supplied annotations.

Storm wordcount

We first consider the Storm distributed wordcount query. Given proper dataflow annotations, Blazes indicates that global ordering of computation on different components is unnecessary to ensure deterministic replay, and hence consistent outcomes.

Component annotations

The word counting topology comprises three components. Component `Splitter` is responsible for tokenizing each tweet into a list of terms—since it is stateless and insensitive to the order of its inputs, we give it the annotation CR. The `Count` component counts the number of occurrences of each word in each batch. We annotate it $OW_{word,batch}$—it is order-sensitive AND stateful (accumulating counts over time), but potentially sealable on word or batch (or both). Lastly, `Commit` writes the final counts to the backing store. `Commit` is also stateful (the backing store is persistent), but since it is append-only and does not record the order of appends, we annotate it CW.

```
Splitter:
  annotation:
    - { from: tweets, to: words, label: C }
Count:
  annotation:
    - { from: words, to: counts, label: OW,
      subscript: [word, batch] }
Commit:
  annotation: { from: counts, to: db, label: CW }
```

Analysis

Blazes performs the following reduction in the absence of any seal annotation:

$$\texttt{Committer}\,\frac{(\mathrm{p})\,\dfrac{\texttt{Count}\,\dfrac{(2)\,\dfrac{\texttt{Splitter}\,\dfrac{(\mathrm{p})\,\dfrac{\mathbf{Async}\quad \mathrm{CR}}{\mathbf{Async}}}{\mathbf{Async}}\quad \mathrm{OW}_{word,batch}}{\mathrm{Taint}}}{\mathbf{Run}}\quad \mathrm{CW}}{\mathbf{Run}}}{\mathbf{Run}}$$

Without coordination, non-deterministic input orders may produce non-deterministic output contents. To ensure that replay—Storm's internal fault-tolerance strategy—is deterministic, Blazes will recommend that the topology be coordinated—the programmer can achieve this by making the topology "transactional" (in Storm terminology), totally ordering the batch commits.

If, on the other hand, the input stream is sealed on *batch*, Blazes instead produces this reduction:

$$\text{(p)}\,\dfrac{\mathbf{Seal}_{batch}\quad \text{CR}}{\texttt{Splitter}\,\dfrac{\mathbf{Seal}_{batch}}{\text{(p)}\,\dfrac{\mathbf{Seal}_{batch}\quad \text{OW}_{word,batch}}{\texttt{Count}\,\dfrac{\mathbf{Async}}{\text{(p)}\,\dfrac{\mathbf{Async}\quad \text{CW}}{\texttt{Committer}\,\dfrac{\mathbf{Async}}{\mathbf{Async}}}}}}}$$

Because a batch is atomic (its contents may be completely determined once a seal record arrives) and independent (emitting a processed batch never affects any other batches), the topology will produce deterministic contents in its (possibly nondeterministically-ordered) outputs—a requirement for Storm's replay-based fault-tolerance—under all interleavings.

Ad-reporting system

Next we describe how we might annotate the various components of the ad-reporting system. As we discuss in Section 4.6, these annotations can be automatically extracted from the Bloom syntax; for exposition, in this section we discuss how a programmer might manually annotate an analogous dataflow written in a language without Bloom's static-analysis capabilities. As we will see, ensuring deterministic outputs will require different mechanisms for the different queries listed in Figure 4.5.

Component annotations

The cache is clearly a stateful component, but since its state is append-only and order-independent we may annotate it *CW*. Because the data-collection path through the reporting server simply appends clicks and impressions to a log, we annotate this path *CW* (it modifies state, but that state is nevertheless convergent) also.

All that remains is to annotate the path through the reporting component corresponding to the various continuous queries enumerated in Section 4.2. `Report` is a replicated component, so we supply the **Rep** annotation for all instantiations. We annotate the query path corresponding to *THRESH CR*—it is both read-only (in the context of the reporting module) and confluent. It never emits a record until the ad impressions have reached the given threshold, and since they can never again drop below the threshold, the query answer "ad X has > 1000 impressions" holds forever.

We annotate queries *POOR* and *CAMPAIGN* OR_{id} and $OR_{id,campaign}$, respectively. They too are read-only queries, having no effect on reporting server state, but can return different contents in different executions, recording the effect of message races between click and request messages.

We give query *WINDOW* the annotation $OR_{id,window}$. Unlike *POOR* and *CAMPAIGN*, *WINDOW* includes the input stream attribute *window* in its grouping clause. Its outputs are therefore partitioned by values of *window*, so BLAZES will be able to employ a coordination-free sealing strategy to force the component to output deterministic results if it can determine that the input stream is sealed on *window*.

```
Cache:
  annotation:
    - { from: request, to: response, label: CR }
    - { from: response, to: response, label: CW }
    - { from: request, to: request, label: CR }
Report:
  annotation:
    - { from: click, to: response, label: CW }
POOR: { from: request, to: response, label: OR,
      subscript: [id] }
THRESH: { from: request, to: response, label: CR }
WINDOW: { from: request, to: response, label: OR,
      subscript: [id, window] }
CAMPAIGN: { from: request, to: response, label: OR,
      subscript: [id, campaign] }
```

Analysis

Having annotated all of the instantiations of the reporting server component for different queries, we may now consider how BLAZES derives output stream labels for the global dataflow. If we supply *THRESH*, BLAZES performs the following reductions, deriving a final label of **Async** for the output path from cache to sink:

$$\texttt{Report}\ \cfrac{(p)\,\cfrac{\textbf{Async}\qquad CW}{\textbf{Async}}\quad (p)\,\cfrac{\textbf{Async}\qquad CR}{\textbf{Async}}\qquad Rep}{(p)\,\cfrac{\textbf{Async}\qquad\qquad CW}{\texttt{Cache}\ \cfrac{\textbf{Async}}{\textbf{Async}}}}$$

All components are confluent, so the complete dataflow produces deterministic outputs without coordination. If we chose, we could encapsulate the service as a single component with annotation *CW*.

If we consider query *POOR* with no input stream annotations, it leads to the following reduction:

$$\texttt{Report}\ \cfrac{(p)\,\cfrac{\textbf{Async}\qquad CW}{\textbf{Async}}\quad (2)\,\cfrac{\textbf{Async}\qquad \mathrm{OR}_{campaign}}{\textbf{NDRead}_{campaign}}\qquad Rep}{(3)\,\cfrac{\textbf{Inst}\qquad\qquad CW}{\texttt{Cache}\ \cfrac{\textbf{Taint}}{\textbf{Split}}}}$$

The poor performers query is not confluent: it produces non-deterministic outputs. Because these outputs mutate a stateful, replicated component (i.e., the cache) that affects system outputs, the output stream is tainted by divergent replica state. Preventing split brain will require a coordination strategy that controls message delivery order to the reporting server.

On the other hand, if the input stream is sealed on *campaign*, BLAZES instead performs this reduction:

$$\texttt{Cache}\ \cfrac{(p)\ \cfrac{\texttt{Report}\ \cfrac{(p)\ \cfrac{\mathbf{Seal}_{campaign} \quad CW}{\mathbf{Seal}_{campaign}} \quad (2)\ \cfrac{\mathbf{Async} \quad \mathrm{OR}_{campaign}}{\mathbf{NDRead}_{campaign}} \quad Rep}{\mathbf{Async}} \quad CW}{\mathbf{Async}}}{\mathbf{Async}}$$

Appropriately sealing inputs to non-confluent components can make them behave like confluent components. Implementing this sealing strategy does not require global coordination, but merely some synchronization between stream producers and consumers—we sketch the protocol in Section 4.4.

Similarly, *WINDOW* (given an input stream sealed on *window*) reduces to **Async**.

Bloom Shopping Cart

Finally, we revisit the "destructive shopping cart" presented in Section 3.3. We show how with appropriate programmer insight about sealable streams, we can rewrite the destructive cart program in such a way that it achieves the same coarse-grained coordination as its "disorderly" counterpart (presented in Section 3.3). Better still, BLAZES synthesizes the required coordination code automatically.

Recall that the initial implementation of a shopping cart in Bloom reused a predefined component (the ReplicatedKVS module shown in Figure 3.3) to store the internal state of the service. Unfortunately, the naive coordination strategy for such a composition—establishing a total order over the calls to *kvput*—requires a round of distributed coordination (consensus, or a call to an ordering service) for every client action. In Section 3.3, we reimplemented the service to require only coarse-grained coordination, by applying a combination of monotonic design patterns (e.g., model the contents of the cart as a set of deletions and a set of insertions) and domain knowledge (e.g., that each "cart" is modified by only one client). For the reward of cheaper synchronization, we paid a a software engineering penalty: we were unable to implement a layered architecture that reused the well-understood replicated key-value store module, and were forced to build the coordination-optimized cart service from scratch.

Component annotations

Unsurprisingly, BLAZES extracts these annotations from its analysis of DestructiveCart:

```
Cart:
  annotation:
    - { from: action, to: put, label: CR }
    - { from: get_response, to: response, CR }
KVS:
  annotation:
    - { from: kvput, to: kvget_response, label: OW,
      subscript: [key] }
    - { from: kvget, to: kvget_response, label: OR,
      subscript: [key] }
```

The Cart component is stateless and confluent: it simply transforms and proxies calls to the underlying key-value store. The `kvput` interface of the KVS is stateful and order-sensitive: every new key value replaces the old one, and this affects the output of calls to `kvget`, which is read-only but order-sensitive (**OR**).

Analysis

In the absence of sealing information, BLAZES warns us that replicas of the cart can reach a split-brain state:

$$\texttt{KVS}\,\dfrac{(2)\,\dfrac{\text{Cart}\,\dfrac{\textbf{Async} \quad \mathit{CR}}{\textbf{Async}} \quad \mathrm{OR}_{\mathit{key}}}{\textbf{NDRead}_{\mathit{key}}} \qquad 2\,\dfrac{\text{Cart}\,\dfrac{\textbf{Async} \quad \mathit{CR}}{\textbf{Async}} \quad \mathrm{OW}_{\mathit{key}}}{\text{Taint}} \quad \mathit{Rep}}{\textbf{Split}}$$

Both interfaces, however, are sealable on `key`: if data producers can provide punctuations on `key`, updates can be applied in batch and reads can be postponed until after all of the updates. Analysis of the composed program reveals that *injectivefd*(*client_action.session*, *kvput.key*) holds due to the rule on Line 9 in Figure 3.8, in which the values for `kvput.key` are drawn directly from `client_action.session`. The `session` attribute is certainly sealable: the semantics of the cart application dictate that after *checkout*, no new additions or deletions to a client's shopping cart are allowed.

$$\texttt{KVS}\,\dfrac{(2)\,\dfrac{\text{Cart}\,\dfrac{\textbf{Async} \quad \mathit{CR}}{\textbf{Async}} \quad \mathrm{OR}_{\mathit{key}}}{\textbf{NDRead}_{\mathit{key}}} \qquad 2\,\dfrac{\text{Cart}\,\dfrac{\textbf{Seal}_{\mathit{session}} \quad \mathit{CR}}{\textbf{Seal}_{\mathit{session}}} \quad \mathrm{OW}_{\mathit{key}}}{\text{Seal}_{\mathit{session}}} \quad \mathit{Rep}}{\textbf{Async}}$$

Discussion

This last example demonstrates how BLAZES improves upon the analysis capabilities provided by Bloom. Monotonicity analysis (Section 3.3) focused the attention of the programmer on program locations that may require coordination; in Section 3.3 we showed how we could exploit this guidance to rewrite the destructive cart program to reduce its implicit coordination requirements. In exchange for improved performance and availability, we were willing to program in a less *declarative* mode, reasoning explicitly about the possible program implementations rather than merely the desired outcomes. BLAZES handles such concerns automatically, allowing programmers to focus their attention on functional requirements. In this respect, BLAZES plays a role similar to a simple query optimizer; among a space of functionally equivalent implementations, it attempts to choose one that minimizes cost, freeing the programmer from having to navigate this space themselves.

Blazes exploits the insight that when data streams are sealable in a way that is compatible with the components into which they flow, *any* program that requires coordination to rule out the anomalies described in Section 4.2 can be rewritten into a program that achieves the required coordination using localized, barrier-based synchronization rather than enforcing a global ordering.

4.6 Bloom Integration

To provide input for the "grey box" functionality of Blazes, programmers must convert their intuitions about component behavior and execution topology into the annotations introduced in Section 4.3. As we saw in Section 4.5, this process is often quite natural; unfortunately, as we learned in Section 4.5, it becomes increasingly burdensome as component complexity increases.

Given an appropriately constrained language, the necessary annotations can be extracted automatically via static analysis. In this section, we describe how we used the Bloom language to enable a "white box" system, in which unadorned programs can be submitted, analyzed and—if necessary to ensure consistent outcomes—automatically rewritten.

Bloom components

Bloom programs are bundles of declarative *rules* describing the contents of logical *collections* and how they change over time. To enable encapsulation and reuse, a Bloom program may be expressed as a collection of *modules* with input and output interfaces associated with relational schemas. Hence modules map naturally to dataflow components.

Each module also defines an internal dataflow from input to output interfaces, whose components are the individual rules. Blazes analyzes this dataflow graph to automatically derive component annotations for Bloom modules.

White box requirements

To select appropriate component labels, Blazes needs to determine whether a component is confluent and whether it has internal state that evolves over time. To determine when sealing strategies are applicable, Blazes needs a way to "chase" [124] the injective functional dependencies described in Section 4.4 transitively across compositions.

As we show, we meet all three requirements by applying standard techniques from database theory and logic programming to programs written in Bloom.

Confluence and state

As we described in Section 4.2, the CALM theorem establishes that all *monotonic* programs are confluent. The Bloom runtime includes analysis capabilities to identify—at the granularity of program statements—nonmonotonic operations, which can be conservatively identified with a syntactic test. A component free of such operations is provably order-insensitive. Similarly, Bloom distinguishes syntactically between transient event streams and stored state. A simple flow analysis automatically determines if a component is stateful: a component is stateful if and only if a persistent `table` is reachable from an `input interface`. Together, these analyses are sufficient to determine the CROW annotations (except for the subscripts, which we describe next) for every Bloom statement in a given module.

Support for sealing

What remains is to determine the appropriate partition subscripts for non-confluent labels (the *gate* in OW_{gate} and OR_{gate}) and to define an efficient procedure for deciding whether `injectivefd` holds.

Recall that in Section 4.3 we chose a subscript for the SQL-like `WINDOW` query by considering its *group by* clause; by definition, grouping sets are independent of each other. Similarly, the columns referenced in the *where* clause of an antijoin identify sealable partitions.[3] Applying this reasoning, Blazes selects subscripts in the following way:

1. If the Bloom statement is an aggregation (*group by*), the subscript is the set of grouping columns.

2. If the statement is an antijoin (*not in*), the subscript is the set of columns occurring in the theta clause that are bound to the anti-joined relation.

We can track the lineage of an individual attribute (processed by a nonmonotonic operator) by querying Bloom's system catalog, which details how each rule application transforms (or preserves) attribute values that appear in the module's input interfaces. To define a sound but incomplete `injectivefd`, we again exploit the common special case that the identity function is injective, as is any series of transitive applications of the identity function. For example, given $S \equiv \pi_a \pi_{ab} \pi_{abc} R$, we have $injectivefd(R.a, S.a)$.

4.7 Evaluation

In Section 4.2, we considered the *consequences* of under-coordinating distributed dataflows. In this section, we measure the *costs* of over-coordination by comparing the performance of two distinct dataflow systems, each under two coordination regimes: a generic order-based coordination strategy and an application-specific sealing strategy.

We ran our experiments on Amazon EC2. In all cases, we average results over three runs; error bars are shown on the graphs.

[3]Intuitively, we can deterministically evaluate `select * from R where x not in (select x from S where y = 'Yahoo!')` for any tuples of R once we have established that a.) there will be no more records in S with y = 'Yahoo!', or b.) there will *never* be a corresponding S.x.

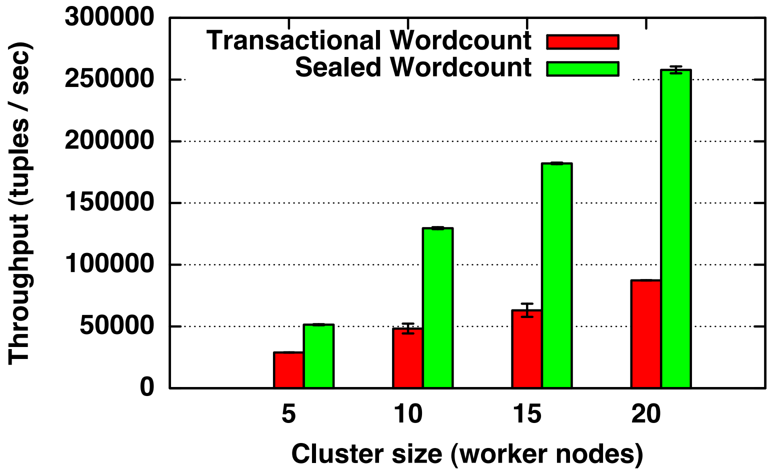


Figure 4.8: The effect of coordination on throughput for a Storm topology computing a streaming wordcount.

Storm wordcount

To evaluate the potential savings of avoiding unnecessary synchronization, we implemented two versions of the streaming wordcount query described in Section 4. Both process an identical stream of tweets and produce the same outputs. They differ in that the first implementation is a "transactional topology," in which the `Commit` components use coordination to ensure that outputs are committed to the backing store in a serial order. [4] The second—which Blazes has ensured will produce deterministic outcomes without any global coordination—is a "nontransactional topology." We optimized the batch size and cluster configurations of both implementations to maximize throughput.

We used a single dedicated node (as the documentation recommends) for the Storm master (or "nimbus") and three Zookeeper servers. In each experiment, we allowed the topology to "warm up" and reach steady state by running it for 10 minutes.

Figure 4.8 plots the throughput of the coordinated and uncoordinated implementations of the wordcount dataflow as a function of the cluster size. The overhead of conservatively deploying a transactional topology is considerable. The uncoordinated dataflow has a peak throughput roughly 1.8 times that of its coordinated counterpart in a 5-node deployment. As we scale up the cluster to 20 nodes, the difference in throughput grows to 3X.

Ad reporting

To compare the performance of the sealing and ordering coordination strategies, we conducted a series of experiments using a Bloom implementation of the complete ad tracking network introduced in Section 4. For ad servers, which simply generate click logs and forward them to reporting

[4]Storm uses Zookeeper for coordination.

servers, we used 10 `micro` instances. We created 3 reporting servers using `medium` instances. Our Zookeeper cluster consisted of 3 `small` instances. All instances were located in the same AWS availability zone.

Ad servers generate a workload of 1000 log entries per server. Servers batch messages, dispatching 50 click log messages at a time, and sleep periodically. During the workload, we also pose a number of requests to the reporting servers, corresponding to advertisements with entries in the click logs. The reporting servers all implement the continuous query *CAMPAIGN*.

Although this system—implemented in the Bloom language prototype—does not illustrate the numbers we would expect in a high-performance implementation, we will see that it highlights some important *relative* patterns across different coordination strategies.

Baseline: No Coordination

For the first run, we do not enable the BLAZES preprocessor. Thus click logs and requests flow in an uncoordinated fashion to the reporting servers. The uncoordinated run provides a lower bound for performance of appropriately coordinated implementations. However, it does not have the same semantics. We confirmed by observation that certain queries posed to multiple reporting server replicas returned inconsistent results.

The line labeled "Uncoordinated" in Figures 4.9 and 4.10 shows the log records processed over time for the uncoordinated run, for systems with 5 and 10 ad servers, respectively.

Ordering Strategy

In the next run we enabled the BLAZES preprocessor but did not supply any input stream annotations. BLAZES recognized the potential for inconsistent answers across replicas and synthesized a coordination strategy based on ordering. By inserting calls to Zookeeper, all click log entries and requests were delivered in the same order to all replicas. The line labeled "Ordered" in Figures 4.9 and 4.10 plots the records processed over time for this strategy.

The ordering strategy ruled out inconsistent answers from replicas but incurred a significant performance penalty. Scaling up the number of ad servers by a factor of two had little effect on the performance of the uncoordinated implementation, but increased the processing time in the coordinated run by a factor of three.

Sealing Strategies

For the last experiments we provided the input annotation $\textbf{Seal}_{\textit{campaign}}$ and embedded punctuations in the ad click stream indicating when there would be no further log records for a particular campaign. Recognizing the compatibility between a stream sealed in this fashion and the aggregate query in *CAMPAIGN* (a "group-by" on *id, campaign*), BLAZES synthesized a seal-based coordination strategy that delays answers for a particular campaign until that campaign is fully determined.

Using the seal-based strategy, reporting servers do not need to wait until events are globally ordered before processing them. Instead, events are processed as soon as a reporting server can determine that they belong to a partition that is sealed. After each ad server forwards its final

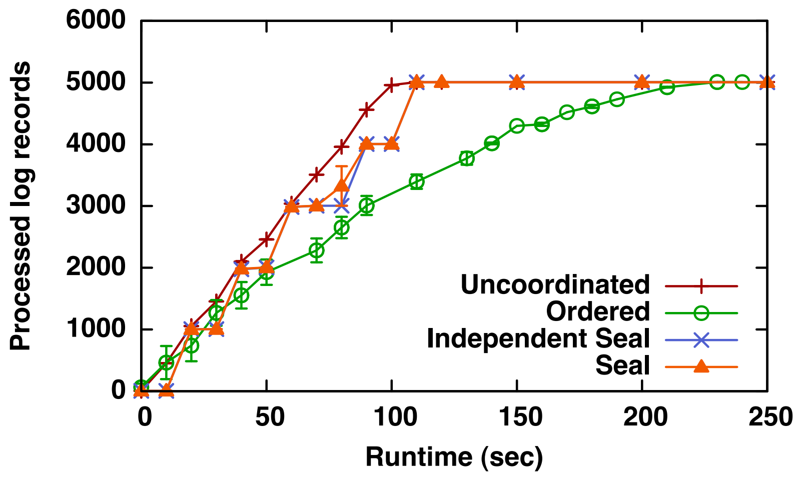


Figure 4.9: Log records processed over time, 5 ad servers

click record for a given campaign to the replicated reporting servers, it sends a seal message for that campaign, which contains a digest of the set of click messages it generated. The reporting servers use Zookeeper to determine the set of ad servers responsible for each campaign. When a reporting server has received seal messages from all producers for a given campaign, it compares the buffered click records to the seal digest(s); if they match, it emits the partition for processing.

Figures 4.9 and 4.10 compare the performance of seal-based strategies to ordered and uncoordinated runs. We plot two topologies: “Independent seal” corresponds to a partitioning in which each campaign is mastered at exactly one adserver, while in “Seal,” all ad servers produce click records for all campaigns. Note that both runs that used seals closely track the performance of the uncoordinated run; doubling the number of ad servers has little effect on the overall processing time.

Figure 4.11 plots the 10-server run but omits the ordering strategy, to highlight the differences between the two seal-based topologies. As we would expect, “independent seals” result in executions with slightly lower latencies because reporting servers may process partitions as soon as a single seal message appears (since each partition has a single producer). By contrast, the step-like shape of the non-independent seal strategy reflects the fact that reporting servers delay processing input partitions until they have received a seal record from every producer. Partitioning the data across ad servers so as to place advertisement content close to consumers (i.e., partitioning by ad id) caused campaigns to be spread across ad servers. This partitioning conflicted with the coordination strategy, which would have performed better had it associated each campaign with a unique producer. We revisit the notion of “coordination locality” in Section 4.9.

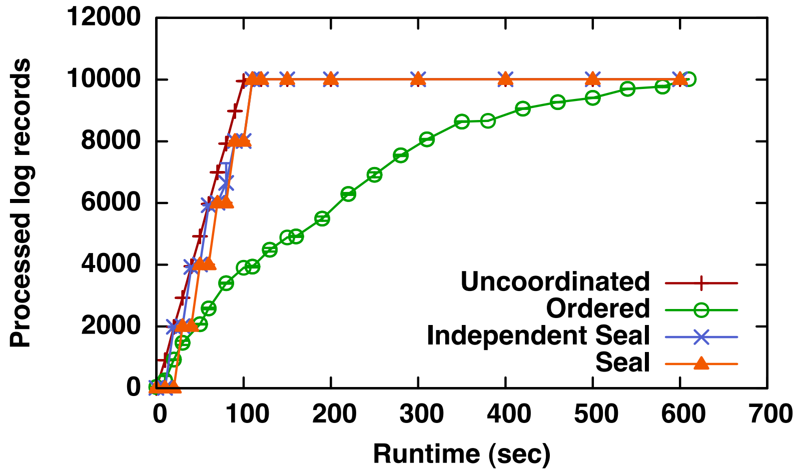


Figure 4.10: Log records processed over time, 10 ad servers

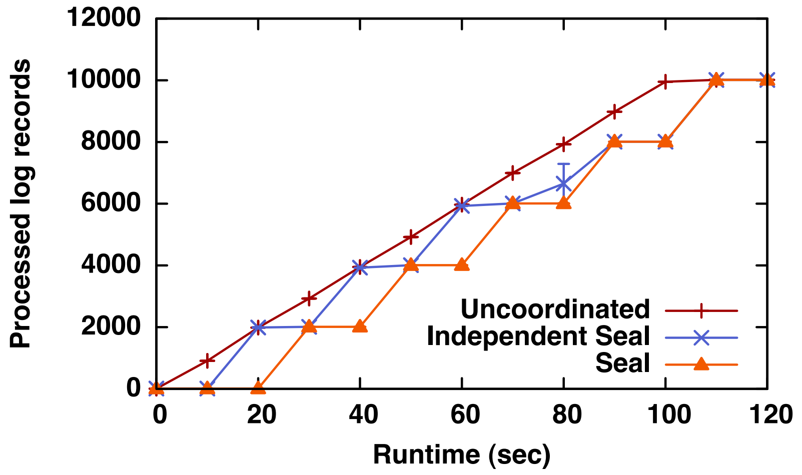


Figure 4.11: Seal-based strategies, 10 ad servers

4.8 Related Work

Our approach to automatically coordinating distributed services draws inspiration from the literature on both distributed systems and databases. Ensuring consistent replica state by establishing a total order of message delivery is the technique adopted by state machine replication [148]; each component implements a deterministic state machine, and a global coordination service such as atomic broadcast or Multipaxos decides the message order.

In the context of Dedalus, Marczak et al. draw a connection between stratified evaluation of conventional logic programming languages and distributed protocols to ensure consistency [127]. They describe a program rewrite that ensures deterministic executions by preventing any node from performing a nonmonotonic operation until that operation's inputs are "determined." Agents processing or contributing to a distributed relation carry out a voting-based protocol to agree when the contents of the relation are completely determined. This rewrite—essentially a restricted version of the sealing construct defined in this chapter—treats entire input collections as sealable partitions, and hence is not defined for unbounded input relations.

Commutativity of concurrent operations is a subject of interest for parallel as well as distributed programming languages. Commutativity analysis [144] uses symbolic analysis to test whether different method-invocation orders always lead to the same result; when they do, lock-free parallel executions are possible. λ_{par} [102] is a parallel functional language in which program state is constrained to grow according to a partial order and queries are restricted, enabling the creation of programs that are "deterministic by construction." CRDTs [153] are convergent replicated data structures; any CRDT could be treated as a dataflow component annotated as CW.

Like reactive distributed systems, streaming databases [8, 24, 41] must operate over unbounded inputs—we have borrowed much of our stream formalism from this tradition. The CQL language distinguishes between monotonic and nonmonotonic operations; the former support efficient strategies for converting between streams and relations due to their pipelineability. The Aurora system also distinguishes between "order-agnostic" and "order-sensitive" relational operators.

Similarly to our work, the Gemini system [111] attempts to efficiently and correctly evaluate a workload with heterogeneous consistency requirements, taking advantage of cheaper strategies for operations that require only weak orderings. They define a novel consistency model called RedBlue consistency, which guarantees convergence of replica state without enforcing determinism of queries or updates. By contrast, Blazes makes guarantees about composed services, which requires reasoning about the properties of streams as well as component state.

4.9 Discussion

Blazes allows programmers to avoid the burden of deciding *when* and *how* to use the (precious) resource of distributed coordination. With this difficulty out of the way, the programmer may focus their insight on other difficult problems, such as *placement*—both the physical placement of data and the logical placement of components.

Rules of thumb regarding data placement strategies typically involve predicting patterns of access that exhibit spatial and temporal locality; data items that are accessed together should be near one another, and data items accessed frequently should be cached. Our discussion of Blazes, particularly the evaluation of different seal-based strategies in Section 4.7, hints that access patterns are only part of the picture: because the dominant cost in large-scale systems is distributed coordination, we must also consider *coordination locality*—a measure inversely proportional to the number of nodes that must communicate to deterministically process a segment of data. If coordination locality is in conflict with spatial locality (e.g., the non-independent partitioning strategy that clusters ads likely to be served together at the cost of distributing campaigns across multiple nodes), problems emerge.

Given a dataflow of components, Blazes determines the need for (and appropriately applies) coordination. But was it the right dataflow? We might wish to ask whether a different logical dataflow that produces the same output supports cheaper coordination strategies. Some design patterns emerge from our discussion. The first is that, when possible, replication should be placed *upstream* of confluent components. Since they are tolerant of all import orders, weak and inexpensive replication strategies (like gossip) are sufficient to ensure confluent outputs. Similarly, caches should be placed *downstream* of confluent components. Since such components never retract outputs, simple, append-only caching logic may be used.[5] More challenging and compelling is the possibility of again pushing these design principles into a compiler and automatically rewriting dataflows.

[5]Distributed garbage collection, based on sealing, is an avenue of future research.

Chapter 5

Lineage-driven Fault Injection

What a faint-heart! We must work outward from the middle of the maze. We will start with something simple.
– Thomasina, in *Arcadia* [158].

Throughout this thesis, we discussed a variety of mechanisms that provide fault-tolerance via *redundancy*, including the reliable delivery protocols presented in Figure 3.1, the replay strategy used by Storm (Section 4), and the replication strategies used by the Bloom KVS (Section 3.3) and the ad reporting network (Section 4). Until now, we assumed that all of these mechanisms provided adequate fault-tolerance, and focused on the *consistency* concerns that naturally arise in the presence of redundant state and computation. Chapters 3 and 4 describe languages, techniques and tools that help ensure that uncertainty in the ordering and timing of message delivery do not lead to disagreement among replicas, across replays, and so on.

Assuming that in the fullness of time all partitions heal and all messages are delivered made it easier for us to focus narrowly on assurances about data consistency, using tools such as Bloom's consistency analysis and Blazes. How can we achieve similar assurances that our fault-tolerance mechanisms are implemented correctly and provide the guarantees that we expect from them? In this chapter, we develop Lineage-driven fault injection, an analysis methodology that both quickly identifies fault-tolerance bugs in distributed programs and, when possible, provides strong assurances that no such bugs exist. It does so by using data lineage, obtained during program executions, to reason about redundancy of support for correct outcomes.

5.1 Fault-tolerant distributed systems

Fault tolerance is a critical feature of modern data management systems, which are often distributed to accommodate massive data sizes [163, 43, 81, 7, 29, 53, 115]. Fault-tolerant protocols such as atomic commit [72, 155], leader election [65], process pairs [74] and data replication [11, 156, 162] are experiencing a renaissance in the context of these modern architectures.

With so many mechanisms from which to choose, it is tempting to take a bottom-up approach to distributed system design, enriching new system architectures with well-understood fault tolerance mechanisms and henceforth assuming that failures will not affect system outcomes. Unfortunately, fault-tolerance is a *global* property of entire systems, and guarantees about the behavior of individual components do not necessarily hold under composition. It is difficult to design and reason about the fault-tolerance of individual components, and often equally difficult to assemble a fault-tolerant system even when given fault-tolerant components, as witnessed by recent data management system failures [36, 125] and bugs [79, 97].

Top-down testing approaches—which perturb and observe the behavior of complex systems—are an attractive alternative to verification of individual components. Fault injection [129, 6, 55, 79, 93] is the dominant top-down approach in the software engineering and dependability communities. With minimal programmer investment, fault injection can quickly identify shallow bugs caused by a small number of independent faults. Unfortunately, fault injection is poorly suited to discovering rare counterexamples involving complex combinations of multiple instances and types of faults (e.g., a network partition followed by a crash failure). Approaches such as Chaos Monkey [6] explore faults randomly, and hence are unlikely to find rare error conditions caused by complex combinations of failures. Worse still, fault injection techniques—regardless of their search strategy—cannot effectively guarantee *coverage* of the space of possible failure scenarios. Frameworks such as FATE [79] use a combination of brute-force search and heuristics to guide the enumeration of faults; such heuristic search strategies can be effective at uncovering rare failure scenarios, but, like random search, they do little to cover the space of possible executions.

An ideal top-down solution for ensuring that distributed data management systems operate correctly under fault would enrich the fault injection methodology with the best features of formal component verification. In addition to identifying bugs, a principled fault injector should provide assurances. The analysis should be *sound*: any generated counterexamples should correspond to meaningful fault tolerance bugs. When possible, it should also be *complete*: when analysis completes without finding counterexamples for a particular input and execution bound, it should *guarantee* that no bugs exist for that configuration, even if the space of possible executions is enormous.

To achieve these goals, we propose a novel top-down strategy for discovering bugs in distributed data management systems: *lineage-driven fault injection* (LDFI). LDFI is inspired by the notion of *data lineage* [37, 54, 134, 94, 177, 78], which allows it to directly connect system outcomes to the data and messages that led to them. LDFI uses data lineage to reason *backwards* (from effects to causes) about whether a given correct outcome could have failed to occur due to some combination of faults. Rather than generating faults at random (or using application-specific heuristics), a lineage-driven fault injector chooses only those failures that could have affected a known good outcome, exercising fault-tolerance code at increasing levels of complexity. Injecting faults in this targeted way allows LDFI to provide completeness guarantees like those achievable with formal methods such as model checking [171, 135, 85, 105], which have typically been used to verify small protocols in a bottom-up fashion. When bugs are encountered, LDFI's top-down approach provides—in addition to a counterexample trace—fine-grained data lineage visualizations to help programmers understand the *root cause* of the bad outcome and consider possible remediation strategies.

We present MOLLY, an implementation of LDFI. Like fault injection, MOLLY finds bugs in large-scale, complex distributed systems quickly, in many cases using an order of magnitude fewer executions than a random fault injector. Like formal methods, Molly finds *all* of the bugs that could be triggered by failures: when a Molly execution completes without counterexamples it certifies that no fault-tolerance bugs exist for a given configuration. Molly integrates naturally with root-cause debugging by converting counterexamples into data lineage visualizations. We use Molly to study a collection of fault-tolerant protocols from the database and distributed systems literature, including reliable broadcast and commit protocols, as well as models of modern systems such as the Kafka reliable message queue. In our experiments, Molly quickly identifies 7 critical bugs in 14 fault-tolerant systems; for the remaining 7 systems, it provides a guarantee that no invariant violations exist up to a bounded execution depth, an assurance that state-of-the-art fault injectors cannot provide.

Example: Kafka replication

To ground and motivate our work, we consider a recently-discovered bug in the replication protocol of the Kafka [7] distributed message queue. In Kafka 0.80 (Beta), a Zookeeper service [87]—a strongly consistent metadata store—maintains and publishes a list of up-to-date replicas (the "in-sync-replicas" list or ISR), one of which is chosen as the leader, to all clients and replicas. Clients forward write requests to the leader, which forwards them to all replicas in the ISR; when the leader has received acknowledgments from all replicas, it acknowledges the client.

If replication is implemented correctly (and assuming no Byzantine failures) a system with three replicas should be able to survive one (permanent) crash failure while ensuring a "stable write" invariant: acknowledged writes will be stably stored on a non-failed replica. Kingsbury [97] demonstrates a vulnerability in the replication logic by witnessing an execution in which this invariant is violated despite the fact that only one server crashes.

In brief, the execution proceeds as follows: two nodes b and c from a replica set $\{a, b, c\}$ are partitioned away from the leader a and the Zookeeper service; as a result, they are removed from the ISR. Node a is now the leader and sole member of the quorum. It accepts a write, acknowledges the client without any dissemination, and then crashes. The acknowledged write is lost.

The durability bug—which seems quite obvious in this post-hoc analysis—illustrates how difficult it can be to reason about the complex interactions that arise via composition of systems and multiple failures. Both Zookeeper and primary/backup replication are individually correct software components, but multiple kinds and instances of failures (message loss failure followed by node failure) result in incorrect behavior in the composition of the components. The problem is not so much a protocol bug (the client receives an acknowledgment only when the write is durably stored on *all* replicas) as it is a dynamic misconfiguration of the replication protocol, caused by a (locally correct) view change propagated by the Zookeeper service. Kingsbury used his experience and intuition to predict and ultimately witness the bug. But is it possible to encode that kind of intuition into a general-purpose tool that can identify a wide variety of bugs in fault-tolerant programs and systems?

Molly, a lineage-driven fault injector

Given a description of the Kafka replication protocol, we might ask a question about forward executions: starting from an initial state, could some execution falsify the invariant? This question gives us very little direction about how to search for a counterexample. Instead, LDFI works backwards from results, asking *why* a given write is stable in a particular execution trace. For example, a write initiated by (and acknowledged at) the client is stable because (among other reasons) the write was forwarded to a (correct) node *b*, which now stores the write in its log. It was forwarded to *b* by the leader node *a*, because *b* was in *a*'s ISR. *b* was in the ISR because the Zookeeper service considered *b* to be up and forwarded the updated view membership to *a*. Zookeeper believed *b* to be up because *a* received timely acknowledgment messages from *b*. Most of the preceding events happened due to deterministic steps in the protocol. However, certain events (namely communication) were uncertain; in a different execution, they might not have succeeded. These are precisely the events we should explore to find the execution of interest: due to a temporary partition that prevents timely acknowledgments from *b* and *c*, they are removed from the ISR, and the rest is history.

LDFI takes the sequence of computational steps that led to a good outcome (the outcome's *lineage*) and reasons about whether some combination of failures could have prevented all "support" for the outcome. If it finds such a combination, it has discovered a schedule of interest for fault injection: based on the known outcome lineage, under this combination of faults the program might fail to produce the outcome. However, in most fault-tolerant programs multiple independent computations produce the important outcomes; in an alternate execution with failures, a different computation might produce the good outcome in another way. As a result, LDFI alternates between identifying potential counterexamples using lineage and performing concrete executions to confirm whether they correspond to true counterexamples.

Figure 5.1 outlines the architecture of Molly, an implementation of LDFI. Given a distributed program and representative inputs, Molly performs a *forward* step, obtaining a outcome by performing a failure-free concrete evaluation of the program. The *hazard analysis* component then performs a *backward* step, extracting the lineage of the outcome and converting it into a CNF formula that is passed to a SAT solver. The SAT solutions—failures that could falsify all derivations of the good outcome—are transformed into program inputs for the next forward step. Molly continues to execute the program over these failure inputs until it either witnesses an invariant violation (producing a counterexample) or exhausts the potential counterexamples (hence guaranteeing that none exist for the given input and failure model). To enable counterexample-driven debugging, Molly presents traces of buggy executions to the user as a collection of visualizations including both a Lamport diagram [104] capturing communication activity in the trace, and lineage graphs detailing the data dependencies that produced intermediate node states.

The remainder of the chapter is organized as follows. In Section 5.2, we describe the system model and the key abstractions underlying LDFI. Section 5.3 provides intuition for how LDFI uses lineage to reason about possible failures, in the context of a game between a programmer and a malicious environment. Section 5.4 gives details about how the Molly prototype was implemented using off-the-shelf components, including a Datalog evaluator and a SAT solver. In Section 3.3,

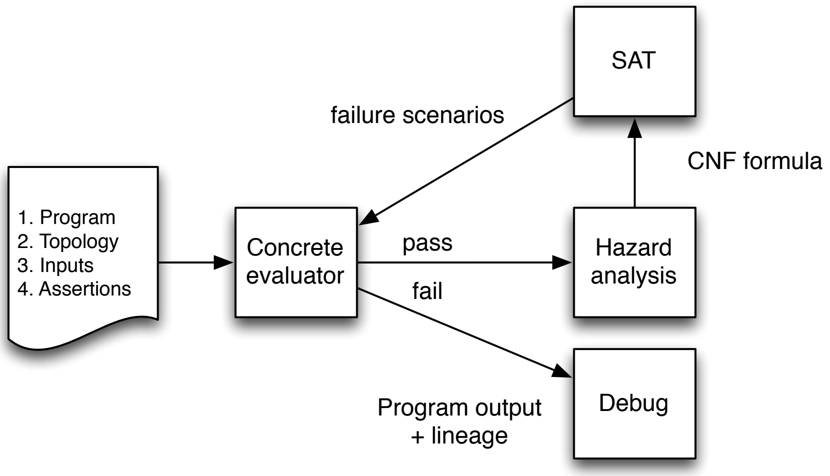


Figure 5.1: The Molly system performs forward/backward alternation: *forward* steps simulate concrete distributed executions with failures to produce lineage-enriched outputs, while *backward* steps use lineage to search for failure combinations that could falsify the outputs. Molly alternates between forward and backward steps until it has exhausted the possible falsifiers.

we study a collection of protocols and systems using Molly, and measure Molly's performance both in finding bugs and in guaranteeing their absence. Section 5.8 discusses limitations of the approach and directions for future work.

5.2 System model

Even in the absence of faults, distributed protocols can be buggy in a variety of ways, including sensitivity to message reordering or wall-clock timing. To maintain debugging focus and improve the efficiency of LDFI, we want to specifically analyze the effect of real-world faults on program outcomes. In this section, we describe a system model—along with its key simplifying abstractions—that underlies our approach to verifying fault-tolerant programs. While our simplifications set aside a number of potentially confounding factors, we will see in Section 3.3 that Molly is nevertheless able to identify critical bugs in a variety of both classical and current protocols. In Section 5.8, we reflect on the implications of our focused approach, and the class of protocols that can be effectively verified using our techniques.

Synchronous execution model

A general-purpose verifier must explore not only the non-deterministic *faults* that can occur in a distributed execution (such as message loss and crash failures), but also non-determinism in ordering and timing with respect to delivery of messages and scheduling of processes. Verifying the resilience of a distributed or parallel program to reordering—a key concern in the design of Bloom and the subject of Chapter 4—is a challenging research problem in its own right [17, 101, 144, 153]. In that vein of research, it is common to make a strong simplifying assumption: assume that all messages are eventually delivered, and systematically explore the reordering problem [159, 166].

We argue that for a large class of fault-tolerant protocols, we can discover or rule out many practically significant bugs using a dual assumption: *assume that successfully delivered messages are received in a deterministic order, and systematically explore failures*. To do this, we can simply evaluate an asynchronous distributed program in a synchronous simulation. Of course any such simplification forfeits completeness in the more general model: in our case, certain bugs that could arise in an asynchronous execution may go unnoticed. We trade this weaker guarantee (which nevertheless yields useful counterexamples) for a profound reduction in the number of executions that we need to consider, as we will see in Section 3.3. We discuss caveats further in Section 5.8.

Failure specifications

LDFI simulates failures that are likely to occur in large-scale distributed systems, such as (permanent) crash failures of nodes, message loss and (temporary) network partitions [56, 70, 28]. Byzantine failures are not considered, nor are *crash-recovery* failures, which involve both a window of message loss and the loss of ephemeral state. Verifying recovery protocols by modeling crash-recovery failures is an avenue of future work.

To ensure that verification terminates, we bound the *logical time* over which the simulation occurs. The notion of logical time—often used in distributed systems theory but generally elusive in practice—is well-defined in our simulations due to the synchronous execution abstraction described above. The internal parameter **EOT** (for end-of-time) represents a fixed logical bound on executions; the simulation then explores executions having no more than *EOT* global transitions (rounds of message transmission and state-changing internal events).

If the verifier explored all possible patterns of message loss up to *EOT*, it would always succeed in finding counterexamples for any non-trivial distributed program by simply dropping all necessary messages. However, infinite, arbitrary patterns of loss are uncommon in real distributed systems [27, 28]. More common are periods of intermittent failure or total partition, which eventually resolve (and then occur again, and so on). LDFI incorporates a notion of failure quiescence, allowing programs to attempt to recover from periods of lost connectivity. In addition to the *EOT* parameter, a second internal parameter indicates the *end of finite failures* (**EFF**) in a run, or the logical time at which message loss ceases. Molly ensures that $EFF < EOT$ to give the program time to recover from message losses. If $EFF = 0$, Molly does not explore message loss—this models a fail-stop environment, in which processes may only fail by crashing.

A *failure specification* (Fspec) consists of three integer parameters $\langle EOT, EFF, Crashes\rangle$; the first two are as above, and the third specifies the maximum number of crash failures that should be explored. For example, a failure specification of $\langle 6, 4, 1\rangle$ indicates that executions of up to 6 global transitions should be explored, with message loss permitted only from times 1-4, and zero or one node crashes. A crashed node behaves in the obvious way, ceasing to send messages or make internal transitions. A set of failures is *admissible* if it respects the given Fspec: there are no message omissions after *EFF* and no more than *Crash* crash failures.

In normal operation, Molly sets the Fspec parameters automatically by performing a *sweep*. First, *EFF* is set to 0 and *EOT* is increased until nontrivially correct executions [1] are produced. Then *EFF* is increased until either an invariant violation is produced—in which case *EOT* is again increased to permit the protocol to recover—or until $EFF = EOT - 1$, in which case both are increased by 1. This process continues until a user-supplied wall clock bound has elapsed, and Molly reports either the minimal parameter settings necessary to produce a counterexample, or (in the case of bug-free programs) the maximum parameter settings explored within the time bound. In some cases (e.g., validating protocols in a fail-stop model), users will choose to override the sweep and set the parameters manually.

Language

LDFI places certain requirements on the systems and languages under test. The backwards step requires clearly identified program *outputs*, along with fine-grained data lineage [44, 99] that captures details about the uncertain *communication* steps contributing to the outcome. The forward step requires that programs be executable with a runtime that supports interposition on the communication mechanism, to allow the simulator to drive the execution by controlling message loss and delivery timing. Languages like Erlang [25] and Akka [142] are attractive candidates because of their explicit communication, while aspect-oriented programming [95] could be used with a language such as Java to facilitate both trace extraction and communication interposition. Backwards slicing techniques [140] could be used to recover fine-grained lineage from the executions of programs written in a functional language.

For the Molly prototype, we chose to use Dedalus. Dedalus satisfies all of the requirements outlined above. Data lineage can be extracted from logic program executions via simple, well-understood program rewrites [99]. More importantly, Dedalus (and similar languages) abstract away the distinction between events, persistent state and communication (everything is just *data* and relationships among data elements) and make it simple to identify redundancy of computation and data in its various forms (as we will see in Section 5.3). A synchronous semantics for Dedalus, consistent with the synchronous execution assumption described in Section 5.2, was proposed by Interlandi et al. [88]

Note that because Bloom programs can be translated directly into Dedalus programs, the Molly prototype can accept Bloom programs as input as well. Since this automatic translation produces somewhat verbose Dedalus programs, we wrote the example programs in this Chapter by hand in Dedalus.

As we saw in Section 3.1, all state in Dedalus is captured in *relations*; computation is expressed via *rules* that describe how relations change over time. Dedalus programs are intended to be executed in a distributed fashion, such that relations are partitioned on their first attribute (their *location specifier*). Listing 5.1 shows a simple broadcast program written in Dedalus. Line 1 is a

[1]Executions in which no messages are sent are typically vacuously correct with respect to invariants.

```
1  log(Node, Pload) :- bcast(Node, Pload);
2  node(Node, Neighbor)@next :- node(Node, Neighbor);
3  log(Node, Pload)@next :- log(Node, Pload);
4  log(Node2, Pload)@async :- bcast(Node1, Pload),
5        node(Node1, Node2);
```

Listing 5.1: **simple-deliv**, a Dedalus program implementing a best-effort broadcast.

deductive rule, and has the same intuitive meaning as a Datalog rule: it says that if some 2-tuple exists in `bcast`, then it also exists in `log`. Lines 2-3 are *inductive* rules, describing a relation between a particular state and its *successor* state. Line 2, for example, says that if some 2-tuple exists in `node` at some time t, then that tuple also exists in `node` at time $t + 1$ (and by induction, forever after). Both are local rules, describing computations that individual nodes can perform given their internal state and the current set of events or messages. By contrast, the rule on lines 4-5 is a *distributed* rule, indicating an uncertain derivation across process boundaries. It expresses a simple *multicast* as a join between a stream of events (`bcast`) and the persistent relation `node`. Note that the conclusion of the rule—a `log` tuple—exists (assuming that a failure does not occur) at a different time (strictly later) than its premises, as well as at a different place (the address represented by the variable *Node2*).

Correctness properties

A program is fault-tolerant (with respect to a particular Fspec) if and only if its correctness assertions hold in all executions, where the space of executions is defined by all possible combinations of admissible failures. Distributed invariants are commonly expressed as implications of the form *precondition* → *postcondition*; an invariant violation is witnessed by an execution in which the precondition holds but the postcondition does not. For example, the *agreement* invariant for a consensus protocol states that *if* an agent decides a value, *then* all agents decide that value. The Kafka stable write invariant described in Section 5.1 states that if a write is acknowledged, then it exists on a correct (non-crashed) replica.

To support this pattern, Molly automatically defines two built-in *meta-outcomes* of programmer-defined arity, called `pre` and `post`. Molly users may express correctness assertions by defining these special relations—representing abstract program outcomes—as "views" over program state. If meta-outcomes are not defined, all persistent relations are treated as outcomes. This is acceptable for some simple protocols, such as naive reliable delivery (as we will see in Section 5.3); for more complex protocols, meta-outcomes should be used in order to mask unnecessary details. For example, a delivery protocol that suppresses redundant retries via message acknowledgments requires a meta-outcome that masks the exact number of ACKs and exposes only the contents of the `log` relation. Similarly, a consensus protocol is intended to reach *some* decision, though the decision it reaches may be different under different failures, so its meta-outcome should abstract away the particular decision.

Listing 5.2 shows a meta-outcome for reliable delivery, capturing the basic agreement require-

```
1 missing_log(A, Pl) :- log(X, Pl), node(X, A), notin log(A, Pl) ;
2 pre(X, Pl) :- log(X, Pl), notin crash(_, X, _) ;
3 post(X, Pl) :- log(X, Pl), notin missing_log(_, Pl) ;
```

Listing 5.2: A correctness specification for reliable broadcast. Correctness specifications define relations `pre` and `post`; intuitively, invariants are always expressed as implications: $pre \rightarrow post$. An execution is incorrect if `pre` holds but *post* does not, and is vacuously correct if `pre` does not hold. In reliable broadcast, the precondition states that a (correct) process has a log entry; the postcondition states that *all* correct processes have a log entry.

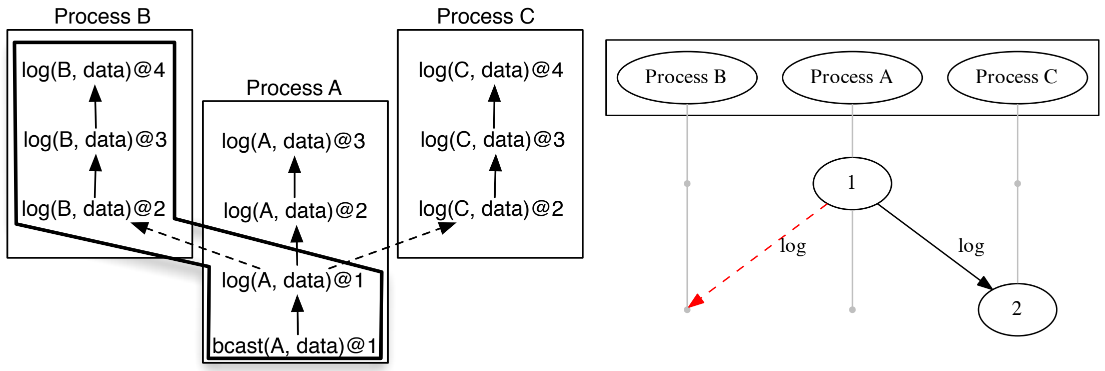


Figure 5.2: The unique support for the outcome `log(B, data)` is shown in bold; it can be falsified (as in the counterexample) by dropping a message from A to B at time 1 ($O_{(A,B,1)}$).

ment: *if a correct node delivers a message, then all correct nodes deliver it*. Line 1 defines a missing log entry as one that exists on some node but is absent from another. Line 2 defines the precondition: a log entry exists on a correct (non-crashed) node. If `pre` does not hold, then the execution is vacuously correct. Line 3 defines the postcondition: no node is missing the log entry. If `pre` holds but `post` does not, the invariant is violated and the lineage of these meta-outcomes can be presented to the user as a counterexample.

To run MOLLY, a user must provide a program with concrete inputs, and indicate which relations define the program's *outcomes*—by default, using `pre` and `post` as defined above. We now turn to an overview of how MOLLY automates the rest of the bug-finding process.

5.3 Using lineage to reason about fault-tolerance

One of the difficult truths of large-scale services is that if something can go wrong, eventually it will. Hence a reliable fault tolerance solution needs to account for unlikely events. A useful lens

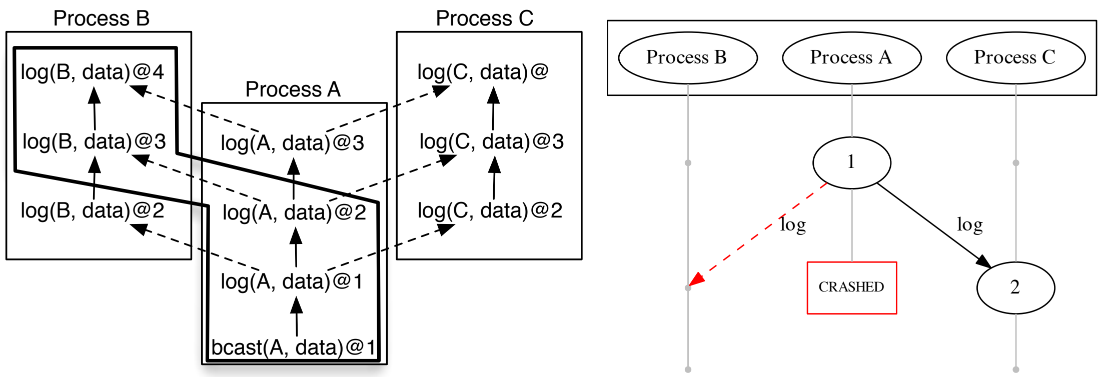


Figure 5.3: `retry-deliv` exhibits redundancy in time. All derivations of `log(B, data)@4` require a direct transmission from node A, which could crash (as it does in the counterexample). One of the redundant supports (falsifier: $O_{(A,B,2)}$) is highlighted.

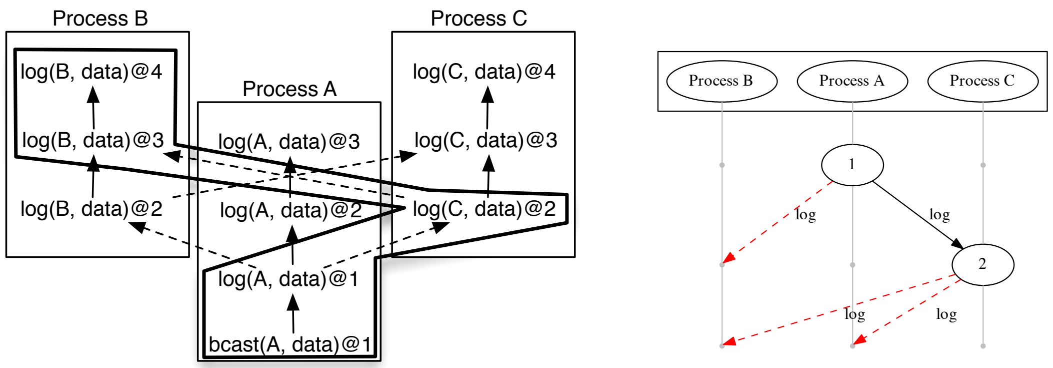


Figure 5.4: The lineage diagram for **classic-deliv** reveals redundancy in space but not time. In the counterexample, a makes a partial broadcast which reaches b but not c. b then attempts to relay, but both messages are dropped. A support (falsifier: $O_{(A,C,1)} \vee O_{(C,B,2)}$) is highlighted.

for efficiently identifying events (likely or otherwise) that could cause trouble for a fault-tolerant program is to view protocol implementation as a game between a programmer and an adversary. In this section, we describe the LDFI approach as a repeated game (a match) in which an adversary tries to "beat" a protocol developer under a given system model. Of course, the end goal is for the developer to harden their protocol until it "can't lose"—at which point the final protocol can truly be called fault tolerant under the model. We show that a winning strategy for both the adversary (played by Molly) and the programmer is to is to use *data lineage* to reason about the *redundancy of support* (or lack thereof) for program outcomes.

To play a match, the programmer and the adversary must agree upon a correctness specification, inputs for the program and a failure model (for example, the adversary agrees to crash no more than one node, and to drop messages only up until some fixed time). In each round of the match the programmer submits a distributed program; the adversary, after running and observing the execution, is allowed to choose a collection of faults (from the agreed-upon set) to occur in the next execution. The program is run again under the faults, and both contestants observe the execution. If the correctness specification is violated, the programmer loses the round (but may play again); otherwise, the adversary is allowed to choose another set of failures and the round continues. If the adversary runs out of moves, the programmer wins the round.

A match: reliable broadcast protocols

For this match, the contestants agree upon reliable broadcast as the protocol to test, with the correctness specification shown in Listing 5.2. Input is a single record `bcast(A, data)@1`, along with a fully-connected `node` relation for the agents $\{A, B, C\}$—i.e., Node A attempts to broadcast the payload "**data**" to nodes B and C. The adversary agrees to inject message loss failures no later than logical time 2, and to crash at most one server (EFF=2, Crashes=1). In Figures 5.2-5.4,5.5 and 5.6, we represent the lineage of the outcomes (the final contents of `log`) as directed graphs, such that a record p has an edge to record q if p was used to compute q. Within each graph (which shows all *supports* for all outcomes), we highlight an individual support of the outcome (`log(B, data)@4`). Uncertain steps in the computations (i.e. messages) are shown as dashed lines. As we will see in Section 5.4, a message omission failure can be encoded with a Boolean variable of the form $O_{(Sender,Receiver,SenderTime)}$. For each lineage diagram, we show a *falsifier*: a propositional formula representing a set of failures that could invalidate the highlighted support. Let's play!

Round 1: naive broadcast

The programmer's first move is to submit the naive broadcast program **simple-deliv** presented in Listing 5.1. Figure 5.2 shows the lineage of three outcomes (the contents of the **log** relation on all nodes at time 4) for the failure-free execution of the program.

The adversary can use this representation of outcome lineage to guide its choice of failures to inject. To falsify an outcome (say `log(B, data)`) the adversary can simply "cut" the dotted line—that is, drop the message A sent to B at time 1. In the next execution, the adversary injects this single failure and the property expressed in Listing 5.2 is violated. The adversary wins the first round.

Round 2: retrying broadcast

The programmer was defeated, but she too can learn something from the lineage graph and the counterexample in Figure 5.2. The adversary won Round 1 easily because **simple-deliv** has no *redundancy*: the good outcome of interest is supported by a *single* message. A fault-tolerant program must provide redundant ways to achieve a good outcome; in the context of this game, one of those "ways" must be out of the reach of the adversary. The programmer makes an incremental improvement to **simple-deliv** by adding a rule for **bcast** that converts it from an ephemeral event true at one logical time to a persistent relation that drives retransmissions:

```
bcast(N, P)@next :- bcast(N, P);
```

Instead of making a single attempt to deliver each message, this program (henceforth called **retry-deliv**) makes an unbounded number of attempts. Intuitively, this alteration has made the reliable delivery protocol robust to a certain class of non-deterministic failures—namely, message omissions—by ensuring that messages exhibit *redundancy in time*.

Figure 5.3 shows the outcome lineage for an execution of **retry-deliv**. This time, while the adversary has more difficulty choosing a move, the winning strategy once again involves reasoning directly about outcome lineage. Since *A* makes an unbounded number of attempts [2] to send the `log` message to *B* and *C*, no finite pattern of omissions can falsify either outcome. The weakness of the **retry-deliv** algorithm is its asymmetry: the responsibility for redundant behaviors falls on *A* alone—this is easy to see in Figure 5.3, in which all transmissions originate at node *A*. The adversary, perceiving this weakness, might first attempt to immediately crash *A*. If it did so, this would result in a vacuous counterexample, since the delivery invariant is only violated if *some* but not all agents successfully delivery the message. In Section 5.4, we'll see how Molly avoids exploring such vacuously correct executions. However, causing *A* to crash after a successful transmission to one node (say *C*) but not the other is sufficient to falsify one outcome (`log(B, data)`). Exploring this potential counterexample via a concrete execution reveals the "true" counterexample shown in Figure 5.3. The adversary wins again.

Round 3: redundant broadcast

Reviewing the counterexample and the lineage of the failure-free execution, the programmer can see how the adversary won. The problem is that **retry-deliv** exhibits redundancy in time but not in space. Each broadcast attempt is independent of the failure of other attempts, but dependent on the fact that *A* remains operational. She improves the protocol further by adding another line:

```
bcast(N, P)@next :- log(N, P);
```

[2] These are finite executions, so the number of attempts is actually bounded by the *EOT* (4 in these figures) in any given execution. However, since the adversary has agreed to not drop messages after time 2, the program is guaranteed to make *more* attempts than there are failures.

Now *every* node that has a `log` entry assumes responsibility for broadcasting it to all other nodes. As Figure 5.5 reveals, the behavior of all nodes is now symmetrical, and the outcomes have redundant support both in space (every correct node relays messages) and time (every node makes an unbounded number of attempts). The adversary has no moves and forfeits; at last, the programmer has won a round.

Round 4: finite redundant broadcast

In all of the rounds so far, the adversary was able to either choose a winning move or decide it has no moves to make by considering a single concrete trace of a failure-free simulation. This is because the naive variants that the programmer supplied—which exhibit infinite behaviors—reveal all of their potential redundancy in the failure-free case. A practical delivery protocol should only continue broadcasting a message while its delivery status is unknown; a common strategy to avoid unnecessary retransmissions is acknowledgment messages. In Round 4, the programmer provides the protocol **ack-deliv** shown in Listing 5.3, in which each agent retries only until it receives an ACK.

The failure-free run of **ack-deliv** (Figure 5.6) exhibits redundancy in space (all sites relay) but not in time (each site relays a finite number of times and ceases before *EOT* when acknowledgments are received). The adversary perceives that it can falsify the outcome `log(B, data)` by dropping the message *A* sent to *B* at time 1, *and* either the message *A* sent to *C* at time 1 *or* the message *C* sent to *B* at time 2 (symbolically, $O_{(A,B,1)} \wedge (O_{(A,C,1)} \vee O_{(C,B,2)})$). It chooses this set of failures to inject, but in the subsequent run the failures trigger additional broadcast attempts—which occur when ACKs are not received—and provide additional support for the outcome. At each round the adversary "cuts" some edges and injects the corresponding failures; in the subsequent run new edges appear. Eventually it gives up, when the agreed-upon failure model permits no more moves. The programmer wins again.

Round 5: "classic" broadcast

For the final round, the programmer submits a well-known reliable broadcast protocol originally specified Birman et al. [132]:

```
(At a site receiving message m)
  if message m has not been received already
    send a copy of m to all other sites [...]
    deliver m [...]
```

This protocol is correct in the fail-stop model, in which processes can fail by crashing but messages are not lost. The programmer has committed a common error: deploying a "correct" component in an environment that does not match the assumptions of its correctness argument.

The classic broadcast protocol exhibits redundancy in space but not time; in a failure-free execution, it has infinite behaviors like the protocols submitted in Rounds 2-3, but this redundancy is vulnerable to message loss. The adversary, observing the lineage graph in Figure 5.4, immediately finds a winning move: drop messages from *A* to *B* at time 1, and from *C* to both *A and B* at time 2.

```
ack(S, H, P)@next :- ack(S, H, P);
rbcast(Node2, Node1, Pload)@async :- log(Node1, Pload),
    node(Node1, Node2), notin ack(Node1, Node2, Pload);
ack(From, Host, Pl)@async :- rbcast(Host, From, Pl);
rbcast(A, A, P) :- bcast(A, P);
log(N, P) :- rbcast(N, _, P);
```

Listing 5.3: Redundant broadcast with ACKs.

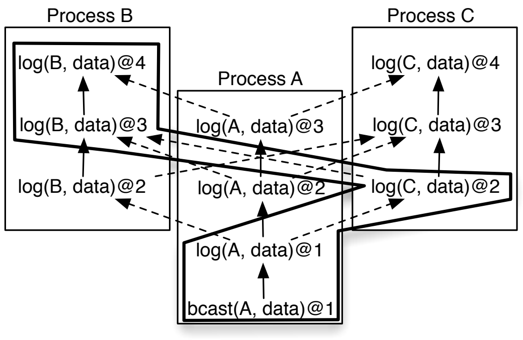


Figure 5.5: Lineage for the redundant broadcast protocol **redun-deliv**, which exhibits *redundancy in space and time*. A redundant support (falsifier: $O_{(A,C,1)} \vee O_{(C,B,2)}$) is highlighted. The lineage from this single failure-free execution is sufficient to show that no counterexamples exist.

Hazard analysis

In the game presented above, both players used the lineage of program outcomes to reason about their next best move. The role of the programmer required a certain amount of intuition: given the lineage of an outcome in a failure-free run and a counterexample run, change the program so as to provide more redundant support of the outcome. The role of the adversary, however, can be automated: Molly is an example of such an adversary.

Instead of randomly generating faults, the adversary used lineage graphs to surgically inject only those faults that could have prevented an outcome from being produced. As we observed in the game, data lineage from a single execution can provide multiple "supports" for a particular outcome. We saw evidence of this in the Kafka replication protocol as well: a stable write may be stable for multiple reasons, because it exists on multiple replicas, each of which may have received multiple transmissions—hence the lineage describing how the write got to each replica is a separate support, sufficient in itself to produce the outcome.

A lineage-driven fault injector needs to enumerate all of the supports of a target outcome, and devise a minimal set of faults (consistent with the failure model) that falsifies *all* of them. Because each individual support can be falsified by the loss of *any* of its contributing messages (a disjunction), LDFI can transform the graph representation into a CNF formula that is true if *all* supports are falsified (a conjunction of disjunctions) and pass it to an off-the-shelf SAT solver. Each satisfying assignment returned by the solver is a *potential counterexample*—it is sufficient to

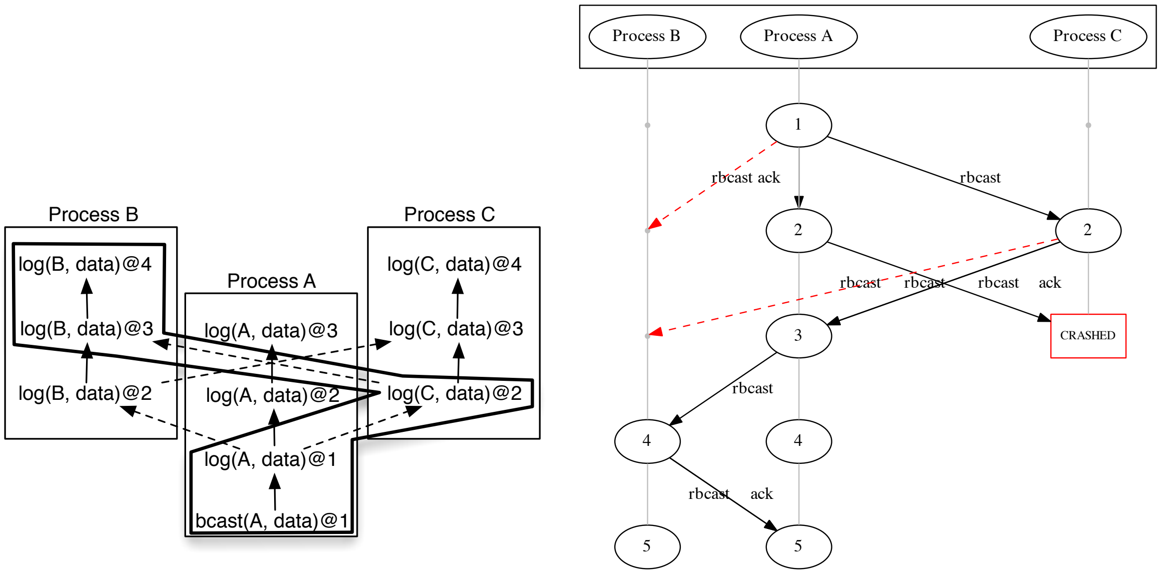


Figure 5.6: Lineage for the finite redundant broadcast protocol **ack-deliv**, which exhibits redundancy in space in this execution, but only reveals redundancy in time in runs in which ACKs are not received (i.e., when failures occur). The message diagram shows an unsuccessful fault injection attempt based on analysis of the lineage diagram: drop messages from A to B at time 1, from C to B at time 2, and then crash C. When these faults are injected, **ack-deliv** causes A to make an additional broadcast attempt when an ACK is not received.

falsify all the support of the outcome of which the lineage-driven fault injector is aware, given a particular concrete execution trace. Note that these potential counterexamples comprise the *only* faults that it needs to bother considering, precisely because if those faults do not occur it knows (because it has a “proof”) that the program will produce the outcome!

As we saw in the case of **ack-deliv**, given the faults in a potential counterexample the program under test may produce the outcome in some other way (e.g., via failover logic or retry). Molly converts the faults back into inputs, and performs at least one more forward evaluation step; this time, either the program fails to produce the outcome (hence we have a true counterexample) or it produces the outcome with new lineage (i.e., the program’s fault-tolerance strategy worked correctly), and we continue to iterate.

Molly automates this process. It collects lineage to determine how outputs are produced in a simulated execution, and transforms this lineage into a CNF formula that can be passed to a solver. If the formula is unsatisfiable, then no admissible combination of faults can prevent the output from being produced; otherwise, each satisfying assignment is a potential counterexample that must be explored. As we will see in Section 3.3, Molly’s forward / backward alternation quickly either identifies a bug or guarantees that none exist for the given configuration. When a bug is encountered, Molly presents visualizations of the outcome lineage and the counterexample trace, which as we saw are a vital resource to the programmer in understanding the bug and improving the fault-tolerance of the program.

5.4 The Molly system

In this section, we describe how we built the Molly prototype using off-the-shelf components including a Datalog evaluator and a SAT solver.

Program rewrites

To simulate executions of a Dedalus program, we translate it into a Datalog program that can be executed by an off-the-shelf interpreter. To model the temporal semantics of Dedalus (specifically state update and non-deterministic failure) we rewrite all program rules to reference a special *clock* relation. We also add additional rules to record the *lineage* or data provenance of the program's outputs.

Clock encoding and Dedalus rewrite

Due to the synchronous execution model and finiteness assumptions presented in Section 5.2, we can encode the set of transitions that occur in a particular execution—some of which may fail due to omissions or crashes—in a finite relation, and use this relation to drive the execution. We define a relation *clock* with attributes ⟨*From, To, SndTime*⟩. To model local state transitions in inductive (*@next*) rules, we ensure that *clock* always contains a record $\langle n, n, t \rangle$, for all nodes n and times $t < EOT$. Executions in which no faults occur also have a record $\langle n, m, t \rangle$ for all pairs of nodes n and m. To capture the loss of a message from node a to node b at time t (according to a's clock), we delete the record $\langle a, b, t \rangle$ from `clock`. To be consistent with the *EFF* parameter, this deletion can only occur if $t \leq EFF$. Finally, to model a fail-stop crash of a node a at time t, we simply delete records $\langle a, b, u \rangle$ for all nodes b and times $u \geq t$.

We also rewrite Dedalus rules into Datalog rules. For all relations, we add a new integer attribute $Time$ (as the last attribute), representing the logical time at which the record exists. To rewrite the premises of a rule, we modify each rule premise to reference the $Time$ attribute and ensure that all premises have the same $Time$ attribute and location specifier (this models the intended temporal semantics: an agent may make conclusions from knowledge only if that knowledge is available in the same place, at the same time). To rewrite the conclusion of a rule, we consider the Dedalus temporal annotations:

Example 8 *Deductive* rules have no temporal annotations and their intended semantics match that of Datalog, so they are otherwise unchanged.

```
log(Node, Pload) :- bcast(Node, Pload);
                 ↓
log(Node, Pload, Time) :- bcast(Node, Pload, Time);
```

Example 9 *Inductive* rules—which capture local state transitions—are rewritten to remove the *@next* annotation and to compute the value of $Time$ for the rule's conclusion by incrementing the $SndTime$ attribute appearing in the premises.

```
node(Node, Neighbor)@next :- node(Node, Neighbor);
                     ↓
node(Node, Neighbor, SndTime+1) :- node(Node, Neighbor, SndTime),clock(Node, Node,
  SndTime);
```

Example 10 *Asynchronous* rules—representing uncertain communication across process boundaries—are rewritten in the same way as inductive rules. Note however that because the values of To and $From$ are distinct, the transition represented by a matching record in *clock* might be a failing one (i.e., it may not exist in `clock`).

```
log(Node2, Pload)@async :-
    bcast(Node1, Pload),
    node(Node1, Node2);
                ↓
log(Node2, Pload, SndTime+1) :-
    bcast(Node1, Pload, SndTime),
    node(Node1, Node2, SndTime),
    clock(Node1, Node2, SndTime);
```

Lineage rewrite

In order to interpret the output of a concrete run and reason about fault events that could have prevented it, we record per-record data lineage capturing the computations that produced each output.

We follow the *provenance-enhanced rewrite* described by Kohler et al [99]. For every rule, the rewrite produces a new "firings" relation that captures bindings used in the rule's premises.

For every rule r in the given (rewritten from Dedalus as described above) Datalog program, we create a new relation r_{prov} (called a "firings" relation) and a new rule r', such that

1. r' has the same premises as r,
2. r' has r_{prov} as its conclusion, and
3. r_{prov} captures the bindings of all premise variables.

For example, given the asynchronous rule in Example 10, Molly synthesizes a new rule:

```
log1$_{prov}$(Node1, Node2, Pload, SndTime) :-
    bcast(Node1, Pload, SndTime),
    node(Node1, Node2, SndTime),
    clock(Node1, Node2, SndTime);
```

Rules with aggregation use two provenance rules, one to record variable bindings and another to perform aggregation. This prevents the capture of additional bindings from affecting the grouping attributes of the aggregation. For example:

```
r(X, count<Z>) :- a(X, Y), b(Y, Z)
                  ↓
r$_{bindings}$(X, Y, Z) :- a(X, Y), B(Y, Z)
r$_{prov}$(X, count<Z>) :- r$_{bindings}$(X, _, Z)
```

Proof tree extraction

We query the firings relations to produce *derivation graphs* for records [165]. A derivation graph is a directed bipartite graph consisting of `rule` nodes that correspond to rule firings, and `goal` nodes that correspond to records. There is an edge from each goal node to every rule firing that derived that tuple, and an edge from each rule firing to the premises (goal nodes) used by that rule. The lineage graphs in Section 5.3 (Figures 5.2-5.6) are abbreviated derivation graphs, in which goal nodes are represented but rule nodes are hidden.

To construct the derivation graph for a record r, we query the firings relations for rules that derive r. For each matching firing, we substitute the bindings recorded in the firing relation into the original rule to compute the set of premises used by that rule firing, and recursively compute the derivation graphs for each of those premises. Note that each rule node represents a firing of a rule with a particular set of inputs; it is possible for a single outcome to have multiple derivations via the same rule, each involving different premises.

Each derivation graph yields a finite forest of *proof trees*. Each proof tree corresponds to a separate, independent support of the tree's root goal (i.e., its outcome). Given a proof tree, we can determine which messages were used by the proof's rule firings; the loss of any of these messages will falsify that particular proof.

Solving for counterexamples

Given a forest of proof trees, a naive approach to enumerating potential counterexamples is to consider all allowable crash failure and message omissions that affect messages used by proofs. This quickly becomes intractable, since the set of fault combinations grows exponentially with *EFF*.

Instead, we use the proof trees to perform a SAT-guided search for failure scenarios that falsify *all* known proofs of our goal tuples. For each goal, we construct a SAT problem whose variables encode crash failures and message omissions and whose solutions correspond to faults that falsify all derivations in the concrete execution.

Each proof tree is encoded as a disjunction of the message omissions ($O_{(from,to,time)}$) and crash failures ($C_{(node,time)}$) that can individually falsify the proof. By taking the conjunction of these formulas, we express that we want solutions that falsify all derivations. For example,

$$(O_{(a,c,2)} \vee C_{(a,2)} \vee C_{(a,1)}) \wedge (O_{(b,c,1)} \vee C_{(b,1)})$$

corresponds to a derivation graph that represents two proofs, where the first proof can be falsified by either dropping messages from a to c at time 2 or by a crashing at some earlier time, and the second proof can be falsified by b crashing or the loss of its messages sent at time 1.

In addition, we translate the failure constraints into SAT clauses such that messages sent after the *EFF* cannot be lost, at most *MaxCrashes* nodes can crash, and nodes can only crash once.

If the resulting SAT problem is unsatisfiable, then there exists at least one proof that cannot be falsified by any allowable combination of message losses and crash failures—hence the program is fault-tolerant with respect to that goal! Otherwise, each SAT solution represents a potential counterexample that must be explored.

We solve a separate SAT problem for each goal tuple, and the union of the SAT solutions is the set of potential counterexamples that we must test—each potential counterexample corresponds to a "move" of the adversary in the game presented in Section 5.3. If the user has defined correctness properties using the built-in `pre` and `post` meta-outcomes defined in Section 5.3, we perform an additional optimization. Each meta-outcome is handled as above, and a potential counterexample is reported for each set of faults that falsifies a record in `post` *unless* those faults also falsify the corresponding record in `pre`. We need not explore such faults, as they would result in a vacuously correct outcome with respect to that property.

5.5 Completeness of LDFI

Algorithm 1 LDFI

```
Require: $\mathcal{P}$ is a Datalog¬ program produced by rewriting a Dedalus program
Require: $\mathcal{E}$ is an EDB including clock facts
Require: $g \in \mathcal{P}(\mathcal{E})$ is a goal fact
function LDFI($\mathcal{P}, \mathcal{E}, g$)
    $\mathcal{R} \leftarrow$ provenance-enhanced rewrite of $\mathcal{P}$
    $\mathcal{G} \leftarrow RGG(\mathcal{R}, \mathcal{E}, g)$
    $\varphi \leftarrow \texttt{clocks}(g, \mathcal{G})$
    if $\varphi$ is satisfiable then
        while there are more satisfying models of $\varphi$ do
            $\mathfrak{A} \leftarrow$ the next model of $\varphi$
            $D \leftarrow \{\texttt{clock}(f,t,l) \in \mathcal{E} | \mathfrak{A} \models O_{f,t,l} \lor (\mathfrak{A} \models C_{f,l'} \land l' < l)\}$
            if $g \notin \mathcal{P}(\mathcal{E} \setminus D)$ then
                Yield $D$
            end if
        end while
    else
        return $\emptyset$
    end if
end function
```

LDFI provides a *completeness* guarantee unattainable by other state-of-the-art fault injection techniques: if a program's correctness specification can be violated by some combination of failures (given a particular execution bound), LDFI discovers that combination; otherwise it reports that none exists. In this section, we provide a formalization and proof for that claim.

Section 5.4 described how a (distributed) Dedalus program can be rewritten into a fragment of Datalog¬, whose execution may be simulated by an off-the-shelf Datalog evaluator. Given two Datalog relations p and q, we write $p \xrightarrow{\mathcal{P}} q$ if and only if there exists a rule r in $\mathcal{P}$ such that p is the relation of a subgoal in r and q is the relation in the head (we omit the $\mathcal{P}$ when the context is clear). We write $\xrightarrow{\mathcal{P}}^{+}$ to denote the transitive closure of $\xrightarrow{\mathcal{P}}$. We assume that submitted Dedalus programs are *stratifiable* [165]; that is, no predicates depend negatively on themselves, directly or transitively. It is easy to see that the rewrite procedure produces stratified Datalog¬ programs. In particular, the only change that rewriting makes is to introduce the clock relation on the right-hand-side of some rules. Since clock never appears on the left-hand-side of a rule, the rewrite does not introduce any new transitive cyclic dependencies.

A *fact* is a predicate symbol all of whose arguments are constants; e.g., `log(A, ‘‘data’’)`. We write `relation(f)` to indicate the predicate name of a fact f. For example, if f = `log(A, ‘‘data’’)` then `relation`(f) = `log`. We are specifically interested in the set of EDB facts $C \equiv \{c \in \mathcal{E} | \texttt{relation}(c) = \texttt{clock}\}$ Each such fact $c \in C$ (henceforth called *clock facts*) is of the form `clock(from, to, time)` and intuitively represents connectivity from computing node *from* to node *to* at time *time* on *from*’s local clock. Recall that in the Dedalus to Datalog¬ rewrite, every “asynchronous” rule is rewritten to include `clock` as a positive subgoal. For convenience we use a named field notation (as in SQL): given a clock fact c we write c.`from` to indicate the value in the first column of c; similarly with `to` and `time` (the second and third columns, respectively).

Given a stratifiable Datalog¬ program $\mathcal{P}$ and an *extensional database* (EDB) $\mathcal{E}$ (a set of base facts comprising the program’s input), we write $\mathcal{P}(\mathcal{E})$ to denote the (unique) perfect model of $\mathcal{P}$ over $\mathcal{E}$. The model $\mathcal{P}(\mathcal{E}) \supseteq \mathcal{E}$ is itself a set of facts. Lineage analysis operates over a derivation graph [99, 165], a bipartite *rule/goal* graph $\mathcal{G} = (R \cup G, E)$, where G is a set of *goal* facts and R is a set of rule firings [99]. An edge $(x, y) \in E$ associates either

1. a goal x with a rule y used to derive it, or

2. a rule firing x with a subgoal y that provided bindings.

We write `RGG`($\mathcal{P}, \mathcal{E}$) to represent the rule/goal graph produced by executing program $\mathcal{P}$ over input $\mathcal{E}$.

LDFI constructs boolean formulae and passes them to a SAT solver. Given a model $\mathfrak{A}$ returned by a solver and a formula φ, we write $\mathfrak{A} \models \varphi$ if φ is true in $\mathfrak{A}$. We are concerned with the truth values of a set of propositional variables $\{O_{from,to,time} \cup C_{from,time}\}$ such that `from,to` are drawn from the domain of locations (the first attribute of every fact, and in particular the `clock` relation) and `time` consists of integers less than EOT. Recall that $O_{from,to,time}$ represents message loss from node `from` to node `to` at time `time`, while $C_{from,time}$ represents a (permanent) crash failure of node `from` at time `time`.

We model failures in distributed systems as deletions from the EDB clock relation. By construction, such deletions affect precisely the (transitive) consequences of *async* Dedalus rules, and match intuition: message loss is modeled by individual deletions (a loss of connectivity between two endpoints at a particular logical time), while crash failures are modeled by batch deletions (loss of connectivity between some endpoint and all others, from a particular time onwards).

Definition A *fault set* is a set of facts $D \equiv \{f \in \mathcal{E} | \texttt{relation}(f) = \texttt{clock}\}$

Given a program $\mathcal{P}$, an EDB $\mathcal{E}$ and a distinguished set of "goal" relation names $\mathcal{G}$ (in the common case, $\mathcal{G} \equiv \{\texttt{post}\}$), we identify a set of goal facts $\mathcal{F} = \{g \in \mathcal{P}(\mathcal{E}) | \texttt{relation}(g) \in \mathcal{G}\}$. For each goal fact $g \in \mathcal{F}$, we wish to know whether there exists a fault set which, if removed from $\mathcal{E}$, prevents $\mathcal{P}$ from producing g.

Definition A *falsifier* of a goal fact g is a fault set D such that $g \in \mathcal{P}(\mathcal{E})$, but $g \notin \mathcal{P}(\mathcal{E} \setminus \mathcal{D})$. A falsifier is *minimal* if there does not exist a falsifier D' of g such that $D' \subset D$.

LDFI identifies potential falsifiers of program goals by inspecting the data lineage of concrete executions. We may view a lineage-driven fault injector as a function from a program, an EDB and a goal fact to a set of fault sets; we write $\texttt{LDFI}(\mathcal{P}, \mathcal{E}, g)$ to denote the (possibly empty) set of fault sets that LDFI has determined could falsify a goal fact g produced by applying the Datalog¬ program $\mathcal{P}$ to the EDB $\mathcal{E}$.

A lineage-driven fault injector is *sound* if, when $D \in \texttt{LDFI}(\mathcal{P}, \mathcal{E}, g)$, D is indeed a falsifier of g. Soundness is trivially obtained by the forward/backward execution strategy: for any potential counterexamples, LDFI performs a concrete execution (Algorithm 1, Line 9) to determine if the set of omissions constitutes a correctness violation, and outputs D only if $g \notin \mathcal{P}(\mathcal{E} \setminus D)$.

To prove completeness, we present the LDFI system discussed in Section 5.4 formally in Algorithm 1. Most of the work of LDFI is performed by the recursive function `clocks` defined in Algorithm 2, which operates over derivation graphs and returns a boolean formula whose satisfying models represent potential counterexamples. Given a node n in the graph $\mathscr{G}$ (either a rule node or a goal node, as described above), `clocks` returns a formula whose satisfying models intuitively represent faults that could prevent n from being derived. If n is a clock fact (Line 4 of Algorithm 2), then `clocks` returns a disjunction of boolean variables representing conditions (losses and crashes) that could remove this fact from the EDB; if n is a non-clock leaf fact, `clocks` simply returns `true` (Line 7). Otherwise, n is either a rule node or a non-leaf goal. If n is a rule node, then all of its child (goal) nodes were required to cause the rule to fire; invalidating *any* of them falsifies the rule—hence to invalidate n, we take the disjunction of the formulae that invalidate its children (Line 24). By contrast, if n is a non-leaf positive goal, then each of its (rule) children represents an alternative derivation of n via different rules—to invalidate n, we must invalidate *all* of its alternative derivations: hence we consider the conjunction of the formulae that invalidate its children (Line 20).

The last case to consider is if n is a negative goal: that is, some rule fired because (among other reasons) a particular fact did *not* exist (e.g., a retry was triggered because a timeout fired and there was *no* log of an acknowledgment message). The derivation graph $RGG(\mathcal{P}, \mathcal{E})$ does not explicitly represent the reasons why a particular tuple does not exist. There are a variety of options for exploring why-not provenance, as we discuss in the related work. The Molly prototype currently offers three alternatives to users. The first is to ignore the provenance of negated goals—this is clearly acceptable for monotonic programs, and can be useful for quickly identifying bugs, but is incomplete. The second is similar to the approach used by Wu et al. [168] to debug software-defined networks, which uses surrogate tuples to stand in for facts that do *not* hold at a particular time and location. This approach seems to work well in practice. Finally, we support an optimized version of the conservative approach described in detail in the completeness proof below. We consider as a possible cause of a negated goal tuple g any tuple t in the model $\mathcal{P}(\mathcal{E})$ for which two conditions hold:

1. $\texttt{relation}(t) \overset{\mathcal{P}}{\rightarrow}^{+} \texttt{relation(g)}$ (i.e., the relation containing g is reachable from the relation containing t) via an *odd* number of negations (based on static analysis of the program), and

2. the timestamp of t is less than or equal to the timestamp of g.

For the purposes of the proof we consider a conservative over-approximation of the set of possible causes for the nonexistence of a fact. Line 17 enumerates the set of (positive) facts z such that $\texttt{relation}(n)$ is reachable from $\texttt{relation}(z)$. `clocks` then invokes itself recursively and returns the disjunction of the falsifying formulae for all such goals r. The intuition is that since we do not know the exact reason why a fact does not exist, we over-approximate the set of possible causes by considering any fact z that *could* reach n (based on a static analysis of the dependency relation $\rightarrow$) as one which, if made false, could cause n to appear and falsify a derivation that required n to be absent. The attentive reader will observe that falsifying z can only make n true if n depends *negatively* on z—therefore we could further constrain the set of facts enumerated in Line 17 of Algorithm 2 to include only those z from which n is reachable via an odd number of negations. Because LDFI is sound, it is always safe to over-approximate the set of possible falsifiers, so for simplicity of presentation we omit this optimization from the proof.

We first establish a lemma regarding the behavior of Algorithm 2.

Lemma 5.5.0.1. *Given a program $\mathcal{P}$, EDB $\mathcal{E}$, their derivation graph $\mathcal{G} = RGG(\mathcal{P}, \mathcal{E})$, and a goal fact $g \in \mathcal{P}(\mathcal{E})$, if D is a minimal falsifier of g, then there exists a model $\mathfrak{A}$ of the boolean formula $\texttt{clocks}(g, \mathcal{G})$ such that for every $f \in D$, either $\mathfrak{A} \models O_{f.from, f.to, f.time}$ or $\mathfrak{A} \models C_{f.from, t}$ for some t such that $t \leq f.time$.*

Proof. Proof is by induction on the structure of $\mathcal{G}$.

Base case: g is a leaf goal. We assume the antecedent: D is a minimal falsifier of g. Consider any $f \in D$. Because D is minimal, it must be the case that without f, g cannot be derived by $\mathcal{P}$ over $\mathcal{E}$. So if g is a leaf goal, it must be the case that $g = f$. So $\texttt{clocks}(g, \mathcal{G}) = O_{f.\texttt{from}, f.\texttt{to}, f.\texttt{time}} \vee \bigvee_{t=0}^{f.\texttt{time}}(C_{f.\texttt{from}, t})$ and its satisfying models are exactly those that make true either $O_{f.\texttt{from}, f.\texttt{to}, f.\texttt{time}}$ or any $C_{f.\texttt{from}, t}$ with $t \leq f.\texttt{time}$.

Inductive case 1: g is a rule. By the inductive hypothesis, Lemma 5.5.0.1 holds for all subformulae $\texttt{clocks}(g', \mathcal{G})$ such that $(g, g') \in E$. Line 24 returns the disjunction of the subformulae; since Lemma 5.5.0.1 for each, it surely also holds for their disjunction (any model of one of the subformulae is a model of the whole disjunction).

Inductive case 2: g is a non-leaf goal. We consider two cases:

If g is positive, consider all subformulae $\texttt{clocks}(r, \mathcal{G})$ such that $(g, r) \in E$—call those subformulae $\varphi, \psi, [\ldots]$. By the inductive hypothesis, there exist models $\mathfrak{A}, \mathfrak{B}, [\ldots]$ such that $\mathfrak{A} \models \varphi, \mathfrak{B} \models \psi, [\ldots]$, and for all $f \in D$, either $\mathfrak{A} \models O_{f.\texttt{from}, f.\texttt{to}, f.\texttt{time}}$ or $\mathfrak{A} \models C_{f.\texttt{from}, t}$) for some t such that $t \leq f.time$, and similarly for $\mathfrak{B}$, $[\ldots]$, etc.

We must now show that there necessarily exists a model $\mathfrak{Z}$ such that $\mathfrak{Z} \models \varphi \wedge \psi \wedge [\ldots]$ and for all $f \in D$, either $\mathfrak{Z} \models O_{f.\texttt{from},f.\texttt{to},f.\texttt{time}}$ or $\mathfrak{Z} \models C_{f.\texttt{from},t}$). Note that by construction, $\varphi, \psi, [\ldots]$ contain only the boolean connectives and ($\wedge$) and or ($\vee$): in particular, they do not contain negation. Hence it cannot be the case that their conjunction is unsatisfiable. We construct $\mathfrak{Z}$ by making true every propositional variable that is true in any of the $\mathfrak{A}, \mathfrak{B}, [\ldots]$. Observe that $\mathfrak{Z} \models \varphi$, since all variables true in $\mathfrak{A}$ are true in $\mathfrak{Z}$ and φ does not contain negation—similarly for $\psi, [\ldots]$. Hence $\mathfrak{Z} \models \varphi \wedge \psi \wedge [\ldots]$.

Finally, if g is negative, then C (Line 17) is the enumeration of the $z \in G$ such that g is (statically) reachable from z based on $\overset{\mathcal{P}}{\rightarrow}^{+}$. Note that if g has only positive support (no predicate z' such that $z' \overset{\mathcal{P}}{\rightarrow}^{+} g$ appears as a negative subgoal in a rule), then no falsifiers of "not g" exist—EDB deletions cannot cause new facts to appear except in the presence of negation—and so the lemma holds vacuously. However, it is possible that g depends *negatively* (specifically, via an odd number of negations) on some (positive) z: if z were to disappear, then g could be derived. C over-approximates this set of positive facts using the (static) relation $\overset{\mathcal{P}}{\rightarrow}^{+}$; for each, `clock` obtains a falsifying formula for z (which by the inductive hypothesis satisfies the lemma). If a falsifier of g exists, it must be a falsifier of one of the z. Hence (by a similar argument to the inductive case 1 above) the lemma holds for the disjunction of the $z \in C$s. □ □

Theorem 5.5.1. ***Completeness of LDFI**: Given a program $\mathcal{P}$, and EDB $\mathcal{E}$, for every minimal falsifier D of goal fact $g \in \mathcal{P}(\mathcal{E})$, there exists a $D' \in \mathit{LDFI}(\mathcal{P}, \mathcal{E}, g)$ such that D' is a falsifier of g and $D \subseteq D'$.*

Proof. By Lemma 5.5.0.1, there is a model $\mathfrak{A}$ of the boolean formulae denoted by $\texttt{clocks}(g, \mathscr{G})$ such that, for every $f \in D$, either $\mathfrak{A} \models O_{f.\texttt{from},f.\texttt{to},f.\texttt{time}}$ or $\mathfrak{A} \models O_{f.\texttt{from},t}$ for some t such that $t \leq f.time$. Algorithm 1 enumerates all satisfying models: one of them is $\mathfrak{A}$. As seen in Line 8, for every $f \in D'$ there is a fact $\texttt{clock}(f.\texttt{from}, f.\texttt{to}, f.\texttt{time})$ in the falsifier corresponding to model $\mathfrak{A}$ returned by LDFI. □ □

5.6 Evaluation

In this section, we use Molly to study a variety of fault-tolerant protocols from the database and distributed systems literature, as well as components of modern systems. We then measure the performance of Molly along two axes: its efficiency in discovering bugs, and its coverage of the combinatorial space of faults for bug-free programs.

Algorithm 2 Clocks algorithm

```
Require: $\mathscr{G} = (R \cup G, E)$ is a bipartite rule-goal graph.
Require: $n \in (R \cup G)$
 1: function CLOCKS($n, \mathscr{G}$)
 2:     if $n \in G$ then                                                ▷ $n$ is a goal
 3:         if $\neg\exists r(r \in R \wedge (n, r) \in E)$ then                        ▷ $n$ is a leaf
 4:             if $\texttt{relation}(n) = \texttt{clock}$ then
 5:                 if $n.\texttt{time} < EFF$ then
 6:                     $\varphi \leftarrow$ new bool: $O_{n.\texttt{from},n.\texttt{to},n.\texttt{time}}$
 7:                 else
 8:                     $\varphi \leftarrow$ false
 9:                 end if
10:                 $\psi \leftarrow \bigvee_{i \leftarrow 0}^{n.\texttt{time}}$ new bool:$C_{n.\texttt{from},i}$
11:                 return $(\varphi \vee \psi)$
12:             else
13:                 return true                                ▷ Ignore non-clock leaves
14:             end if
15:         else                                                   ▷ $n$ is a non-leaf goal
16:             if $n$ is negative then                            ▷ $n$ was a negated subgoal
17:                 $C \leftarrow \{z \in G | \texttt{relation}(z) \overset{\mathcal{P}}{\rightarrow}^{+} \texttt{relation}(n)\}$
18:                 return $\bigvee_{z \in C} clocks(z, \mathscr{G})\}$
19:             else
20:                 return $\bigwedge_{(n,r) \in E} clocks(r, \mathscr{G})\}$
21:             end if
22:         end if
23:     else if $n \in R$ then                                          ▷ $n$ is a rule
24:         return $\bigvee_{(n,g) \in E} clocks(g, \mathscr{G})\}$
25:     end if
26: end function
```

Case Study: Fault-tolerant protocols

We implemented a collection of fault-tolerant distributed programs in Dedalus, including a family of reliable delivery and atomic commitment protocols and the Kafka replication subsystem described in Section 5.1. We analyze them with Molly and describe the outcomes.

Molly automatically produces Lamport diagrams [104], like those shown in Section 5.3, to help visualize the message-level behavior of individual concrete executions and to enable counterexample-driven debugging. In each diagram, solid vertical lines represent individual processes; time moves from top to bottom. Messages between processes are shown as diagonal lines connecting process lines; lost messages are shown as dashed lines. Vertices represent events and are numbered to reflect global logical time; if a process crashes, its node contains the string "CRASHED." When it comes time to debug systems to discover (and ultimately remedy) the *cause* of an invariant violation, Molly produces lineage diagrams similar to those shown in Section 5.3.

Commit Protocols

We used Dedalus to implement three commit protocol variants from the database literature, which were developed and extended over a period of about five years [34, 72, 155]. As we would hope, Molly immediately confirmed the known limitations of these protocols, and produced concrete counterexamples both for non-terminating executions and for executions in which conflicting decisions are made.

For all commit protocols, we specify two invariants as implications between pre- and postconditions:

- **Agreement**: *If* an agent decides to commit (respectively, abort), *then* all agents decide to commit (abort).
- **Termination**: *If* a transaction is initiated, *then* all agents decide either commit or abort for that transaction.

Molly automatically identified the limitations of early commit protocols that subsequent work attempted to correct. Figure 5.7 illustrates the well-known blocking problem associated with two-phase commit (2PC) [72]. If the coordinator process fails after preparing a transaction, the system is left in a state in which the transaction outcome is unknown but all agents are holding locks waiting for the outcome (a violation of the *termination* property).

The collaborative termination protocol (CTP) [34] attempts to ameliorate the blocking problem by augmenting the 2PC protocol so as to allow agents who suspect that the coordinator has failed to exchange their knowledge about the outcome. It is well-known, however, that although CTP allows *more* executions to terminate, it has blocking executions under the same failure assumptions as classic 2PC. Figure 5.8 shows a counterexample that Molly discovered after a single forward/backward execution—due to space limitations, this diagram can be found in the Appendix.

Three-phase commit [155] solves the blocking problem—under the assumption of a connected and synchronous network—by adding an additional protocol round and corresponding agent state.

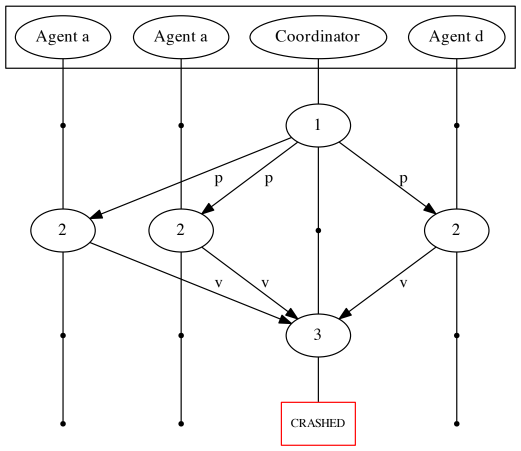


Figure 5.7: A blocking execution of 2PC. Agents a, b and d successfully prepare (**p**) and vote (**v**) to commit a transaction. The coordinator then fails, and agents are left uncertain about the outcome—a violation of the **termination** property.

It uses simple timeouts as a failure detector; depending on the state a coordinator or agent is in when a timeout fires, that site can unilaterally determine the transaction outcome. Hence there are no "blocking" states.

If we relax the assumption of a connected network by allowing finite message failures, however, MOLLY discovers bad executions such as the one shown in Figure 5.9. In this case, message losses from the coordinator to certain agents (a and b) cause the agents to conclude that the coordinator has failed. Since they are in the `canCommit` state, they decide to roll forward to commit. Meanwhile the coordinator—which has detected that agent d (who originally agreed to commit) has failed—has decided to abort. This outcome is arguably worse than blocking: due to the incorrectness of the failure detector under message omissions, agents have now made conflicting decisions, violating the *agreement* property.

As we saw in the case of **classic-deliv** in Section 5.3, the bad execution results from deploying a protocol in an environment that violates the assumptions of its correctness guarantee—an all-too-common hazard in composing systems from individually-verified components.

Other fault-tolerant protocols

We used MOLLY to study other agreement protocols, including Paxos [106] and the bully leader election protocol [65]. As we discuss in Section 5.8, desirable termination properties of such protocols are difficult to verify due to their sensitivity to asynchrony. Nevertheless we are able to validate their agreement properties by demonstrating that no counterexamples are found for reasonable parameters (as noted in Table 5.2).

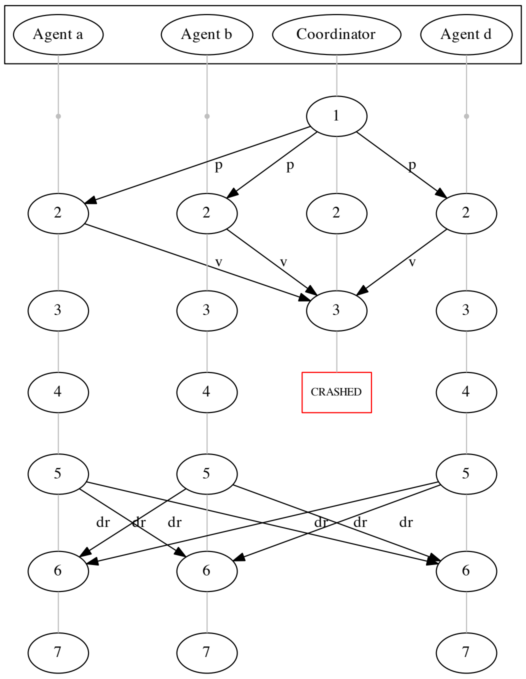


Figure 5.8: A blocking execution of 2PC-CTP. As in Figure 5.7, the coordinator has crashed after receiving votes (**v**) from all agents, but before communicating the commit decision to any of them. The agents detect the coordinator failure with timeouts and engage in the collaborative termination protocol, sending a *decision request* message (**dr**) to all other agents. However, since all agents are uncertain of the transaction outcome, none can decide and the protocol blocks, violating **termination**.

Flux [152] is a replica synchronization protocol for streaming dataflow systems based on the process pairs [74] fault-tolerance strategy. Flux achieves fault-tolerance by ensuring that a pair of replicas receives the same message stream without loss, duplication or reordering; at any time, should one replica fail, the other can take over. Despite its succinct specification, Flux is considered to be significantly more complicated than alternative fault-tolerance strategies for streaming systems, because of the interaction between the protocol's granularity (tuple-at-a-time) and the various combinations of failures that can occur during operation [174]. Using MOLLY, we were able to certify that Flux is resilient to omission and crash failures up to a significant depth of execution (see Table 5.2). To the best of our knowledge, this effort represents the most thorough validation of the Flux protocol to date.

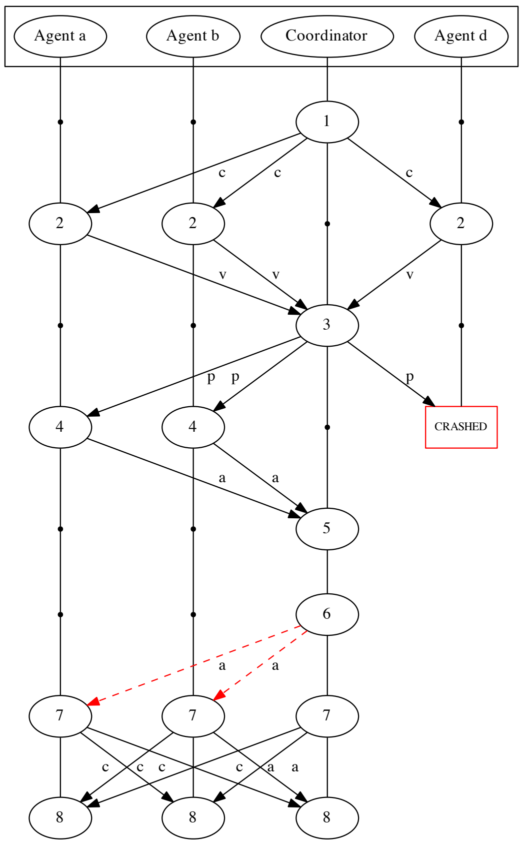


Figure 5.9: An incorrect run of three-phase commit in which message loss causes agents to reach conflicting decisions (the coordinator has decided to abort (**a**), but the two (non-crashed) agents have decided to commit (**c**)). This execution violates the **agreement** property.

Kafka replication bug

To reproduce the Kafka replication bug described in Section 5.1, we provide a single durability invariant:

- **Durability**: *If* a write is acknowledged at the client, *then* it is stored on a correct (non-crashed) replica.

MOLLY easily identified the bug. Figure 5.10 shows that MOLLY found the same counterexample described by Kingsbury. A brief network partition isolates two of the replicas (*b* and *c*) from *a* and the Zookeeper service, and *a* is elected as the new leader (assuming it is not the leader already). Because *a* believes itself to be the sole surviving replica, it does not wait for acknowledgments before acknowledging the client. Then *a* crashes, violating the durability requirement.

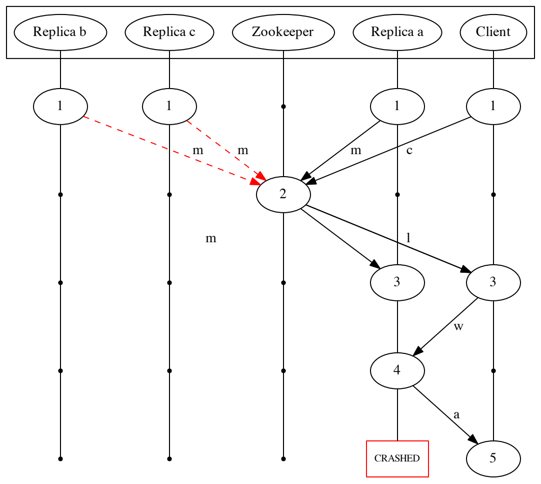


Figure 5.10: The replication bug in Kafka. A network partition causes *b* and *c* to be excluded from the ISR (the *membership* messages (**m**) fail to reach the Zookeeper service). When the client writes (**w**) to the leader *a*, it is immediately acknowledged (**a**). Then *a* fails and the write is lost—a violation of **durability**.

In reproducing this durability bug, we relied heavily on MOLLY's ability to model different components of a large-scale system at different levels of specificity. We focused first on the primary/backup replication protocol logic, which we implemented in significant detail (roughly a dozen LOC in Dedalus). Based on the intuition that the bug lay at the boundary of the replication protocol and the Zookeeper service and not in the service itself, we sketched the Zookeeper component and the client logic, ignoring details such as physical distribution (we treat the Zookeeper cluster as a single abstract node) and the underlying atomic broadcast protocol [91]. Had this model failed to identify a bug, we could subsequently have enriched the sketched specifications.

Measurements

A lineage-driven fault injector must do two things efficiently: identify bugs or provide a bounded guarantee about their absence. In this section, we measure MOLLY's efficiency in finding counterexamples for 7 buggy programs, compared to a random fault injection approach. We then measure how quickly it *covers* the combinatorial space of possible faults for bug-free programs.

Table 5.1 lists the buggy protocols and (minimal) parameter settings for which MOLLY found a counterexample, The table lists the protocol size (in lines of code), the number of concrete executions explored before the counterexample was found, and the elapsed time. In order to factor apart the impact of the abstractions presented in Section 5.2 from that of the pruning performed by hazard analysis, we also implemented a random fault injector for MOLLY. In random mode, MOLLY chooses failure combinations at random and avoids the overhead of SAT and lineage extraction.

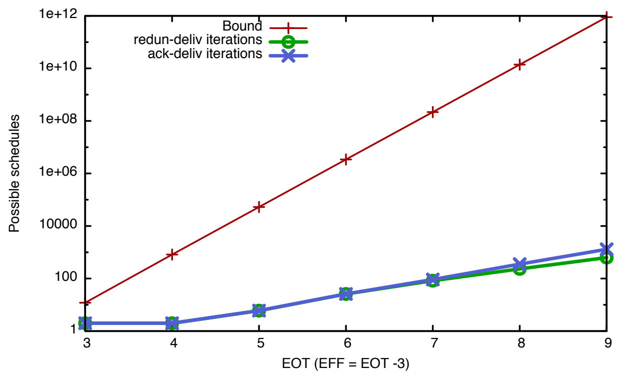


Figure 5.11: For the **redun-deliv** and **ack-deliv** protocols, we compare the number of concrete executions that Molly performed to the total number (Combinations) of possible failure combinations, as we increase *EOT* and *EFF*.

Table 5.1 also shows the average number of executions (*exe*) and average execution time (*wall*, in seconds) before the random fault injector discovered a counterexample (averages are from 25 runs).

The performance of random fault injection reveals the significance of LDFI's abstractions in reducing the search space: for all of the (buggy) protocols we studied, random fault injection eventually uncovered a counterexample. In the case of relatively shallow failure scenarios, the random approach performed competitively with the hazard analysis-based search. In more complex examples—in particular the Kafka bug—Molly's hazard analysis outperforms random fault injection by an order of magnitude.

Even more compelling than its rapid discovery of counterexamples is Molly's reduction of the search space for bug-free programs. A random strategy may find certain bugs quickly, but to guarantee that no bugs exist (given a particular set of parameters) requires exhaustive enumeration of the space of executions, which is exponential both in *EFF* and in the number of nodes in the simulation. By contrast, Molly's hazard analysis is guaranteed to discover and explore only those failures that could have invalidated an outcome. Table 5.2 compares the space of possible executions (**Combinations**) that would need to be explored by an exhaustive strategy to the number of concrete executions (**exe**) performed by Molly (providing 100% coverage of the relevant execution space), for a number of bug-free protocol implementations. In all cases, we report the maximum parameter values reached by the sweep procedure given a 120 second time bound.

Figure 5.11 plots the growth in the number of concrete executions considered as the *EFF* is increased, for the **ack-deliv** and **redun-deliv** protocols presented in Section 5.3, against the upper bound (the number of possible failure combinations for that Fspec) on a log-linear scale. It illustrates the impact of redundancy in individual executions on the pruning strategy. By revealing massive redundancy in every run, **redun-deliv** protocol allows Molly to rule out an exponentially larger set of potential counterexamples in each backward step.

Program	*Counterexample*	*LOC*	*EOT*	*EFF*	*Crashes*	*Combinations*	*Random*		*Molly*	
							exe	*wall*	*exe*	*wall*
simple-deliv	Figure 5.2	4	4	2	0	4.10×10^{03}	4.08	0.16	2	0.12
retry-deliv	Figure 5.3	5	4	2	1	4.07×10^{04}	75.24	1.28	3	0.12
classic-deliv	Figure 5.4	5	5	3	0	2.62×10^{05}	116.16	1.81	5	0.24
2pc	Figure 5.7	16	5	0	1	24	5.48	0.31	2	0.22
2pc-ctp	Figure 5.8	25	8	0	1	36	8.56	1.04	3	1.01
3pc	Figure 5.9	24	9	7	1	2.43×10^{26}	40.60	6.24	55	9.60
Kafka	Figure 5.10	18	6	4	1	1.85×10^{25}	1183.12	133.30	38	3.74

Table 5.1: MOLLY finds counterexamples quickly for buggy programs. For each verification task, we show the minimal parameter settings (*EOT, EFF* and *Crashes*) to produce a counterexample, alongside the number of possible combinations of failures for those parameters (**Combinations**), the number of concrete program executions MOLLY performed (**exe**) and the time elapsed in seconds (**wall**). We also measure the performance of random fault injection (**Random**), showing the average for each measurement over 25 runs.

Program	*LOC*	*EOT*	*EFF*	*Combinations*	*exe*
redun-deliv	7	11	10	8.07×10^{18}	11
ack-deliv	5	8	7	3.08×10^{13}	673
paxos-synod	33	7	6	4.81×10^{11}	173
bully-le	11	10	9	1.26×10^{17}	2
flux	41	22	21	6.20×10^{76}	187

Table 5.2: MOLLY guarantees the absence of counterexamples for correct programs for a given configuration. For each bug-free program, we ran MOLLY in parameter sweep mode for 120 seconds without discovering a counterexample. We show the highest parameter settings (**Fspec**) explored within that bound, the number of possible combinations of failures (**Combinations**), and the number of concrete executions MOLLY used to cover the space of failure combinations (**Exe**).

5.7 Related Work

In this section, we compare LDFI to existing techniques for testing and verifying fault-tolerant distributed systems.

Model checking [96, 64, 85, 135, 171, 136, 105] is a widely used technique for systematically checking distributed systems for violations of correctness properties. Model checkers can provide guarantees that *no* bad executions are possible by exhaustively checking all program states reachable from a set of initial states. Model checking is ideally suited to specifying and exhaustively testing individual *components* (particularly protocols) of distributed systems. For practical distributed systems—systems that run for long periods of time, and are built from a variety of components—this state space is often too large to exhaustively explore. Some attempts to manage this complexity include abstraction refinement [48, 31], running model checkers concurrently with execution to detect possible invariant violations in the local neighborhood [169], and heuristic search ordering of the state space [171]. LDFI sidesteps the state explosion problem by asking a

	LDFI	Model checking						Test generation			Fault injection	
	Molly	MoDist [171]	TLA+ [105]	Chess [136]	CrystalBall [169]	MaceMC [96]	SPIN [85]	quick-check [47]	Execution synthesis [175]	Symbolic execution [151, 38]	FATE & DESTINI [79]	Random fault injection [6]
Tests failures	X	X	X		X	X	X				X	X
Executable code	X	X		X	X	X		X	X	X	X	X
Safety violations	X	X	X	X	X	X	X	X			X	X
Liveness violations	X		X		X	X						
Explains bugs	X								X			
Generates inputs								X		X		
Tests interleavings		X	X	X	X	X	X		X			
Unmodified systems		X		X				X	X	X		X

Figure 5.12: Overview of approaches to verifying the fault-tolerance of distributed systems.

simpler, more targeted question: *given* a class of good outcomes, can some combination of *faults* prevent them? The complexity of this problem depends on the depth of the lineage of the good outcomes rather than on the size of the global state space.

Fault injection frameworks [6, 55, 79, 93] interpose on the execution of distributed programs to explore the consequences of multiple failures on specific outcomes. Fault injection techniques typically use either a random [129, 6] or heuristic [79, 55] strategy to explore the space of possible failures. FATE and DESTINI [79] has been used to reproduce dozens of known bugs in cloud software, as well as to discover new ones. Like Molly, it uses Datalog as a specification language; unlike Molly, it uses a combination of brute force and heuristic search to explore failure combinations. Molly takes a more complete approach, providing assurances that no bugs exist for particular configurations and execution bounds.

Figure 5.12 places LDFI in the feature space of existing tools that identify bugs in large-scale distributed systems. It is worthwhile to note that two of the key features not provided by LDFI—input generation and testing of interleavings—are provided by existing tools that are compatible with LDFI. A remaining area for investigation is achieving the benefits of LDFI for source code implemented in a widely-used language: Section 3.3 discusses alternative approaches.

LDFI focuses specifically on the effects of faults on outcomes, and is compatible with a variety of other techniques that address orthogonal issues. At a high level, Molly's alternating execution strategy resembles concolic execution [151], which similarly alternates between calls to a concrete evaluator and a symbolic solver. As we discuss in Section 5.8, concolic testing and other symbolic execution approaches (e.g. Klee [38]) are ideal for discovering bad *inputs*, and hence are complementary to LDFI. When verifying individual components, LDFI can be used as a complementary approach to model checkers such as Chess [136], which focus strictly on non-determinism in interleavings. Like test generation approaches such as execution synthesis [175], Molly "explains" bugs, albeit at a higher level of abstraction: via data and dependencies rather than stepwise program execution. Unlike execution synthesis, Molly does not require *a priori* knowledge of bugs, but discovers them.

Like reverse data management [131], LDFI uses provenance to reason about what changes to input relations would be necessary to produce a particular change in output relations ("how-to" queries). When the submitted Dedalus program is logically monotonic [17, 127], LDFI answers how-to queries using positive *why* provenance [44]; otherwise LDFI must also consider the *why-not* provenance [86] of negated rule premises (e.g., "an acknowledgment was not received"). Systems such as Artemis [84] address the why-not problem (for monotonic queries) as a special case of *what-if* analysis, and ask what new (perhaps partially-specified) tuples would need to be *added* to a database to make a missing tuple appear. First-order games [100, 143] use a game-theoretic execution strategy to answer why-not queries. Wu et al. [168] describe a practical approach to answering why-not queries for software-defined networks (SDNs). LDFI can be viewed as a narrow version of the "how-to" provenance problem, restricted to considering deletions on a single distinguished input relation (the clock), in the presence of possibly non-monotonic queries. Many of the why-not provenance techniques discussed above could assist in implementing LDFI—this is a promising avenue for future work.

5.8 Future Work

To conclude our discussion, we reflect on some of the limitations of the Molly prototype, as well as directions for future work. Our narrow focus on the fault-tolerance of distributed systems allowed us to significantly simplify the verification task, but these simplifying abstractions come with tradeoffs.

It is clearly impractical to exhaustively explore all possible inputs to a distributed system, as they are unbounded in general. We have assumed for the purposes of this discussion that the program inputs—including the execution topology—are given *a priori*, either by a human or by a testing framework. However, our approach is compatible with a wide variety of techniques for exploring system inputs, including software unit testing, symbolic execution [151, 38] and input generation [47, 19].

LDFI assumes that the distributed protocols under test are "internally deterministic" (i.e., deterministic modulo the nondeterminism introduced by the environment). It leverages this assumption—common in many fault-tolerant system designs [148]—to provide its completeness guarantee: if some execution produces a proof tree of an outcome, any subsequent execution with the same faults will also. While Molly can be used to find bugs in fundamentally non-deterministic protocols like anti-entropy [160] or randomized consensus [33], certifying such protocols as bug-free will require additional research.

The pseudo-synchronous abstraction presented in Section 5.2—which made it possible to discover complex bugs by rapidly exploring multiple heterogeneous failures—does come at a cost. For an important class of fault-tolerant distributed algorithms (e.g., those that attempt to solve consensus), an abstraction that factors apart partial failure and asynchrony is fundamentally incomplete, because these algorithms are required to (attempt to) distinguish between delay and failure [40, 63]. For example, when we verify an algorithm like Paxos [106] (described in Section 3.3), the conclusion that Paxos is tolerant to a particular set of failures does not imply that Paxos terminates in all executions. Relaxing the synchronicity abstraction is an avenue of future work, but Section 3.3 provides evidence that the tradeoff is worthwhile.

5.9 Discussion

Fault tolerance code is hard to test in a controlled environment, yet likely to fail catastrophically at scale unless it is fully debugged. Ad hoc approaches like random fault injection are easy to integrate with real-world code but unable to provide bullet-proof assurances. LDFI presents a middle ground between pragmatism and formalism, dictated by the importance of verifying fault tolerance in spite of the complexity of the space of faults. LDFI works with executable code, though it requires that code to be written in a language that meets the requirements outlined in Section 5.2.

By walking this middle ground, LDFI and MOLLY offer significant benefits over the state of the art in three dimensions. First, LDFI provides radical improvements in the efficiency of fault injection by narrowing down the choice of relevant faults to inject. Second, LDFI enables MOLLY to provide useful software engineering tools, illustrating tricky fault-tolerance bugs with concrete traces complete with auto-generated visualizations of root causes (lineage diagrams) and communication visualizations (Lamport diagrams). Finally, LDFI makes it possible to formally "bless" code as being correct up to a significant depth of execution, something that is infeasible with traditional fault injection techniques.

Chapter 6

Concluding Remarks

Distributed systems are increasingly ubiquitous, but remain stubbornly hard to program and reason about. This thesis reported significant progress on both fronts. We demonstrated that *query languages* are a natural fit for expressing both distributed *protocols* and large-scale *infrastructure*. By obscuring details such as differences in state representation and control-flow patterns, such languages allow programmers to focus on the principal citizen of distributed computation: *data*, changing as it moves through space and time. We developed the disorderly programming languages Dedalus and Bloom, which preserve the systems-as-queries model while adding the ability to unambiguously characterize change and uncertainty. By capturing such details in their model-theoretic semantics, Dedalus and Bloom make it possible to reason directly about the relationship between distributed programs and their *outcomes*, despite the widespread nondeterminism in their environments. Blazes and LDFI then cash in on this promise, realizing powerful analyses in practical tools that help programmers reason about whether their programs are consistent and fault-tolerant, and assisting in their repair if they are not.

Blazes showed that the disorderly programming philosophy and the analysis techniques it enables can be applied outside the walled garden of Dedalus and Bloom, and can add value to existing distributed systems (such as Apache Storm) with the aid of programmer insight. A recent collaboration with Netflix has provided strong evidence that with the appropriate tracing and fault injection infrastructures, LDFI can also be applied to existing large-scale systems. More valuable future work lies in exploring the middle ground between the "boil the ocean" approach of rewriting systems infrastructure in experimental languages and improving our understanding of existing, unmodified systems. We are actively exploring "specification mining" techniques to extract Blazes-style annotations or synthesize Dedalus subprograms by observing service behavior.

Molly automated the role of the adversary in the game presented in Section 5.3. But what about the role of the programmer? In future work, it would be interesting to explore using the backwards reasoning approach of LDFI to assist in fault-tolerant program synthesis. Given a distributed program with a fault-tolerance bug, it seems possible to use the lineage of its failure-free run (along with a counterexample) to effectively guide the search through program transformations that provide additional redundant support of the program outcome. Similar techniques should also facilitate adapting existing fault-tolerant algorithms (like the **classic-deliv** protocol) to new failure assumptions.

More work needs to be done to unify the analysis techniques developed in this thesis. Blazes and LDFI are complementary techniques that provide programmers with guarantees that their programs are robust to asynchrony and partial failure (respectively). Unfortunately, focusing on one of these sources of uncertainty at a time does not yield a complete solution. Many of the most challenging problems in distributed systems exist at the intersection of asynchrony and partial failure, precisely because it is impossible to distinguish between delay and failure in a truly asynchronous distributed system. It would be interesting to explore hybrid approaches that use data lineage to communicate details about consistency anomalies back to programmers, and that augment programs with additional fault-tolerance mechanisms much as Blazes augmented programs with synchronization.

Another avenue for future work is to apply the disorderly programming philosophy to the debugging of large-scale data management systems. Existing debugging mechanisms (including statement-level debuggers and tracing and replay infrastructures) reflect the sequential model of computation underlying most current programming languages, and focus the programmer's attention on stepwise computation rather than on the modification and movement of data. LDFI—which used data lineage extracted from execution traces not just to identify bugs, but to explain those bugs to the programmer at an appropriately high level of abstraction—hints at the promise of "declarative debugging" techniques for large-scale systems.

This thesis presented a variety of complementary solutions—spanning language design, theory, program analysis, and tooling—to the growing crisis of pervasive distribution. Instead of attempting to *mask* the various complexities that arise from asynchrony and partial failure (as did many historical approaches such as strongly consistent storage infrastructures and distributed transactions) the solutions we propose provide programmers with tools to *manage* complexity, ignoring it when possible and engaging with it when required for program correctness. Realizing this agenda required rethinking the languages we use to program distributed systems as well as the semantics that we use to model them. The frameworks presented in the latter half of the thesis provide an example of what can be built upon this foundation, and (we believe) paint a hopeful picture of the future of computing in a disorderly and distributed world.

Bibliography

[1] Apache Samza. http://samza.apache.org/.

[2] Appalachian trail conservancy guide: trail markings. http://www.appalachiantrail.org/hiking/hiking-basics/trail-markings. Accessed: 2013-11-25.

[3] Consensus-based replication in Hadoop. http://www.wandisco.com/system/files/documentation/Meetup-ConsensusReplication.pdf.

[4] HDFS scalability with multiple namenodes. https://issues.apache.org/jira/browse/HDFS-1052.

[5] High availability for the Hadoop distributed file system. http://blog.cloudera.com/blog/2012/03/high-availability-for-the-hadoop-distributed-file-system-hdfs/.

[6] The Netflix Simian Army. http://techblog.netflix.com/2011/07/netflix-simian-army.html, 2011.

[7] Kafka 0.8.0 Documentation. https://kafka.apache.org/08/documentation.html, 2013.

[8] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. B. Zdonik. Aurora: a New Model and Architecture for Data Stream Management. *VLDB Journal*, 12(2):120–139, Aug. 2003.

[9] H. Abelson and G. J. Sussman, editors. *Structure and Interpretation of Computer Programs*. McGraw Hill, second edition, 1996.

[10] A. Abouzeid et al. HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads. In *VLDB*, 2009.

[11] P. A. Alsberg and J. D. Day. A Principle for Resilient Sharing of Distributed Resources. ICSE '76.

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

[13] 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.

[14] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, and R. C. Sears. BOOM Analytics: Exploring Data-centric, Declarative Programming for the Cloud. In *EuroSys*, 2010.

[15] 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*, 2010.

[16] P. Alvaro, N. Conway, J. M. Hellerstein, and D. Maier. Blazes: coordination analysis for distributed programs. http://arxiv.org/abs/1309.3324, 2013. Technical Report.

[17] P. Alvaro, N. Conway, J. M. Hellerstein, and W. R. Marczak. Consistency Analysis in Bloom: a CALM and Collected Approach. CIDR'12.

[18] P. Alvaro, N. Conway, J. M. Hellerstein, and W. R. Marczak. Consistency Analysis in Bloom: a CALM and Collected Approach. In *CIDR*, 2011.

[19] P. Alvaro, A. Hutchinson, N. Conway, W. R. Marczak, and J. M. Hellerstein. BloomUnit: Declarative Testing for Distributed Programs. DBTest '12.

[20] P. Alvaro, W. R. Marczak, N. Conway, J. M. Hellerstein, D. Maier, and R. Sears. Dedalus: Datalog in Time and Space. In *Proc. Datalog 2.0 Workshop*, 2011.

[21] P. Alvaro, J. Rosen, and J. M. Hellerstein. Lineage-driven fault injection. In *SIGMOD*, 2015.

[22] T. J. Ameloot, F. Neven, and J. Van den Bussche. Relational Transducers for Declarative Networking. PODS'12.

[23] T. J. Ameloot, F. Neven, and J. Van den Bussche. Relational Transducers for Declarative Networking. In *PODS*, 2011.

[24] A. Arasu, S. Babu, and J. Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *VLDB Journal*, 15(2), June 2006.

[25] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. 2007.

[26] M. P. Ashley-Rollman, P. Lee, S. C. Goldstein, P. Pillai, and J. D. Campbell. A Language for Large Ensembles of Independently Executing Nodes. In *Proceedings of the International Conference on Logic Programming*, 2009.

[27] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Highly Available Transactions: virtues and limitations. VLDB'14.

[28] P. Bailis and K. Kingsbury. The Network is Reliable. *Commun. ACM*, 2014.

[29] J. Baker, C. Bond, J. C. Corbett, 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. CIDR'11.

[30] M. Balazinska, J.-H. Hwang, and M. A. Shah. Fault-Tolerance and High Availability in Data Stream Management Systems. In L. Liu and M. T. Özsu, editors, *Encyclopedia of Database Systems*, pages 1109–1115. Springer US.

[31] T. Ball, V. Levin, and S. K. Rajamani. A Decade of Software Model Checking with SLAM. *Commun. ACM*, 2011.

[32] N. Belaramani, J. Zheng, A. Nayate, R. Soulé, M. Dahlin, and R. Grimm. Pads: A policy architecture for distributed storage systems. In *NSDI*, 2009.

[33] M. Ben-Or. Another Advantage of Free Choice (Extended Abstract): Completely Asynchronous Agreement Protocols. PODC '83.

[34] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[35] K. Birman, G. Chockler, and R. van Renesse. Toward a Cloud Computing Research Agenda. *SIGACT News*, 40(2):68–80, June 2009.

[36] H. Blodget. Amazon's Cloud Crash Disaster Permanently Destroyed Many Customers' Data. http://www.businessinsider.com/amazon-lost-data-2011-4, April 2011.

[37] P. Buneman, S. Khanna, and W.-c. Tan. Why and Where: A Characterization of Data Provenance. ICDT'01.

[38] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. OSDI'08.

[39] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: An engineering perspective. In *PODC*, pages 398–407, 2007.

[40] T. D. Chandra, V. Hadzilacos, and S. Toueg. The Weakest Failure Detector for Solving Consensus. *J. ACM*, July 1996.

[41] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[42] K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. Comput. Syst.*, 3(1), 1985.

[43] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a Distributed Storage System for Structured Data. OSDI'06.

[44] J. Cheney, L. Chiticariu, and W.-C. Tan. Provenance in Databases: Why, How, and Where. *Found. Trends databases*, April 2009.

[45] D. Chimenti, R. Gamboa, R. Krishnamurthy, S. Naqvi, S. Tsur, and C. Zaniolo. The LDL System Prototype. *IEEE Trans. on Knowl. and Data Eng.*, 2(1):76–90, 1990.

[46] D. C. 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 *5th ACM Conference on Embedded networked Sensor Systems (SenSys)*, 2007.

[47] K. Claessen and J. Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. ICFP '00.

[48] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided Abstraction Refinement for Symbolic Model Checking. *J. ACM*, 2003.

[49] J. G. Cleary, M. Utting, and R. Clayton. Data Structures Considered Harmful. In *Australasian Workshop on Computational Logic*, 2000.

[50] T. Condie, D. Chu, J. M. Hellerstein, and P. Maniatis. Evita raced: Metacompilation for declarative networks. 2008.

[51] N. Conway, P. Alvaro, E. Andrews, and J. M. Hellerstein. Edelweiss: Automatic Storage Reclamation for Distributed Programming. In *VLDB*, 2014.

[52] N. Conway, W. R. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier. Logic and Lattices for Distributed Programming. In *SoCC*, 2012.

[53] 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. OSDI'12.

[54] Y. Cui, J. Widom, and J. L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.*, June 2000.

[55] S. Dawson, F. Jahanian, and T. Mitton. ORCHESTRA: A Fault Injection Environment for Distributed Systems. Technical report, FTCS, 1996.

[56] J. Dean. Designs, Lessons and Advice from Building Large Distributed Systems. http://www.cs.cornell.edu/projects/ladis2009/talks/deankeynoteladis2009.pdf, 2009. Ladis'09 Keynote.

[57] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*.

[58] M. A. Derr, S. Morishita, and G. Phipps. The Glue-Nail Deductive Database System: Design, Implementation, and Evaluation. *The VLDB Journal*, 3:123–160, 1994.

[59] J. Edwards. Coherent Reaction. In *OOPSLA*, 2009.

[60] J. Eisner, E. Goldlust, and N. A. Smith. Dyna: a declarative language for implementing dynamic programs. In *Proc. ACL*, 2004.

[61] J. Field, M.-C. Marinescu, and C. Stefansen. Reactors: A Data-Oriented Synchronous/Asynchronous Programming Model for Distributed Applications. *Theoretical Computer Science*, 410(2-3), February 2009.

[62] M. J. Fischer. The consensus problem in unreliable distributed systems (A brief survey). In *Proceedings of the 1983 International FCT-Conference on Fundamentals of Computation Theory*, pages 127–140, 1983.

[63] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*, April 1985.

[64] D. Fisman, O. Kupferman, and Y. Lustig. On verifying fault tolerance of distributed protocols. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *LNCS*. Springer Berlin Heidelberg, 2008.

[65] H. Garcia-Molina. Elections in a distributed computing system. *IEEE Trans. Comput.*, January 1982.

[66] H. Garcia-Molina and K. Salem. Sagas. In *SIGMOD*, 1987.

[67] D. Gelernter. Generative communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7:80–112, January 1985.

[68] W. M. Georg Lausen, Bertram Lud*ä*scher. On Active Deductive Databases: The Statelog Approach. In *Transactions and Change in Logic Databases*, pages 69–106, 1998.

[69] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP*, 2003.

[70] P. Gill, N. Jain, and N. Nagappan. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. SIGCOMM '11.

[71] C. Gray and D. Cheriton. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *SOSP*, 1989.

[72] J. Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, 1978.

[73] J. Gray. The transaction concept: virtues and limitations (invited paper). In *Proceedings of the Seventh International Conference on Very Large Data Bases*, pages 144–154, 1981.

[74] J. Gray. Why do computers stop and what can be done about it?, 1985.

[75] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. In *SIGMOD*, 1996.

[76] S. Greco, L. Palopoli, and P. Rullo. Netlog: A Logic Query Language for Network Model Databases. *Data and Knowledge Engineering*, 1991.

[77] S. Greco and C. Zaniolo. Greedy Algorithms in Datalog with Choice and Negation. In *JICSLP'98: Proceedings of the 1998 Joint International Conference and Symposium on Logic Programming*, pages 294–309, Cambridge, MA, USA, 1998. MIT Press.

[78] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. PODS '07.

[79] H. S. Gunawi, T. Do, P. Joshi, P. Alvaro, J. M. Hellerstein, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, K. Sen, and D. Borthakur. FATE and DESTINI: A Framework for Cloud Recovery Testing. NSDI'11.

[80] H. S. Gunawi et al. SQCK: A Declarative File System Checker. In *OSDI*, 2008.

[81] D. Hastorun et al. Dynamo: Amazon's Highly Available Key-Value Store. In *SOSP*, 2007.

[82] P. Helland and D. Campbell. Building on Quicksand. In *CIDR*, 2009.

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

[84] M. Herschel, M. A. Hernández, and W.-C. Tan. Artemis: A System for Analyzing Missing Answers.

[85] G. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.

[86] J. Huang, T. Chen, A. Doan, and J. F. Naughton. On the Provenance of Non-answers to Queries over Extracted Data. VLDB'08.

[87] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX ATC*, 2010.

[88] M. Interlandi, L. Tanca, and S. Bergamaschi. Datalog in time and space, synchronously. CEUR'13.

[89] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *EuroSys*, 2007.

[90] M. B. Jones. Interposition agents: transparently interposing user code at the system interface. In *SOSP*, 1993.

[91] F. P. Junqueira, B. C. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. DSN '11.

[92] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, Aug 1974. North Holland, Amsterdam.

[93] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. Ferrari: A flexible software-based fault and error injection system. *IEEE Trans. Comput.*, Feb 1995.

[94] G. Karvounarakis, Z. G. Ives, and V. Tannen. Querying data provenance. SIGMOD '10.

[95] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. marc Loingtier, and J. Irwin. Aspect-oriented Programming. ECOOP'97.

[96] C. E. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. NSDI'07.

[97] K. Kingsbury. Call me maybe: Kafka. http://aphyr.com/posts/293-call-me-maybe-kafka, 2013.

[98] J. Kirsch and Y. Amir. Paxos for system builders. Technical Report CNDS-2008-2, Johns Hopkins University, 2008.

[99] S. Köhler, B. Ludäscher, and Y. Smaragdakis. Declarative datalog debugging for mere mortals. In *Datalog in Academia and Industry*, LNCS. Springer Berlin Heidelberg, 2012.

[100] S. Köhler, B. Ludäscher, and D. Zinn. First-Order Provenance Games. In *In Search of Elegance in the Theory and Practice of Computation*, volume 8000 of *LNCS*. Springer, 2013.

[101] L. Kuper and R. R. Newton. LVars: Lattice-based Data Structures for Deterministic Parallelism. FHPC'13.

[102] L. Kuper and R. R. Newton. A Lattice-Theoretical Approach to Deterministic Parallelism with Shared State. Technical Report TR702, Indiana University, Oct. 2012.

[103] M. S. Lam et al. Context-sensitive program analysis as database queries. In *PODS*, 2005.

[104] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[105] L. Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 1994.

[106] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[107] L. Lamport. Paxos made simple. *SIGACT News*, 32(4):51–58, December 2001.

[108] B. W. Lampson. Getting computers to understand. *J. ACM*, 50(1), 2003.

[109] Hadoop jira issue tracker, July 2009. `http://issues.apache.org/jira/browse/HADOOP`.

[110] J. Leibiusky, G. Eisbruch, and D. Simonassi. *Getting Started with Storm - Continuous Streaming Computation with Twitter's Cluster Technology*. O'Reilly, 2012.

[111] 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.

[112] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, and D. Maier. Out-of-order Processing: a New Architecture for High-performance Stream Systems. *PVLDB*, 1(1):274–288, 2008.

[113] B. Liskov. Distributed programming in Argus. *CACM*, 31:300–312, 1988.

[114] M. Liu and J. Cleary. Declarative Updates in Deductive Databases. *Journal of Computing and Information*, 1:1435–1446, 1994.

[115] 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. SOSP '11.

[116] B. T. Loo. Querying network graphs with recursive queries. Technical Report UCB-CS-04-1332, UC Berkeley, 2004.

[117] B. T. Loo. *The Design and Implementation of Declarative Networks*. PhD thesis, 2006.

[118] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking: language, execution and optimization. In *SIGMOD*, 2006.

[119] 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, 2009.

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

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

[122] L. Lu and J. G. Cleary. An Operational Semantics of Starlog. In *Principles and Practice of Declarative Programming*, 1999.

[123] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1997.

[124] D. Maier, A. O. Mendelzon, and Y. Sagiv. Testing Implications of Data Dependencies. *ACM Transactions on Database Systems*, 4:455–469, 1979.

[125] O. Malik. When the Cloud Fails: T-Mobile, Microsoft Lose Sidekick Customer Data. http://gigaom.com/2009/10/10/when-cloud-fails-t-mobile-microsoft-lose-sidekick-customer-data/, Oct 2009.

[126] Y. Mao. On the declarativity of declarative networking. In *NetDB*, 2009.

[127] 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.

[128] W. R. Marczak et al. Declarative reconfigurable trust management. In *CIDR*, 2009.

[129] P. D. Marinescu and G. Candea. LFI: A practical and general library-level fault injector. In *DSN*. IEEE, 2009.

[130] D. Mazières. Paxos made practical. http://www.scs.stanford.edu/~dm/home/papers/paxos.pdf, January 2007.

[131] A. Meliou and D. Suciu. Tiresias: The Database Oracle for How-to Queries. SIGMOD '12.

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

[133] I. S. Mumick and O. Shmueli. How Expressive is Stratified Aggregation? *Annals of Mathematics and Artificial Intelligence*, 15(3-4):407–435, Sept. 1995.

[134] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer. Provenance-aware storage systems. ATEC '06.

[135] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. *SIGOPS Oper. Syst. Rev.*, 2002.

[136] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. OSDI'08.

[137] J. A. Navarro and A. Rybalchenko. Operational Semantics for Declarative Networking. In *PADL*, pages 76–90, 2009.

[138] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed Stream Computing Platform. In *ICDMW*, 2010.

[139] Nokia Corporation. disco: massive data – minimal code, 2009. http://discoproject.org/.

[140] R. Perera, U. A. Acar, J. Cheney, and P. B. Levy. Functional Programs That Explain Their Work. ICFP '12.

[141] T. C. Przymusinski. *On the Declarative Semantics of Deductive Databases and Logic Programs*, pages 193–216. Morgan Kaufmann, Los Altos, CA, 1988.

[142] N. Raychaudhuri. *Scala in Action*. Manning Publications Co., 2013.

[143] S. Riddle, S. Köhler, and B. Ludäscher. Towards Constraint Provenance Games. TaPP'14.

[144] M. C. Rinard and P. C. Diniz. Commutativity Analysis: a New Analysis Technique for Parallelizing Compilers. *ACM Trans. Program. Lang. Syst.*, 19(6):942–991, Nov. 1997.

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

[146] K. A. Ross. A syntactic stratification condition using constraints. In *International Symposium on Logic Programming*, pages 76–90, 1994.

[147] D. Saccà and C. Zaniolo. Stable Models and Non-Determinism in Logic Programs with Negation. In *ACM Symposium on Principles of Database Systems (PODS)*, pages 205–217, 1990.

[148] F. B. Schneider. Implementing Fault-tolerant Services Using the State Machine Approach: a Tutorial. *ACM Comput. Surv.*, 22(4), Dec. 1990.

[149] T. Schutt et al. Scalaris: Reliable transactional P2P key/value store. In *SIGPLAN Workshop on Erlang*, 2008.

[150] R. Sears and E. Brewer. Stasis: Flexible transactional storage. In *OSDI*, pages 29–44, 2006.

[151] K. Sen and G. Agha. Automated Systematic Testing of Open Distributed Programs. In L. Baresi and R. Heckel, editors, *Fundamental Approaches to Software Engineering*, volume 3922 of *LNCS*. 2006.

[152] M. A. Shah, J. M. Hellerstein, and E. Brewer. Highly Available, Fault-tolerant, Parallel Dataflows. In *SIGMOD*, 2004.

[153] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Research report, INRIA, 2011.

[154] A. Singh et al. Using queries for distributed monitoring and forensics. In *EuroSys*, 2006.

[155] D. Skeen. Nonblocking Commit Protocols. SIGMOD '81.

[156] M. Stonebraker. Concurrency control and consistency of multiple copies of data in distributed ingres. *IEEE Trans. Softw. Eng.*, May 1979.

[157] M. Stonebraker. Inclusion of New Types in Relational Data Base Systems. In *ICDE*, 1986.

[158] T. Stoppard. Arcadia: a play in two acts. Samuel French, Inc., 1993.

[159] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch. Session Guarantees for Weakly Consistent Replicated Data. PDIS '94.

[160] 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.

[161] The Hive Project. Hive home page, 2009. http://hadoop.apache.org/hive/.

[162] R. H. Thomas. A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. *ACM Trans. Database Syst.*, June 1979.

[163] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. SIGMOD '12.

[164] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting Punctuation Semantics in Continuous Data Streams. *TKDE*, 15(3):555–568, 2003.

[165] J. D. Ullman. *Principles of Database and Knowledge-Base Systems: Volume II: The New Technologies*. W. H. Freeman & Co., 1990.

[166] W. Vogels. Eventually Consistent. *CACM*, 52(1):40–44, 2009.

[167] W. White et al. Scaling games to epic proportions. In *SIGMOD*, 2007.

[168] Y. Wu, A. Haeberlen, W. Zhou, and B. T. Loo. Answering Why-not Queries in Software-defined Networks with Negative Provenance. HotNets'13.

[169] M. Yabandeh, N. Knezevic, D. Kostic, and V. Kuncak. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. NSDI'09.

[170] F. Yang et al. Hilda: A high-level language for data-driven web applications. In *ICDE*, 2006.

[171] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. MODIST: Transparent Model Checking of Unmodified Distributed Systems. NSDI'09.

[172] Y. Yu et al. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, 2008.

[173] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. HotCloud'10.

[174] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized Streams: an Efficient and Fault-tolerant Model for Stream Processing on Large Clusters. In *HotCloud*, 2012.

[175] C. Zamfir and G. Candea. Execution Synthesis: A Technique for Automated Software Debugging. EuroSys '10.

[176] C. Zaniolo. The database language GEM. In *SIGMOD*, 1983.

[177] W. Zhou, M. Sherr, T. Tao, X. Li, B. T. Loo, and Y. Mao. Efficient querying and maintenance of network provenance at internet-scale. SIGMOD '10.