Logic and Lattices for Distributed Programming

Neil Conway
William Marczak
Peter Alvaro
Joseph M. Hellerstein
David Maier

Electrical Engineering and Computer Sciences
University of California at Berkeley

June 22, 2012

Acknowledgement

Logic and Lattices for Distributed Programming

Neil Conway
UC Berkeley
nrc@cs.berkeley.edu

William R. Marczak
UC Berkeley
wrm@cs.berkeley.edu

Peter Alvaro
UC Berkeley
palvaro@cs.berkeley.edu

Joseph M. Hellerstein
UC Berkeley
hellerstein@cs.berkeley.edu

David Maier
Portland State University
maier@cs.pdx.edu

ABSTRACT

In recent years there has been interest in achieving application-level consistency criteria without the latency and availability costs of strongly consistent storage infrastructure. A standard technique is to adopt a vocabulary of commutative operations; this avoids the risk of inconsistency due to message reordering. A more powerful approach was recently captured by the *CALM theorem*, which proves that logically monotonic programs are guaranteed to be eventually consistent. In logic languages such as Bloom, CALM analysis can automatically verify that program modules achieve consistency without coordination.

In this paper we present Bloom^L, an extension to Bloom that takes inspiration from both these traditions. Bloom^L generalizes Bloom to support lattices and extends the power of CALM analysis to whole programs containing arbitrary lattices. We show how the Bloom interpreter can be generalized to support efficient evaluation of lattice-based code using well-known strategies from logic programming. Finally, we use Bloom^L to develop several practical distributed programs, including a key-value store similar to Amazon Dynamo, and show how Bloom^L encourages the safe composition of small, easy-to-analyze lattices into larger programs.

1. INTRODUCTION

As cloud computing becomes increasingly common, the inherent difficulties of distributed systems—asynchrony, concurrency, and partial failure—affect a growing segment of the developer community. Traditionally, transactions and other forms of strong consistency encapsulated these problems at the data management layer. But in recent years there has been interest in achieving application-level consistency criteria without incurring the latency and availability costs of strongly consistent storage [8, 17]. Two different frameworks for these techniques have received significant attention in recent research: *Convergent Modules* and *Monotonic Logic*.

Convergent Modules: In this approach, a programmer writes encapsulated modules whose public methods provide certain guarantees regarding message reordering and retry. For example, Statebox is an open-source library that merges conflicting updates to data items in a key-value store; the user of the library need only register commutative, associative, idempotent merge functions [19]. This approach has roots in research in databases and systems [12, 14, 17, 27, 39] as well as groupware [11, 37]. Shapiro, et al. recently proposed a formalism for these approaches called *Conflict-Free Replicated Data Types* (CRDTs), which casts these ideas into the algebraic framework of *semilattices* [34, 35].

CRDTs present two main problems: (a) the programmer bears responsibility for ensuring lattice properties for their methods (commutativity, associativity, idempotence), and (b) CRDTs only provide guarantees for individual data objects, not for application logic in general. As an example of this second point, consider the following:

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

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

Monotonic Logic: In recent work, we observed that the database theory literature on monotonic logic provides a powerful lens for reasoning about distributed consistency. Intuitively, a monotonic program makes forward progress over time: it never "retracts" an earlier conclusion in the face of new information. We proposed the CALM theorem, which established that all monotonic programs are *confluent* (invariant to message reordering and retry) and hence eventually consistent [5, 18, 25]. Monotonicity of a Datalog program is

straightforward to determine conservatively from syntax, so the CALM theorem provides the basis for a simple analysis of the consistency of distributed programs. We concretized CALM into an analysis procedure for *Bloom*, a Datalog-based language for distributed programming [2, 9].

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

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

The CALM theorem obviates any scoping concerns for convergent monotonic logic, but it presents a **type dilemma**. Sets are the only data type amenable to CALM analysis, but the programmer may have a more natural representation of a monotonically growing phenomenon. For example, a monotonic counter is more naturally represented as a growing integer than a growing set. This dilemma leads either to false negatives in CALM analysis and over-use of coordination, or to idiosyncratic set-based implementations that can be hard to read and maintain.

1.1 Bloom^L: Logic and Lattices

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

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

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

2. Bloom^L solves the scope dilemma of CRDTs by providing monotonicity-preserving mappings between lattices via *morphisms* and *monotone functions*, as described

```
1   class ShortestPaths
2     include Bud
3
4     state do
5       table :link, [:from, :to] => [:cost]
6       scratch :path, [:from, :to, :next_hop, :cost]
7       scratch :min_cost, [:from, :to] => [:cost]
8     end
9
10    bloom do
11      path <= link {|l| [l.from, l.to, l.to, l.cost]}
12      path <= (link*path).pairs(:to => :from) do |l,p|
13        [l.from, p.to, l.to, l.cost + p.cost]
14      end
15
16      min_cost <= path.group([:from, :to], min(:cost))
17    end
18  end
```

Figure 1: All-pairs shortest paths in Bloom.

below. By using such mappings, the per-component monotonicity guarantees offered by CRDTs can be extended across multiple items of lattice type. This capability is key to the CALM analysis described above. It is also useful for establishing the monotonicity of subprograms even when the whole program is not designed to be monotonic.

3. For efficient incremental execution, we extend the standard Datalog semi-naive evaluation scheme [7] to support arbitrary lattices. We also describe how an existing Datalog-style engine can be extended to support lattices with relatively minor changes.

1.2 Outline

The remainder of the paper proceeds as follows. Section 2 provides background on Bloom and CALM. In Section 3 we introduce Bloom^L, including cross-lattice morphisms and monotone functions. We detail Bloom^L's built-in lattice types and show how developers can define new lattices. We also describe how the CALM analysis extends to Bloom^L. In Section 4, we describe how we modified the Bloom runtime to support Bloom^L, including our extension to semi-naive evaluation that supports both lattices and relations.

In Sections 5 and 6, we present two case studies. First, we use Bloom^L to implement a distributed key-value store that supports eventual consistency, object versioning using vector clocks, and quorum replication. Second, we revisit the simple e-commerce scenario presented in Alvaro et al. in which clients interact with a replicated shopping cart service [2]. We show how Bloom^L can be used to make the "checkout" operation monotonic and confluent, despite the fact that it requires aggregating over a distributed data set.

2. BACKGROUND

In this section, we review the Bloom programming language and the CALM program analysis. We highlight a simple distributed program for which the CALM analysis over sets yields unsatisfactory results.

Name	Behavior
`table`	Persistent storage.
`scratch`	Transient storage.
`channel`	Asynchronous communication. A fact derived into a `channel` appears in the database of a remote Bloom instance at a non-deterministic future time.
`periodic`	Interface to the system clock.
`interface`	Interface point between software modules.

Table 1: Bloom collection types.

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

Table 2: Bloom operators.

2.1 Bloom

Bloom programs are bundles of declarative *statements* about collections of *facts* (tuples). An *instance* of a Bloom program performs computation by evaluating its statements over the contents of its local database. Instances communicate via asynchronous messaging, as described below.

An instance of a Bloom program proceeds through a series of *timesteps*, each containing three phases.[1] In the first phase, inbound events (e.g., network messages) are received and represented as facts in collections. In the second phase, the program's statements are evaluated over local state to compute all the additional facts that can be derived from the current collection contents. In some cases (described below), a derived fact is intended to achieve a "side effect," such as modifying local state or sending a network message. These effects are deferred during the second phase of the timestep; the third phase is devoted to carrying them out.

The initial implementation of Bloom, called *Bud*, allows Bloom logic to be embedded inside a Ruby program. Figure 1 shows a Bloom program represented as an annotated Ruby class. A small amount of imperative Ruby code is needed to instantiate the Bloom program and begin executing it; more details are available on the Bloom language website [9].

2.1.1 Data model

The Bloom data model is based on *collections*. A collection is an unordered set of *facts*, akin to a relation in Datalog. The Bud prototype adopts the Ruby type system rather than inventing its own; hence, a fact in Bud is just an array of immutable Ruby objects. Each collection has a *schema*, which declares the structure (column names) of the facts in the collection. A subset of the columns in a collection form its *key*: as in the relational model, the key columns functionally determine the remaining columns. The collections used by a Bloom program are declared in a `state` block. For example, line 5 of Figure 1 declares a collection named `link` with three columns, two of which form the collection's key. Ruby is a dynamically typed language, so keys and values in Bud can hold arbitrary Ruby objects.

Bloom provides five collection types to represent differ-ent kinds of state (Table 1). A `table` stores persistent data: if a fact appears in a table, it remains in the table in future timesteps (unless it is explicitly removed). A `scratch` contains transient data—the content of scratch collections is emptied at the start of each timestep. Scratches are akin to SQL views: they are often useful as a way to name intermediate results or as a "macro" construct to enable code reuse. The `channel` collection type enables communication between Bloom instances. The schema of a channel has a distinguished *location specifier* column (prefixed with "@"); when a fact is derived for a channel collection, it appears in the database of the Bloom instance at the address given by the location specifier. The `periodic` and `interface` collection types do not arise in our discussion in this paper; the interested reader is referred to the Bloom website [9].

2.1.2 Statements

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

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

The left-hand side (lhs) is the name of the output collection and the right-hand side (rhs) is an expression that produces a collection. A statement defines how the input collections are transformed before being included (via set union) in the output collection. Bloom allows the usual relational operators to be used on the rhs (selection, projection, join, grouping, aggregation, and negation), although it adopts a syntax intended to be more familiar to imperative programmers. In Figure 1, line 11 demonstrates projection, lines 12–14 perform a join between `link` and `path` using the join predicate `link.to = path.from` followed by a projection to four attributes, and line 16 shows grouping and aggregation. Bloom statements appear in one or more `bloom` blocks.

Bloom provides several operators that determine *when* the rhs will be merged into the lhs (Table 2). The <= operator performs standard logical deduction: that is, the lhs and rhs are true at the same timestep. The <+ and <- operators indicate that facts will be added or removed, respectively, from the lhs collection at the beginning of the *next* timestep. The <~ operator specifies that the rhs will be merged into the lhs collection at some non-deterministic future time. The lhs of a statement that uses <~ must be a channel; the <~ operator

---

[1]There is a precise declarative semantics for Bloom [1, 3], but we describe the language operationally for the sake of exposition.

captures asynchronous messaging.

Bloom allows recursion—i.e., the rhs of a statement can reference the lhs collection, either directly or indirectly. As in Datalog, certain constraints must be adopted to ensure that programs with recursive statements have a sensible interpretation. For deductive statements (<= operator), we require that programs be *syntactically stratified* [6]: cycles through negation or aggregation are not allowed (unless they contain a deferred or asynchronous operator) [3].

2.2 CALM analysis

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

The CALM theorem connects the theory of monotonic logic with the practical problem of distributed consistency [2, 18]. All monotonic programs are "eventually consistent" or *confluent*: for any given input, all program executions result in the same final state regardless of network non-determinism [5, 25]. Hence, monotonic logic is a useful building block for loosely consistent distributed programming.

According to the CALM theorem, distributed inconsistency may only occur at *points of order*: program locations where the output of an asynchronously derived value is consumed by a non-monotonic operator [2]. This is because asynchronous messaging results in non-deterministic arrival order, and non-monotonic operators may produce different conclusions when evaluated over different subsets of their inputs. For example, consider a Bloom program consisting of a pair of collections *A* and *B* (both fed by asynchronous channels) and a rule that sends a message whenever an element of *A* arrives that is not in *B*. This program is non-monotonic and exhibits non-confluent behavior: the messages sent by the program will depend on the order in which the elements of *A* and *B* arrive.

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

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

4   class QuorumVote
5     include Bud

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

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

Figure 2: A non-monotonic Bloom program that waits for a quorum of votes to be received.

2.2.1 Limitations of set monotonicity

The original formulation of the CALM theorem considered only programs that compute more facts over time—that is, programs whose *sets* grow monotonically. Many distributed protocols make progress over time, but their notion of "progress" is often difficult to represent as a growing set of facts. For example, consider the Bloom program in Figure 2. This program receives votes from a client program (not shown) via the `vote_chn` channel. Once at least `QUORUM_SIZE` votes have been received, a message is sent to a remote node to indicate that quorum has been reached (line 17). This program resembles a "quorum vote" subroutine that might be used by an implementation of Paxos [22] or quorum replication [16].

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

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

3. ADDING LATTICES TO BLOOM

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

When designing Bloom^L, we decided to extend Bloom to include support for lattices rather than building a new

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

4  class QuorumVoteL
5    include Bud

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

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

Figure 3: A monotonic Bloom^L program that waits for a quorum of votes to be received.

language from scratch. Hence, Bloom^L is backward compatible with Bloom and was implemented with relatively minor changes to the Bud runtime. We describe how code written using lattices can interoperate with traditional Bloom collections in Section 3.5.

3.1 Definitions

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

A *monotone function* from poset S to poset T is a function $f : S \rightarrow T$ such that $\forall a, b \in S : a \leq_S b \Rightarrow f(a) \leq_T f(b)$. That is, f maps elements of S to elements of T in a manner that respects the partial orders of both posets.

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

3.2 Language concepts

Bloom^L allows both lattices and collections to represent state. A lattice is analogous to a collection type in Bloom, while a *lattice element* corresponds to a particular collection. For example, the `lset` lattice is similar to the `table` collection type provided by Bloom; an element of the `lset` lattice is a particular set. In the terminology of object-oriented pro-

gramming, a lattice is a class that obeys a certain interface and an element of a lattice is an instance of that class. Figure 3 contains an example Bloom^L program.

As with collections, the lattices used by a Bloom^L program are declared in a `state` block. More precisely, a state block declaration introduces an identifier that is associated with a lattice element; over time, the binding between identifiers and lattice elements is updated to reflect state changes in the program. For example, line 10 of Figure 3 declares an identifier `votes` that is mapped to an element of the `lset` lattice. As more votes are received, the lattice element associated with the `votes` identifier changes (it moves "upward" in the `lset` lattice). When a lattice identifier is declared, it is initially bound to the value $\bot$, the smallest element in the lattice. For example, an `lset` lattice initially contains the empty set.

3.2.1 Statements in Bloom^L

Statements take the same form in both Bloom and Bloom^L:
   <identifier> <op> <expression>
The identifier on the lhs can refer to either a set-oriented collection or a lattice element. The expression on the rhs can contain both traditional relational operators (applied to Bloom collections) and methods invoked on lattices. Lattice methods are similar to methods in an object-oriented language and are invoked using the standard Ruby method invocation syntax. For example, line 17 of Figure 3 invokes the `size` method on an element of the `lset` lattice.

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

Bloom^L does not support deletion (`<-` operator) for lattices. Lattices do not directly support asynchronous communication (via the `<~` operator) but lattice elements can be embedded into tuples that appear in channels (Section 3.5.2).

3.2.2 Lattice methods

Bloom^L statements compute values over lattices by invoking methods on lattice elements. Just as a subset of the relational algebra is monotonic, some lattice methods are monotone functions (as defined in Section 3.1). A monotone lattice method guarantees that, if the lattice on which the method is invoked grows (according to the lattice's partial order), the value returned by the method will grow (according to the return value's lattice type). For example, the `size` method provided by the `lset` lattice is monotone because as more elements are added to the set, the size of the set increases. From a CRDT perspective, a lattice's monotone methods constitute a "safe" interface of operations that can be

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

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

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

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

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

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

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

Table 3: Built-in lattices in Bloom^L. Note that `v` denotes a Ruby value, `n` denotes a number, and `blk` indicates a Ruby code block (anonymous function).

invoked in a distributed setting without risk of inconsistency.

A lattice method's signature indicates its monotonicity properties. Bloom^L distinguishes between methods that are monotone and a subset of monotone methods that are *morphisms*. Section 3.1 defines the properties that a morphism must satisfy, but the intuition is that a morphism on lattice T can be distributed over T's least upper bound. For example, the `size` method provided by the `lset` lattice is not a morphism. To see why, consider two elements of the `lset` lattice, $\{1,2\}$ and $\{2,3\}$. `size` is not a morphism because $size(\{1,2\} \sqcup_{lset} \{2,3\}) \neq size(\{1,2\}) \sqcup_{lmax} size(\{2,3\})$. Morphisms can be evaluated more efficiently than monotone methods, as we discuss in Section 4.1.

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

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

3.3 Built-in lattices

Table 3 lists the lattices included with Bloom^L. The built-in lattices provide support for several common notions of "progress": a predicate that moves from false to true (`lbool`), a numeric value that strictly increases or strictly decreases (`lmax` and `lmin`, respectively), and various kinds of collections that grow over time (`lset`, `lpset`, `lbag`, and `lmap`). The behavior of most of the lattice methods should be unsurprising, so we do not describe every method in this section.

The `lbool` lattice represents conditions that, once satisfied, remain satisfied. For example, the `gt` morphism on the `lmax` lattice takes a numeric argument n and returns an `lbool`; once the `lmax` exceeds n, it will remain $> n$. The `when_true` morphism takes a Ruby block; if the `lbool` element has the value `true`, `when_true` returns the result of evaluating the block. For example, see line 19 in Figure 3. `when_true` is similar to an "if" statement.[2]

The collection-like lattices support familiar operations such as union, intersection and testing for the presence of an element in the collection. The `project` morphism takes a code block and forms a new collection by applying the code block to each element of the input collection. Elements for

---

[2]Observe that an "else" clause would test for an upper bound on the final lattice value, which is a non-monotonic property!

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

4      def initialize(x=[])
5        # Input validation removed for brevity
6        @v = x.uniq # Remove duplicates from input
7      end

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

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

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

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

Figure 4: Example implementation of the `lset` lattice.

which the code block returns `nil` are omitted from the output collection, which allows `project` to be used as a filter.

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

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

3.4 User-defined lattices

The built-in lattices are sufficient to express many programs. However, Bloom^L also allows developers to create custom lattices to capture domain-specific behavior. To define a new lattice, a developer creates a Ruby class that meets a certain API contract. Figure 4 shows a simple implementation of the `lset` lattice using a Ruby array for storage.[3]

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

- `initialize(i)`: given a Ruby object i, this method constructs a new lattice element that "wraps" i (this is the standard Ruby syntax for defining a constructor). If i is `nil` (the null reference), this method returns $\bot$, the

---

[3]We omit a few `lset` methods for brevity. While we use an array for simplicity, this is inefficient (e.g., duplicate elimination requires linear time). The built-in `lset` uses a hash-based set data type.

least element of the lattice.

- `merge(e)`: given a lattice element e, this method returns the least upper bound of *self* and e. This method must satisfy the algebraic properties of least upper bound as summarized in Section 3.1—in particular, it must be commutative, associative, and idempotent. Note that e and *self* must be instances of the same class.

Lattices can also define any number of monotone functions, morphisms, and non-monotonic methods. The syntax for declaring morphisms and monotone functions can be seen in lines 13–15 and 21–23 of Figure 4, respectively. Note that lattice elements are *immutable*—that is, lattice methods (including `merge` methods) must return new values, rather than destructively modifying any of their inputs.

Custom lattices must define a keyword that can be used in Bloom^L state blocks. This is done using the `wrapper_name` class method. For example, line 2 of Figure 4 means that "`lset :foo`" in a state block will introduce an identifier `foo` that is associated with an instance of `Bud::SetLattice`.

3.5 Integration with set-oriented logic

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

3.5.1 Converting collections into lattices

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

For example, line 16 of Figure 3 contains a Bloom collection on the rhs and an `lset` lattice on the lhs. This statement is implemented by constructing a singleton `lset` for each fact in the rhs collection and then merging the sets together. The resulting `lset` is then merged with the `votes` lattice referenced by the lhs.

3.5.2 Collections with embedded lattice values

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

Consider a simple Bloom^L statement that derives tuples with an embedded lattice element as a column:

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

where `t1` and `t2` are Bloom collections, `cnt` is a lattice, and

the key of `t1` is its first column. Note that `cnt` might change over the course of a single timestep (specifically, `cnt` can move "upward" according to the lattice's partial order). This might result in multiple tuples `t1` tuples that differ only in the second column, which would violate `t1`'s key.

To resolve this situation, Bloom^L allows multiple facts to be derived that differ only in their embedded lattice values; those facts are merged into a single fact using the lattice's merge function. This is similar to specifying a procedure for how to resolve key constraint violations, a feature supported by some databases [28, 36]. For similar reasons, lattice elements cannot be used as keys in Bloom collections.

3.6 Confluence in Bloom^L

We now describe how the notion of *confluence* (invariance to message reordering) can be generalized from Bloom to Bloom^L programs. In recent work, we provided a model-theoretic characterization of confluence for programs written in Dedalus, the formal language on which Bloom is based [25]. These results apply directly to Bloom, whose semantics are grounded in those of Dedalus. The result of a distributed computation performed in Bloom may be viewed as the set of *ultimate models* or eventual states of the program, given a fixed input and sufficient time for messages to be delivered. If a program has exactly one ultimate model for every input, we say that it is confluent: all message delivery orders result in the same eventual state.

To reason about confluence in Bloom^L, we first observe that lattices are guaranteed to have inflationary behavior over time: a lattice value only increases. Hence if we include lattices in the output of an otherwise confluent Bloom^L program, they will not increase the number of ultimate models. It is not enough, however, that this local property of lattice objects holds: we must also show that anything a Bloom^L program *does* with a lattice value is monotonic. Monotone lattice functions provide a mechanism for reasoning about composition of lattices or between lattices and collections. By including monotone lattice functions among the "safe," monotonic constructs provided by the Bloom language, we can easily extend our CALM analysis to Bloom^L.

4. IMPLEMENTATION

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

4.1 Semi-naive evaluation

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

Semi-naive evaluation improves upon naive evaluation by making fewer redundant derivations [7]. Let Δ_0 represent the initial database state. In the first round, all the rules are evaluated over Δ_0; let Δ_1 represent the new facts derived in this round. In the second round, we only need to compute derivations that are dependent on Δ_1 because everything that can be derived purely from Δ_0 has already been computed.

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

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

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

4.2 Performance validation

To validate the effectiveness of semi-naive evaluation for Bloom^L programs, we wrote two versions of a program to compute the transitive closure of a directed acyclic graph. One version was written in Bloom and used Bloom collections. The other version was written in Bloom^L using morphisms over the `lset` lattice. For the Bloom^L version, we ran the program both with and without semi-naive evaluation enabled. As input, we generated synthetic graphs of various sizes—in a graph with n nodes, each node had roughly $\log_2 n$ outgoing edges. We ran the experiment using a 2.13 Ghz Intel Core 2 Duo processor and 4GB of RAM, running Mac OS X 10.7.4
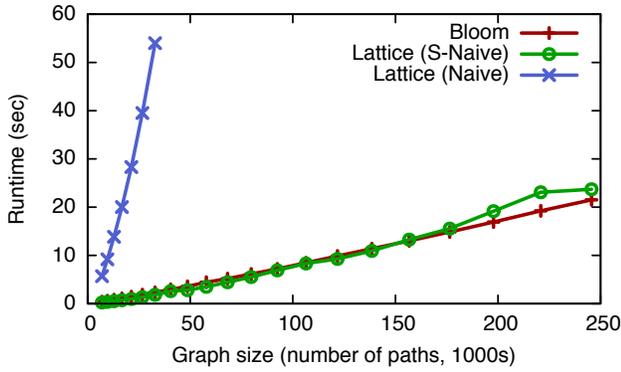
Figure 5: Performance of three different methods for computing the transitive closure of a graph.

and Ruby 1.9.3-p194. We ran each program variant five times on each graph and report the mean elapsed wall-clock time.

Figure 5 shows how the runtime of each program varied with the size of the graph. Note that we only report results for the naive Bloom^L strategy on small input sizes because this variant ran very slowly as the graph size increased. The poor performance of naive evaluation is not surprising: after deriving all paths of length n, naive evaluation will then rederive all those paths at every subsequent round of the fixpoint computation. In contrast, after computing length n paths, a semi-naive strategy will only generate length $n + 1$ paths in the next round. Bloom and semi-naive Bloom^L achieve similar results. We instrumented Bud to count the number of derivations made by the Bloom and semi-naive lattice variants—as expected, both programs made a similar number of derivations. These results suggest that our implementation of semi-naive evaluation for Bloom^L is effective and performs comparably with a traditional implementation of semi-naive evaluation for sets.

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

4.3 Modifying Bud

We were able to extend Bud to support Bloom^L with relatively minor changes. Bud initially had about 7200 lines of Ruby source code (LOC). The core lattice features (the `Bud::Lattice` base class and the mapping from identifiers to lattice elements) required about 300 LOC. Modifying Bud's fixpoint logic to include lattices required only 10 LOC, while the program rewriting required to enable semi-naive evaluation required 100 LOC. Modifying Bud's collection classes to support merging of embedded lattice values required adding or modifying about 125 LOC. The built-in

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

Figure 6: Key-value store interface.

```
1  class KvsReplica
2    include Bud
3    include KvsProtocol
4
5    state { lmap :kv_store }
6
7    bloom do
8      kv_store    <= kvput {|c| {c.key => c.val}}
9      kvput_resp  <~ kvput {|c| [c.reqid, c.client_addr, ip_port]}
10     kvget_resp  <~ kvget {|c| [c.reqid, c.client_addr,
11                                kv_store.at(c.key), ip_port]}
12   end
13 end
```

Figure 7: KVS replica implementation in Bloom^L.

lattice classes constituted an additional 300 LOC. In total, adding support for Bloom^L required less than 900 lines of added or modified code, and took about two person-months of engineering time.

5. CASE STUDY: KEY-VALUE STORE

The next two sections contain case studies that show how Bloom^L can be used to build correct distributed programs. Both case studies are monotonic: that is, both programs consist of monotone functions applied to lattices. As a result, the Bloom^L compiler can verify that both of them are eventually consistent without any need for coordination.[4]

In the first case study, we show that a distributed, eventually consistent key-value store can be *composed* via a series of monotonic mappings between simple lattices. This example highlights the way that Bloom^L overcomes the "scope dilemma" of CRDTs: by composing a complex program from simple lattices (mostly Bloom^L built-ins), we can feel confident that individual lattices are correct, while CALM analysis finishes the job of verifying whole-program correctness.

5.1 Basic Architecture

A key-value store (KVS) provides a lookup service that allows client applications to retrieve the *value* associated with a given *key*. In a typical KVS, key-value pairs are replicated on multiple server replicas for redundancy and the key space is partitioned in some fashion to improve aggregate storage and throughput. *Eventual consistency* is a common correctness criterion: after all client updates have reached all storage

---

[4]Full code listings for these case studies are available at `http://db.cs.berkeley.edu/bloom-lattice`.
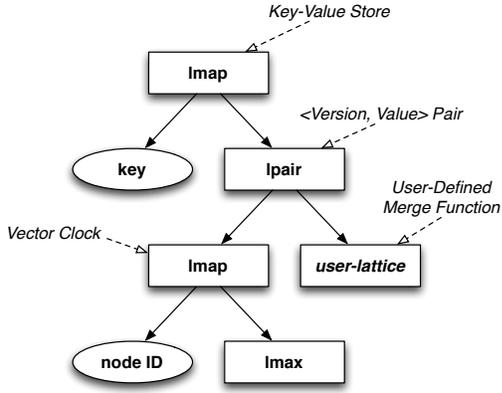
Figure 8: Lattice structure of a KVS with object versioning. Rectangles are lattices and ovals are atomic values.

nodes, all the replicas of a key-value pair will converge to the same final state [39, 41].

Figure 6 shows a simple KVS interface in Bloom^L. Client applications submit *get(key)* and *put(key, val)* operations by inserting into the `kvget` and `kvput` channels, respectively; server replicas return responses via the `kvget_resp` and `kvput_resp` channels.

Figure 7 contains the Bloom^L code for a KVS server replica. An `lmap` lattice is used to maintain the mapping between keys and values (line 5). Since the values in an `lmap` lattice must themselves be lattice elements, for now we assume that clients only want to store and retrieve lattice values; we discuss how to support arbitrary values in Section 5.2. To handle a *put(key, val)* request, a new *key →* *val* `lmap` is created and merged into `kv_store` (line 8). If `kv_store` already contains a value for the given key, the two values will be merged together using the value lattice's merge function (see Section 3.3 for details). Note that we use the Bloom^L features described in Section 3.5 to allow traditional Bloom collections (e.g., channels) and lattices (e.g., the `kv_store` lattice) to be used by the same program. Note also that `ip_port` is a built-in function that returns the IP address and port number of the current Bud instance.

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

5.2 Object Versioning

The basic KVS design is sufficient for applications that want to store monotonically increasing values, such as session logs or increasing counters. To allow storage of values that change in arbitrary ways, we now consider how to support *object versions*. This is a classic technique for recognizing and resolving mutual inconsistency between members of a distributed system [30]; our design is similar to that used by Amazon Dynamo [10].

Each replica associates keys with ⟨*vector-clock, value*⟩ pairs. The vector clock (VC) captures the causal relationship between different versions of a record [13, 26, 10]. Clients get and put ⟨*vector-clock, value*⟩ pairs. When a client updates a value it has previously read, the client increments its own position in the VC and includes the updated vector clock V_U with its *put* operation. Upon receiving an update, the server compares V_U with the VC of the server's version of the record (V_S). If $V_U > V_S$, the server replaces the stored record with the client's update. If $V_S > V_U$, the update is ignored (this situation might arise due to duplication and reordering of messages by the network). If V_U and V_S are incomparable, the two versions are concurrent, so a client-supplied reconciliation function is used to resolve the conflict.

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

5.2.1 Vector Clocks

Vector clocks are a well-known mechanism for recording the causal relationships between events [13, 21]. A vector clock is a map from node identifiers to logical clocks. Let V_e denote the vector clock for event e; let $V_e(n)$ denote the logical clock associated with node n in V_e. If $V_e < V_{e'}$, e causally precedes e', where

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

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

5.2.2 Version-Value Pairs

We now turn to representing ⟨*vector-clock, value*⟩ pairs. To do this, we define a new lattice `lpair` that "wraps" two lattice elements; we use *fst* and *snd* to refer to the first and second elements of an `lpair`, respectively. The discussion above suggests a natural least upper bound for `lpair`:

$$A \sqcup B = \begin{cases} A & \text{if } A.fst > B.fst \\ B & \text{if } A.fst < B.fst \\ \langle A.fst \sqcup B.fst, A.snd \sqcup B.snd \rangle & \text{otherwise} \end{cases}$$

This implements the desired semantics: given two candidate values for a key, the candidate with the strictly greater version number should be preferred. When the two versions are incomparable, a new `lpair` should be formed by merging both elements of the input pairs with one another. In the case of the KVS, *fst* is a vector clock, while *snd* is the user's

10

data; the least upper bound of the *snd* lattice corresponds to a user-defined merge function.[5]

Note that while the *fst* of a given `lpair` increases over time (as new versions are received), the *snd* may not (a newer version might contain a "smaller" *snd*). Again, this is the desired behavior.

5.2.3 Discussion

Figure 8 shows the lattices used in the KVS with object versioning. Surprisingly, adding support for object versioning did not require *any* changes to the KVS replica code! Instead, clients simply store `lpair` values containing a vector clock as the first element and increment their position in the vector clock when submitting updates. The KVS replica merges these `lpair` values into an `lmap` as usual; the merge function of `lpair` handles conflict resolution in the appropriate manner. Moreover, by composing the KVS from a collection of simple lattices, we found it easy to reason about the behavior of the system. For example, convincing ourselves that the KVS replicas will eventually converge only required checking that the individual `lmap`, `lmax`, and `lpair` lattices satisfy the lattice properties, rather than analyzing the behavior of the system as a whole.

Our design compares favorably to traditional implementations of object versioning and vector clocks. For example, the implementation of vector clocks alone in Voldemort (a popular key-value store) requires 216 lines of Java, not including whitespace or comments [23]. In Bloom^L, vector clocks follow directly from the composition of the `lmap` and `lmax` lattices; the *entire KVS* requires less than 100 lines of Ruby and Bloom^L code, including the client library. The `lpair` lattice requires an additional lines 30 of Ruby but is completely generic, and could be included as a built-in lattice.

5.3 Quorum Reads and Writes

To further demonstrate the flexibility of our implementation, we add an additional feature to our KVS: the ability to submit reads and writes to a configurable number of nodes. If a client reads from R nodes and writes to W nodes in a KVS with N replicas, the user can set $R + W > N$ to achieve behavior equivalent to a quorum replication system [16], or use smaller values of R and W if eventual consistency is sufficient. This scheme allows users to vary R and W on a per-operation basis, depending on their consistency and durability requirements.

To support this feature, we can use the Bloom^L quorum voting pattern first introduced in Figure 3. After sending a write to W systems, the KVS client accumulates `kvput_resp` messages into an `lset`. Testing for quorum can be done in a monotonic fashion by mapping the `lset` to an `lmax` (using the `size` method), and then performing a threshold test using

---

[5]If the user stores a value that does not have a natural merge function, similar systems typically provide a default merge function that collects conflicting updates into a set for eventual manual resolution by the user. Such a strategy could easily be implemented monotonically with Bloom^L.
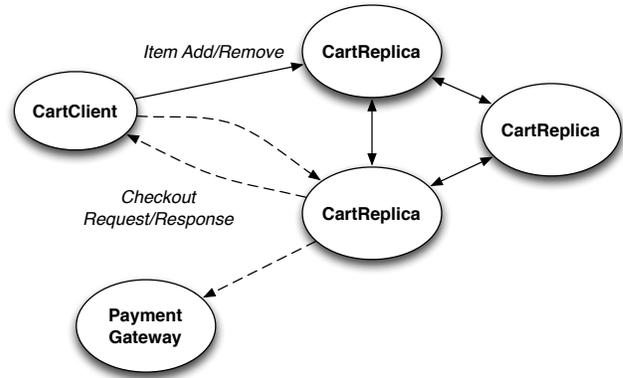

Figure 9: Shopping cart system architecture.

`gt_eq` on `lmax`. As expected, this is monotonic: once quorum has been reached, it will never be retracted.

Quorum reads work in a similar fashion, except that the client must also merge together the R versions of the record it receives. This follows naturally from the discussion in Section 5.2: the client simply takes the least upper bound of the values it receives, which produces the expected behavior. The client can optionally write the merged value back to the KVS (so-called "read repair" [10]); note that the `lpair` merge method also updates the record's vector clock appropriately.

6. CASE STUDY: SHOPPING CARTS

In the previous section, we showed how a complete, consistent distributed program can be composed via monotonic mappings between simple lattice types. In this section, we focus on the way that Bloom^L enables us to overcome the "type dilemma" of Bloom, by demonstrating the use of custom lattice types. Our previous implementation of this case study in Bloom called for coordination due to apparently-nonmonotonic grouping and aggregation [2]; by using custom lattice types in the implementation here, we enable the Bloom^L CALM analysis to verify eventual consistency without any coordination.

In this case study, we consider a simple e-commerce system in which clients interact with a shopping cart service by adding and removing items over the course of a shopping session (Figure 9). The cart service is replicated to improve fault tolerance; client requests can be routed to any of the cart replicas. Eventually, a client submits a "checkout" operation, at which point the cumulative effect of their shopping session should be summarized and returned to the client. In a practical system, the result of the checkout operation might be presented to the client for confirmation or submitted to a payment processor to complete the e-commerce transaction. This case study is based on the cart system from Alvaro et al. [2], which was in turn inspired by the discussion of replicated shopping carts in the Dynamo paper [10].

Alvaro et al. discuss two different designs: a "disorderly" version in which the cart state is represented as a set of op-

```
1   module CartProtocol
2     state do
3       channel :action_msg,
4         [:@server, :session, :reqid] => [:item, :cnt]
5       channel :checkout_msg,
6         [:@server, :session, :reqid] => [:lbound, :addr]
7       channel :response_msg,
8         [:@client, :session] => [:summary]
9     end
10  end
11
12  module MonotoneReplica
13    include CartProtocol
14
15    state { lmap :sessions }
16
17    bloom do
18      sessions <= action_msg do |m|
19        c = LCart.new({m.reqid => [ACTION, m.item, m.cnt]})
20        { m.session => c }
21      }
22      sessions <= checkout_msg do |m|
23        c = LCart.new({m.reqid => [CHECKOUT, m.lbound, m.addr]})
24        { m.session => c }
25      }
26
27      response_msg <~ sessions do |session, c|
28        c.is_complete.when_true {
29          [c.checkout_addr, session, c.summary]
30        }
31      end
32    end
33  end
```

Figure 10: Cart server replica in Bloom^L that supports a monotonic (coordination-free) checkout operation.

erations (allowing monotonic accumulation of adds and removes) and a "destructive" version in which the cart state is managed by a key-value store, which requires a non-monotonic update on each cart action.

6.1 Monotonic checkout

For both the "disorderly" and "destructive" designs, Alvaro et al. argue that processing a checkout request is non-monotonic because it requires aggregating over an asynchronously computed data set—in general, coordination might be required to ensure that all inputs have been received before the checkout response can be returned to the client. However, observe that the client knows exactly which add and remove operations should be reflected in the result of the checkout. If that information can be propagated to the cart service, any server replica can decide if it has enough information to process the checkout operation without needing additional coordination. This design is monotonic: once a checkout response is produced, it will never change or be retracted. Our goal is to translate this design into a monotonic Bloom^L program.

Figure 10 contains the server code for this design (we omit the client code for the sake of brevity). Communication with the client occurs via the channels declared in the `CartProtocol` module. We represent the state of a server replica using an `lmap` lattice that associates session IDs with `lcart` lattice elements. `lcart` is a custom lattice that represents the state of a single shopping cart. An `lcart` contains a set of client operations. Each operation has a unique ID;

operation IDs are assigned by the client in increasing numeric order without gaps. An `lcart` contains two kinds of operations, *actions* and *checkouts*. An action describes the addition or removal of an item from the cart. An `lcart` contains at most one checkout operation—the checkout specifies the smallest operation ID that must be reflected in the result of the checkout, along with the address where the checkout response should be sent. Lines 19 and 23 construct `lcart` elements that contain a single action or checkout operation, respectively. These singleton carts are then merged with the previous cart state associated with the client's session, if any.

An `lcart` is *complete* if it contains a checkout operation as well as all the actions in the ID range identified by the checkout. Hence, testing whether an `lcart` is complete is a monotone function: it is similar to testing whether an accumulating set has crossed a threshold. Hence, if any server replica determines that it has a complete cart, it can send a response to the client without risking inconsistency. Without coordination, the client might receive multiple responses but they will all reflect the same cart contents.

Note that the statement that produces a response to the client (lines 27–31) is contingent on having a complete cart. `summary` is a monotone method that returns a summary of the actions in the cart—an exception is raised if `summary` is called before the cart is complete. Similarly, attempts to construct "illegal" `lcart` elements (e.g., carts containing multiple checkout operations or actions that are outside the ID range specified by the checkout) also result in runtime exceptions, since this likely indicates a logic error in the program. Implementing the `lcart` lattice required 58 lines of Ruby using the lattice API described in Section 3.4.

7. RELATED WORK

This paper relates to work on concurrency control, distributed storage, and non-monotonic logic programming.

Semantics-based concurrency control: The traditional correctness criteria for concurrency control schemes is serializability [29]. However, ensuring serializability can be prohibitively expensive, for instance when transactions are long-running or the nodes of a distributed database are connected via an unreliable network. Thus, many methods have been proposed to allow non-serializable transaction schedules that preserve some *semantic* notion of correctness. In particular, several schemes allow users to specify that certain operations can be commuted with other operations; this enlarges the space of legal schedules, increasing the potential for concurrency [12, 14, 42].

O'Neil describes a method for supporting "escrow" transactions, which allow operations that are only commutative when a certain limited resource is available [27]. For example, credits and debits to a bank account might only commute if the bank account balance can be guaranteed to be non-negative. We are currently exploring how to add support for escrow operations to Bloom^L.

To support concurrent editing of shared documents, the

groupware community has studied a family of algorithms known as *Operational Transformations* (OT) [11, 37]. Many OT algorithms have been proposed but the correctness criteria is typically familiar: after quiesence, all replicas of the document should converge to the same final state, the causal relationship between operations should be preserved, and the final state of the document should reflect the semantic intent of each user's editing operations [38].

Commutativity in distributed systems: Many distributed systems allow users to specify that certain operations are commutative, associative, or idempotent. Helland and Campbell observe that using commutative, associative and idempotent operations is particularly valuable as systems scale and guaranteeing global serializability becomes burdensome [17]. Many distributed storage systems allow users to provide "merge functions" that are used to resolve write-write conflicts between replicas, allowing the system to eventually reach a consistent state (e.g., [10, 19, 24, 31, 39]).

Shapiro et al. recently proposed *Conflict-free Replicated Data Types* (CRDTs), a principled approach to the design of loosely consistent data values [35]. Shapiro et al. provide a formal model for convergence based on join semilattices and a catalog of practical CRDT designs [34]. CRDTs and Bloom^L lattice types often follow similar design patterns to achieve coordination-free convergence. Unlike Bloom^L, the CRDT approach considers the correctness of replicated values in isolation. This allows CRDTs to be more easily adapted into standalone libraries (e.g., Statebox [19]). However, the narrow focus of CRDTs means that, even if a CRDT is correct, application state may remain inconsistent (Section 1).

Non-monotonicity in deductive databases: Adding non-monotonic operators to Datalog increases the expressiveness of the language but introduces significant complexities: unrestricted use of non-monotonicity would allow programs that imply logical contradictions (e.g., "p if $\neg p$"). A simple solution is to disallow recursion through aggregation or negation, which admits only the class of "stratified programs" [6]. Many attempts have been made to assign a semantics to larger classes of programs (e.g., [15, 32, 40]).

The observation that many uses of aggregation and negation have a "monotonic" flavor has been made before. Ross and Sagiv study a class of programs that include "monotonic" aggregates [33]. They propose a model-theoretic semantics for this class of programs that is similar to our semantics for Bloom^L in Section 3. Our work differs from Ross and Sagiv's in several respects: most notably, they use lattices as a way to characterize classes of Datalog programs, whereas we propose Bloom^L as a practical programming language. Accordingly, Ross and Sagiv restrict the usage of monotone aggregates to a single "cost" argument in certain predicates, do not allow user-defined lattices, and do not propose a framework for arbitrary lattices to be composed safely.

Köstler et al. consider Datalog extended with subsumption relations, which allows the user to indicate that certain deductions should be "preferred" over others [20]. These preferences must form a lattice; Köstler et al. then propose a model-theoretic semantics and evaluation scheme based on the lattice's partial order. Like Ross and Sagiv, this work shares some technical similarities with this paper, but differs in its goals and problem domain: Köstler et al. use subsumption to add semantic knowledge to graph traversal and heuristic search programs, but do not propose a general-purpose programming framework.

Zaniolo and Wang identify a class of "monotone aggregates" as part of their work on supporting advanced user-defined aggregates in the $\mathcal{LDL}$++ system [43]. Like us, they observe that monotone aggregates can easily be supported without stratification in a Datalog system based on semi-naive fixpoint. Their characterization of monotone aggregates is different than ours, and they do not consider asynchrony or distribution. In fact, supporting order-dependent aggregates is an explicit goal of their work, whereas we seek to ensure that programs are confluent in the face of message reordering.

8. DISCUSSION AND FUTURE WORK

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

Nevertheless, designing a correct lattice can still be difficult. To address this, we plan to develop tools to give programmers more confidence in the correctness of lattice implementations. For example, we plan to build a test data generation framework that can efficiently cover the space of possible inputs to lattice merge functions, drawing upon recent work on test generation for Bloom [4]. We also plan to explore a restricted DSL for implementing lattices, which would make formal verification of correctness an easier task.

Every join semilattice includes ⊥, a distinguished "smallest element." A natural extension is to consider providing *bounded lattices* that also contain ⊤, a "greatest element." Such a value is already supported by `lbool` (⊤ = `true`), in addition to the `lcart` lattice discussed in Section 6.1. ⊤ behaves differently than other lattice elements: because it is immutable, any function can safely be applied to it (whether monotone or not), without risking inconsistency. Since the merge function for ⊤ will always yield ⊤, this might also allow a more efficient representation. For example, in a complete `lcart`, we need only store the "summarized" cart state, not the log of client operations.

9. CONCLUSION

In this paper, we proposed Bloom^L, a distributed variant of Datalog that extends logic programming with join semilattices. Bloom^L is particularly valuable for enabling

coordination-free, consistent distributed programming, overcoming key hurdles in prior work. Like CRDTs, Bloom^L allows application-specific notions of "progress" to be represented as lattices, and goes further by enabling safe mappings between lattices. Bloom^L improves upon our own earlier work by expanding the space of recognizably monotonic programs, allowing more programs to be verified as eventually consistent via the CALM analysis. In addition to providing richer semantic guarantees than previous approaches, in our experience Bloom^L provides a natural and straightforward language for building distributed systems.

Acknowledgments

10. REFERENCES

[1] P. Alvaro et al. A Declarative Semantics for Dedalus. Technical Report UCB/EECS-2011-120, EECS Department, UC Berkeley, Nov. 2011.

[2] P. Alvaro et al. Consistency Analysis in Bloom: a CALM and Collected Approach. In *CIDR*, 2011.

[3] P. Alvaro et al. Dedalus: Datalog in time and space. In *Datalog Reloaded*. Springer Berlin / Heidelberg, 2011.

[4] P. Alvaro et al. BloomUnit: Declarative Testing for Distributed Programs. In *DBTest*, 2012.

[5] T. J. Ameloot et al. Relational transducers for declarative networking. In *PODS*, 2011.

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

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

[8] K. Birman et al. Toward a cloud computing research agenda. *SIGACT News*, 40(2):68–80, 2009.

[9] Bloom programming language. http://www.bloom-lang.org.

[10] G. DeCandia et al. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.

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

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

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

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

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

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

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

[18] J. M. Hellerstein. The Declarative Imperative: Experiences and Conjectures in Distributed Logic. *SIGMOD Record*, 39(1):5–19, 2010.

[19] B. Ippolito. statebox, an eventually consistent data model for Erlang (and Riak). http://labs.mochimedia.com/archive/2011/05/08/statebox/, 2011.

[20] G. Köstler et al. Fixpoint Iteration with Subsumption in Deductive Databases. *Journal of Intelligent Information Systems*, 4(2):123–148, 1995.

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

[22] L. Lamport. The part-time parliament. *ACM TOCS*, 16(2):133–169, May 1998.

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

[24] W. Lloyd et al. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *SOSP*, 2011.

[25] W. R. Marczak et al. Confluence analysis for distributed programs: A model-theoretic approach. In *Datalog 2.0*, 2012. To appear.

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

[27] P. E. O'Neil. The Escrow transactional method. *ACM TODS*, 11(4):405–430, Dec. 1986.

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

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

[30] D. S. Parker et al. Detection of Mutual Inconsistency in Distributed Systems. *IEEE Transactions on Software Engineering*, SE-9(3):240–247, 1983.

[31] R. Power and J. Li. Piccolo: building fast, distributed programs with partitioned tables. In *OSDI*, 2010.

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

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

[34] M. Shapiro et al. A comprehensive study of convergent and commutative replicated data types. Technical Report RR-7506, INRIA, 2011.

[35] M. Shapiro et al. Conflict-free replicated data types. In *13th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, 2011.

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

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

[38] C. Sun et al. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM TOCHI*, 5(1):63–108, Mar. 1998.

[39] D. B. Terry et al. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *SOSP*, 1995.

[40] A. Van Gelder et al. The well-founded semantics for general logic programs. *JACM*, 38(3):619–649, 1991.

[41] W. Vogels. Eventually Consistent. *CACM*, 52(1):40–44, 2009.

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

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