Blazes: Coordination Analysis and Placement for Distributed Programs

PETER ALVARO, University of California, Santa Cruz

NEIL CONWAY and JOSEPH M. HELLERSTEIN, University of California, Berkeley

DAVID MAIER, Portland State University

Distributed consistency is perhaps the most-discussed topic in distributed systems today. Coordination protocols can ensure consistency, but in practice they cause undesirable performance unless used judiciously. Scalable distributed architectures avoid coordination whenever possible, but under-coordinated systems can exhibit behavioral anomalies under fault, which are often extremely difficult to debug. This raises significant challenges for distributed system architects and developers.

In this article, we present BLAZES, a cross-platform program analysis framework that (a) identifies program locations that require coordination to ensure consistent executions, and (b) automatically synthesizes application-specific coordination code that can significantly outperform general-purpose techniques. We present two case studies, one using annotated programs in the Twitter Storm system and another using the Bloom declarative language.

CCS Concepts: • **Information systems** → **Parallel and distributed DBMSs**;

Additional Key Words and Phrases: Consistency via Coordination

ACM Reference format:

Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and David Maier. 2017. Blazes: Coordination Analysis and Placement for Distributed Programs. *ACM Trans. Database Syst.* 42, 4, Article 23 (October 2017), 31 pages.

<https://doi.org/10.1145/3110214>

1 INTRODUCTION

Over the past decade, the database research community has deliberately widened its focus to “maximize impact ... across computing” by exploring a variety of computer science challenges that seem well-suited to data management technologies [3]. One promising dimension of this expanded agenda is the exploration of declarative languages for new platforms and application domains, including (among others) network protocols, machine learning, and cloud computing.

We would like to thank Michael Armbrust, Peter Bailis, Alan Fekete, Andy Gross, Coda Hale, Tim Kraska, Josh Rosen, Dmitriy Ryaboy and the anonymous reviewers for their helpful feedback on this paper. This work was supported by the Air Force Office of Scientific Research (Grant No. FA95500810352), the Natural Sciences and Engineering Research Council of Canada, the National Science Foundation (Grants No. CNS-0722077, No. IIS-0713661, No. IIS-0803690, and No. IIS-0917349), and gifts from EMC, Microsoft Research, and NTT Multimedia Communications Laboratories.

Authors’ addresses: P. Alvaro, Department of Computer Science, University of California, Santa Cruz, 1156 High Street, MS SOE3, Santa Cruz, CA 95064; email: palvaro@ucsc.edu; N. Conway, 1384 Florida Street, Apt B, San Francisco, 94110; email: nrc@cs.berkeley.edu; J. M. Hellerstein, 387 Soda Hall #1776, Berkeley, CA 94720-1776; email: hellerstein@berkeley.edu; D. Maier, Department of Computer Science, Portland State University, PO Box 751, Portland, OR 97207-0751; email: maier@pdx.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2017 ACM 0362-5915/2017/10-ART23 \$15.00

<https://doi.org/10.1145/3110214>

Distributed systems and cloud computing offer particularly attractive opportunities for declarative languages, given their focus on data-centric applications. Initial work in this domain largely focused on benefits of code simplicity, showing that declarative languages are a natural fit for specifying distributed systems internals and can yield code that is easier to maintain and extend [5, 6, 8, 40]. While that work has the potential for long-term software engineering benefits, it provides little short-term incentive for distributed systems developers to switch from their familiar imperative languages and tools.

In this article, we show how database technology can address a significantly more urgent issue for distributed systems developers: the correctness and efficiency of distributed consistency mechanisms for fault-tolerant services. The need for consistency, and methods for achieving it, have been the subject of extensive debate in the practitioner and research community [15, 18, 30]. Coordination services for distributed consistency, such as Chubby [19] and Zookeeper [32], are in wide use. At the same time, there have been various efforts to address consistency in system-specific ways, including NoSQL systems [20], Internet services infrastructure [12, 19, 24, 25], and even large-scale machine-learning systems [41, 46]. The reason for the interest is clear: for many practitioners, distributed consistency is the most critical issue for system performance and manageability at scale [15].

1.1 Blazes

Recent work has highlighted promising connections between distributed consistency and database theory surrounding monotonicity [8, 9, 31, 54]. In this article, we move beyond theory and language design to develop practical techniques that have direct utility for popular distributed programming platforms like Twitter Storm [37], while providing even more power for the declarative languages like Bloom [17] that are being designed for the future.

Specifically, 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 1. BLAZES can be directly applied to existing programming platforms based on distributed stream or dataflow processing, including Twitter Storm [37], Apache S4 [44], and Spark Streaming [53]. 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.

Fig. 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.

We make the following contributions in this article:

- **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.
- **Grey Box/White Box Approach.** We demonstrate how the semantics of opaque components can be modeled using a system of simple, programmer-supplied *annotations*. When components are implemented in a sufficiently rich language, we show how these annotations can be automatically inferred.
- **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.

1.2 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 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,

Fig. 2. Physical architecture of a Storm word count topology.

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 concerned with ensuring that accurate counts are committed to the store despite the at-least-once delivery semantics. Storm ensures fault-tolerance via replay: if component instances fail or time out, stream sources re-deliver 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 [37].

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 8) can result in a $3\times$ improvement in throughput.

Ad-tracking with Bloom: Figure 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 *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, then 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.

Fig. 3. Physical architecture of an ad-tracking network.

1.3 Outline

The remainder of the article is organized as follows. Section 2 briefly describes the underlying system model of distributed *components* communicating via asynchronous *streams*. Section 3 describes the consistency anomalies that can arise in such a model and strategies that are commonly applied to avoid them. Section 4 presents a simple language of annotations that programmers can use to characterize component-local properties that enable reasoning about the consistency of compositions of components. Section 5 then describes the BLAZES metalanguage and analysis system, which can, given an annotated system, determine what consistency anomalies are possible and identify an appropriate strategy for preventing them. Section 7 presents a case study in which three distributed applications are annotated, analyzed and automatically coordinated when necessary. Section 8 evaluates the savings of selecting a custom coordination strategy. We conclude with related and future work.

2 SYSTEM MODEL

The BLAZES API is based on a simple “black box” model of component-based distributed services. We use dataflow graphs [34] 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 represents the ad-tracking system at two levels of granularity: a dataflow of components and a dataflow of relational operators.

2.1 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 [39] 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) has two input streams, and hence defines two possible dataflow paths (from *c* to *r* and

Fig. 4. Dataflow representations of the ad-tracking network's Report and Cache components.

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.

While streams are unbounded, in practice they are often divided into batches [13, 37, 53] to enable replay-based fault-tolerance. *Runs* are (possibly repeated) executions over finite stream batches.

A stream producer can optionally embed *punctuations* [50] into the stream. A punctuation is used to indicate that the producer will generate no more messages within a particular logical partition of the stream. For example, in Figure 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 5.5, we show how punctuations can enable efficient, localized coordination strategies based on *sealing*.

3 DATAFLOW CONSISTENCY

In this section, we characterize the consistency anomalies that can arise in distributed, fault-tolerant dataflow systems and the various mechanisms that can be used to prevent them. We begin by describing undesirable behaviors that can arise due to the interaction between nondeterministic message orders and fault-tolerance mechanisms. We then review existing strategies for preventing such anomalies, which we generalize into mechanisms that enforce constraints on message delivery (Section 3.2) or exploit semantic properties of components (Section 3.3). We further generalize delivery constraints 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 1.2, and show how slight differences in the queries can lead to different distributed anomalies.

3.1 Anomalies

Asynchronous communication interacts with fault-tolerance mechanisms in subtle ways. Two standard schemes exist for making dataflow systems fault tolerant: *replication* (used in the ad reporting system described in Section 1.2) and *replay* (employed by Storm and Spark) [13]. 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 *consistency*, 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 anomalies that are difficult to debug.

Due to our weak assumptions about transport guarantees, nondeterminism in message delivery order—and as a consequence, in the ordering of system outputs—is expected. The stream anomalies upon which we focus in this article arise precisely when that nondeterminism in *order* leads to nondeterminism in stream *contents*. We focus on three increasingly severe classes of anomalies:

- (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. They are also challenging to test and debug.
- (2) *Cross-instance nondeterminism*, in which replicated instances of the same component produce different output contents in the *same* execution over the same inputs. Cross-instance nondeterminism can lead to inconsistencies across queries. Due to its transient and timing-sensitive nature, cross-instance non-determinism is particularly difficult to debug; replicated streams will differ in contents in some runs, but it may be difficult to reproduce.
- (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.

All three classes of anomaly are *witnesses* of asynchronous execution; nondeterminism in message ordering has “leaked” into program outputs. If we could ensure that every component in a dataflow produces a deterministic *set* of outputs regardless of the *order* in which its inputs are delivered, then we could prevent all of these anomalies. The rest of this section describes existing techniques for ensuring this property on a component-by-component basis.

Much like strong consistency (e.g., linearizability) in distributed storage systems, this guarantee of output determinism is not in itself sufficient to ensure that arbitrary application-specific invariants (e.g., ensuring that a bank account balance is ≥ 0 , or that a set of uniqueness and foreign key constraints are not violated) are upheld in all executions. However, like strong consistency, it eases the programmer’s burden in reasoning about application-specific properties by significantly constraining the space of possible distributed executions.

3.2 Consistency via Coordination

All of the anomalies above are direct consequences of nondeterminism in delivery order interacting with component state. A sufficient strategy for preventing the anomalies (or arbitrary components) is to remove the nondeterminism in delivery order by coordinating the (otherwise asynchronous) processing of inputs.

For example, programmers could adopt the state machine replication approach [47] and use a totally ordered broadcast mechanism [16, 36] to ensure that replicas process messages in the same global order despite nondeterminism in delivery order. This approach has the advantage of generality: it is sufficient to rule out replica disagreement irrespective of component semantics. Unfortunately, this generality comes at a high cost. Global coordination mechanisms such as Paxos and Zookeeper impose significant throughput, latency, and availability penalties in large-scale systems [1, 15, 27].

While enforcing a total order over message delivery is *sufficient* to rule out the most severe dataflow anomalies, in many practical scenarios it is not strictly *necessary*. Custom coordination strategies and restricted programming models often employ coarse-grained *barriers* [28, 51, 53] that rule out anomalies by delaying computation until input batches are completely determined.

Fig. 5. The relationship between potential stream anomalies and remediation strategies based on component properties and delivery mechanisms. For each anomaly (increasing in severity from top to bottom), we show the properties and mechanisms that can be used to *prevent* it. For example, convergent components prevent only replica divergence, a dynamic message ordering mechanism prevents all replication anomalies, and confluent components and sealing can prevent all anomalies involving nondeterministic contents.

Barrier coordination has two clear advantages over global ordering. First, it reduces the *frequency* of coordination, allowing for better overlapping of computation and communication. Second, it often reduces the *scope* of coordination from a global ordering decision to a local signal between producer and consumer. We measure the potential savings for practical streaming applications in Section 8.

3.3 Consistency via Component Properties

The previous section described strategies that prevent nondeterminism of system outputs by removing the nondeterminism in delivery order. Broadly speaking, the other obvious strategy is to ensure that all components produce the same results for all delivery orders. Distributed programs that produce the same outcome for all message delivery orders exhibit none of the anomalies listed in Section 3.1, regardless of the choice of fault-tolerance or delivery mechanisms. Because of the often prohibitive costs of coordination-based strategies, much recent research has explored ways to exploit the semantic properties of individual components to rule out consistency anomalies without requiring coordination [8, 9, 11, 48]. We review some of these component-specific properties.

A dataflow component is *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 shown in Figure 5. The CALM Theorem [8] establishes that programs expressed in *monotonic* logic are confluent [8, 9, 31]. 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.

Recent work in the distributed systems community has focused on replica *convergence* as a key property for avoiding coordination costs without sacrificing consistency [25, 48, 52]. Replicated storage is convergent (or “eventually consistent”) if, when all messages have been delivered, all replicas agree on the set of stored values [52]. Convergent components (e.g., CRDTS [48]) are guaranteed never to exhibit replica divergence and to *eventually* stop exhibiting cross-instance nondeterminism.

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

Fig. 6. Reporting server queries (shown in SQL syntax for familiarity).

Convergence is a local property of component state, while confluence is (as we shall see) a composable property of the *behavior* of components with respect to their outputs. Monotonicity (a property of program logic) is a sufficient condition for confluence. 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 5 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 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, then the result is replica divergence.

Figure 5 captures the relationship between delivery mechanisms, component properties, and output stream anomalies.

3.4 Example Queries

The ad reporting system presented in Section 1.2 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 6 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 [23].

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 non-monotonic: as more clicks are observed, the set of poorly performing ads might shrink as well as grow—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 [2, 10, 21]. 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 non-monotonic query *CAMPAIGN* can produce deterministic outputs.

In summary, *THRESH* is a confluent query, and *WINDOW* and *CAMPAIGN* can be made confluent by exploiting barrier synchronization using sealing. *POOR* is not confluent.¹

3.5 Discussion

In an ideal world, we would build dataflows out of strictly confluent components. Such confluent-by-construction systems could leverage any fault tolerance mechanism without concern for consistency anomalies and without incurring the costs of synchronization mechanisms. Of course, we are willing to pay these costs when they are necessary—that is, for dataflows built from strictly order-sensitive components. Unfortunately, practical systems (such as the ad Reporting system) are often composed from collections of components having different sensitivities to ordering. When we add an order-sensitive component to an otherwise confluent dataflow, how can we avoid throwing out the baby with the bath water by falling back on a global coordination strategy?

As we will see, BLAZES provides a framework that allows programmers to reason *locally* about the properties of individual components and to automatically reason transitively about the *global* properties that follow from them. This global reasoning makes it possible to safely avoid coordination when it is unnecessary and to automatically choose an appropriate coordination mechanism otherwise.

4 ANNOTATED DATAFLOW GRAPHS

So far, we have focused on the consistency anomalies that can affect individual “black box” components and global strategies to prevent them (i.e., totally ordered delivery mechanisms, barriers, and confluent dataflows).

In this section, we extend our discussion to a *grey box* model in which programmers provide simple annotations about the semantic properties of components. In the next section, we will show how BLAZES uses these local annotations to automatically derive the global consistency properties of entire dataflow graphs.

4.1 Annotations and Labels

In this section, we describe a language of *annotations* and *labels* that enriches the “black box” model (Section 2) with additional semantic information. Programmers supply annotations about paths through components and about input streams; using this information, BLAZES automatically derives labels for each component’s output streams.

4.1.1 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 7 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.

¹Strictly speaking, *POOR* is sealable on *id*. However, since every record has a unique *id*, implementing barrier synchronization on *id* would enforce total ordering and provide no benefit over a generic ordered delivery strategy.

<i>Label</i>	<i>Confluent</i>	<i>Stateless</i>
<i>CR</i>	X	X
<i>CW</i>	X	
<i>OR_{gate}</i>		X
<i>OW_{gate}</i>		

Fig. 7. The BLAZES component annotations. A component path is either **Confluent** or **Order-sensitive**, and either changes component state (a **Write** path) or does not (a **Read-only** path).

<i>Stream label</i>	<i>ND Order</i>	<i>ND Contents</i>	<i>Permanent ND Contents</i>
<i>Async</i>	X		
<i>Seal_{key}</i>	X		
<i>NDRead_{gate}</i>	X	X	
<i>Taint_{name}</i>	X	X	X

Fig. 8. Stream labels, capturing the anomalies that a particular dataflow can exhibit. *Async* and *Seal* may be supplied by programmers based on semantic knowledge of input streams; the other labels are derived by the analysis.

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 the path does not modify the component's state. For example, the single path through the *Splitter* component in the *Storm* dataflow can be marked *CR*—it is a stateless component, and is trivially commutative as the contents of its output stream at any time are determined only by the contents of its input stream at that time.

CW denotes a path that is confluent and stateful. That is to say, calls to the API may modify the internal state of the component, but the final state is determined only by the input contents and not their order. Convergent objects such as CRDTs [48] can be labeled *CW*. In the *Storm* example, we can label the path through the *Commit* component *CW*—it changes the state of the database, but the order in which commits occur is not reflected in that state.

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 ranges. For example, we label the single path through the *Count* component *OW_{word,batch}* to indicate that while the output of *Count* is sensitive to the order in which its inputs appear in general, records with different values for the pair *word, batch* commute. 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 all records belong to the same 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,² the output would be deterministic. Hence

²This rules out races by ensuring that the query comes *after* all relevant click records.

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 5.

4.1.2 Sealed Stream Labels. Programmers can also supply optional annotations to describe the semantics of streams. The **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 **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.

Sealed stream labels are optional; by default, BLAZES labels all input streams as **Async**.

4.1.3 Derived Stream Labels. Given an annotated component with labeled input streams, BLAZES derives a label for each of its output streams. Figure 8 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 5, 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 on different executions or different stream instances. **Async** is conservatively applied as the default label; in general, we assume that communication between components is asynchronous.

Streams labeled **NDRead** may exhibit cross-run non-determinism, having different contents in different runs, or cross-instance non-determinism within a single run if the system is replicated. Finally, streams labeled **Taint** may exhibit persistent nondeterministic contents, or replica divergence in replicated systems.

4.2 Configuration Files

Programmers interact with BLAZES by providing *configuration files* that describe the dataflow topology and component path annotations. A configuration file comprises three sections:

- (1) *Declarations*, which associate each component path with an input interface, an output interface, and an annotation,
- (2) *Wirings*, which describe how components are connected in a global dataflow, and
- (3) *Labels*, describing the optional sealed stream labels for unconnected input interfaces.

For example, a configuration file for the Storm topology described in Section 1.2 is shown in Figure 9. Each of the three components defines a single labeled path. The wirings express the “dataflow glue,” capturing how the outputs of *Splitter* become the inputs of *Count*, whose outputs become the inputs of *Commit*. Finally, because the *tweet* interface of *Splitter* (as well as the *db* interface of *Commit*) is unconnected, it represents a global input (respectively, output) to the dataflow. BLAZES supplies the default *Async* label: inputs are nondeterministically ordered.

Obtaining such a configuration from a Storm topology is straightforward, even if done by hand. Clearly, the challenging part of writing a configuration file is choosing the annotations. 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). Last, *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*.

```

numbers
|
| Declarations:
|   Splitter:
|     - [ "tweet", "words", "CR" ]
|
|   Count:
|     - [ "words", "counts", "OW", ["word", "batch"]]
|
|   Commit:
|     - ["counts", "db", "CW" ]
|
| Wiring:
|   - ["Splitter", "words", "Count", "words"]
|   - ["Count", "counts", "Commit", "counts"]
|
| Labels:
|   - ["Splitter", "tweet", "Async"]

```

Fig. 9. Configuration file for the Storm word-count topology.

```

ex ::= stream | application | combination
application ::= (apply component ex)
combination ::= (combine string ex ex)
stream ::= taint | nontaint
nontaint ::= ndread | seal | async
labeled ::= ndread | seal
ndread ::= (NDRead label)
seal ::= (Seal label)
taint ::= Taint l (Taint string)
async ::= Async
component ::= (order op label)
order ::= C | O
op ::= R | W
label ::= (string ...)
e-cxt ::= (apply component e-cxt)
         | (combine string e-cxt ex)
         | (combine string stream e-cxt)
         | []

```

Fig. 10. Grammar for BLAZES.

```

let G = (V, E), a fresh directed labeled graph
function graph(file)
  while dec = next declaration do
    Add nodes dec.from, dec.to to V
  end while

  while wire = next wiring do
    Add edge (wire.from, wire.to, NONE) to E
  end while
end function

```

Fig. 11. Algorithm to convert an annotations configuration into a dataflow graph.

5 COORDINATION ANALYSIS AND CODE GENERATION

BLAZES constructs a dataflow graph from a system specification such as the one shown in Figure 9, and uses component and stream annotations to determine if the dataflow is guaranteed to produce deterministic outcomes. If it cannot make this guarantee, then it augments the program with coordination code. In this section, we describe how dataflow graphs are constructed from configuration files, how they are converted into a compact expression, and ultimately how the expression is analyzed to determine coordination locations. We conclude by describing how the output of the analysis is used to select a synchronization strategy when necessary.

The BLAZES metalanguage and analysis system are formalized in PLT Redex [26] and included in their entirety in this section.

5.1 Analysis Metalanguage

Figure 10 shows the grammar for the BLAZES metalanguage. An expression *ex* is a primitive stream label (Section 4), an *application* of an input stream to an annotated component path, or a *combination* of derived labels contributing to the same output interface. A component annotation captures the order sensitivity (O or C, for “order-sensitive” or “confluent”) and stateful effects (W or R, for “write” or “read-only”) of inputs on outputs. These annotations are either provided by the programmer (as described in Section 4) or inferred directly from the program (as described in Section 6).

Fig. 12. A dataflow graph for the Storm word-count topology, for which BLAZES yields the following expression: $(\text{apply } (C \ W) (\text{apply } (O \ W \ ('word' \ 'batch'))) (\text{apply } (C \ R) \ ???))$.

```

require G = (V, E), a labeled directed graph
function expr(interface)
  let Adj = { (from, label) | (from, to, label) ∈ E ∧ to = interface }
  if |Adj| = 0
    return user-specified label
  else
    return Combine(Apply(Adj1.label, expr(Adj1.input)), [ , ... ])
  end if
end function

```

Fig. 13. Algorithm to convert a dataflow graph into a BLAZES metalanguage expression.

Figure 11 shows the simple procedure for converting a configuration file into a labeled directed graph. The configuration file for the Storm word-count system yields the (trivial) linear dataflow shown in Figure 12.

Given a dataflow graph produced by the graph algorithm and an unconnected output interface (e.g., `Commit.db`), the `expr` algorithm (shown in Figure 13) computes a BLAZES metalanguage expression for that output stream. In the next section, we show how BLAZES reduces any such expression to a single stream label, representing a worst-case analysis of potential consistency anomalies that can occur for that interface, and how it identifies a potential location to place co-ordination code (e.g., a barrier or a call to an ordering API).

5.2 Reduction Relation

Once we have constructed an expression, determining an appropriate output label for a given interface is a matter of simplifying the expression according to a reduction relation. Figure 14 shows the complete reduction relation for BLAZES. In this section, we provide intuition for each of the rules.

5.2.1 Application Rules. *Application* rules model how the properties of streams change as they transit component paths. As BLAZES is a worst-case analysis tool, a stream label represents the most severe stream anomalies that can occur; similarly, applying a component to a stream label can only *increase* the severity of anomalies.

The first three rules capture the conditions under which an output stream may become “tainted” with nondeterministic *contents*. As we describe in Section 4.1.3, the consequences of a tainted stream will differ based on the fault tolerance strategy used. As we saw in Section 3, the most severe stream anomalies arise when replication is used to guard against component failure—in this case, nondeterministic contents can lead to replica divergence.

ND Witness captures the intuition that if nondeterministically-ordered stream crosses an order-sensitive, stateful component path, its output streams may (permanently) exhibit nondeterministic contents. **Propagate Taint** establishes that if an input stream has nondeterministic contents, so also may any output stream to which it contributes. **Long Taint** is the most interesting of these first three rules. It establishes that if an input stream exhibits (even transient) nondeterminism in

$(\text{apply } (O \ W \ _) \ \text{async}) \Longrightarrow \text{Taint}$	[ND Witness]
$(\text{apply } _ \ \text{taint}) \Longrightarrow \text{taint}$	[Propagate Taint]
$(\text{apply } (_ \ W \ _) \ \text{ndread}) \Longrightarrow \text{Taint}$	[Long Taint]
$(\text{apply } (O \ R \ \text{label}) \ \text{async}) \Longrightarrow (\text{NDRead } \text{label})$	[Transient ND]
$(\text{apply } (_ \ R \ _) \ (\text{NDRead } \text{label}_2)) \Longrightarrow (\text{NDRead } \text{label}_2)$	[Propagate ND]
$(\text{apply } (C \ _ \ _) \ (\text{Seal } \text{label}_1)) \Longrightarrow (\text{Seal } \text{label}_1)$	[Propagate Seal 1]
$(\text{apply } (O \ op \ \text{label}_1) \ (\text{Seal } \text{label}_2)) \Longrightarrow (\text{Seal } \text{label}_2)$ where $\text{compatible}?\llbracket \text{label}_2, \text{label}_1 \rrbracket$	[Propagate Seal 2]
$(\text{apply } (O \ op \ \text{label}_1) \ (\text{Seal } \text{label}_2)) \Longrightarrow (\text{apply } (O \ op \ \text{label}_1) \ \text{Async})$ where $\neg\llbracket \text{compatible}?\llbracket \text{label}_2, \text{label}_1 \rrbracket \rrbracket$	[Lose Seal]
$(\text{apply } (C \ _ \ _) \ \text{async}) \Longrightarrow \text{Async}$	[Confluent]
$(\text{combine } \text{string} \ \text{Taint} \ \text{stream}) \Longrightarrow (\text{Taint } \text{string})$	[Taint Trumps 1]
$(\text{combine } \text{string} \ \text{nontaint} \ \text{Taint}) \Longrightarrow (\text{Taint } \text{string})$	[Taint Trumps 2]
$(\text{combine } \text{string} \ (\text{Taint } \text{string}_2) \ \text{stream}) \Longrightarrow (\text{Taint } \text{string}_2)$	[Taint Trumps 3]
$(\text{combine } \text{string} \ \text{nontaint} \ (\text{Taint } \text{string}_2)) \Longrightarrow (\text{Taint } \text{string}_2)$	[Taint Trumps 4]
$(\text{combine } _ \ \text{async} \ \text{nontaint}) \Longrightarrow \text{nontaint}$	[Async Yields 1]
$(\text{combine } _ \ \text{labeled} \ \text{async}) \Longrightarrow \text{labeled}$	[Async Yields 2]
$(\text{combine } _ \ (\text{NDRead } \text{label}_1) \ (\text{NDRead } \text{label}_2)) \Longrightarrow (\text{NDRead } \text{union}\llbracket \text{label}_1, \text{label}_2 \rrbracket)$	[Merge NDRead]
$(\text{combine } _ \ (\text{Seal } \text{label}_1) \ (\text{Seal } \text{label}_2)) \Longrightarrow (\text{Seal } \text{union}\llbracket \text{label}_1, \text{label}_2 \rrbracket)$	[Compatible Seals]
$(\text{combine } _ \ (\text{Seal } \text{label}_1) \ (\text{NDRead } \text{label}_2)) \Longrightarrow (\text{Seal } \text{label}_1)$ where $\text{compatible}?\llbracket \text{label}_1, \text{label}_2 \rrbracket$	[Key/Gate Pass 1]
$(\text{combine } _ \ (\text{NDRead } \text{label}_2) \ (\text{Seal } \text{label}_1)) \Longrightarrow (\text{Seal } \text{label}_1)$ where $\text{compatible}?\llbracket \text{label}_1, \text{label}_2 \rrbracket$	[Key/Gate Pass 2]
$(\text{combine } _ \ (\text{Seal } \text{label}_1) \ (\text{NDRead } \text{label}_2)) \Longrightarrow (\text{NDRead } \text{label}_2)$ where $\neg\llbracket \text{compatible}?\llbracket \text{label}_1, \text{label}_2 \rrbracket \rrbracket$	[Key/Gate Fail 1]
$(\text{combine } _ \ (\text{NDRead } \text{label}_2) \ (\text{Seal } \text{label}_1)) \Longrightarrow (\text{NDRead } \text{label}_2)$ where $\neg\llbracket \text{compatible}?\llbracket \text{label}_1, \text{label}_2 \rrbracket \rrbracket$	[Key/Gate Fail 2]

Fig. 14. The complete reduction relation for BLAZES.

contents, it may permanently pollute a downstream stateful component (even a confluent one, as we discuss in Section 3.3 with the example of a replicated cache).

The next two rules identify and propagate the *NDRead* label, signifying transient nondeterminism in output stream contents. Transient ND captures the following intuition. Asynchronous requests arrive with nondeterministic order and timing. Even if calls to a particular API do not change the state of a component, their outcomes may have nondeterministic contents. Consider a component that models a set, exposing an API that allows a set add operation, as well as a read operation that returns the current set contents. We could label the add operation as (C, W) —it is confluent, since add is commutative. But we must label the read operation as O, R —it does not affect the state of the component, but its outputs are nondeterministic, as they depend on races between concurrent add and read operations. Propagate ND establishes that when streams with nondeterministic contents transit a read-only component, its output too may have nondeterministic contents.

The next three rules describe how seal labels are handled in BLAZES. Confluent components (Propagate Seal 1) and order-sensitive components whose *gate* label are compatible with the seal

Fig. 15. The lattice implied by the *combine* rules.

label (Propagate Seal 2) propagate the seal labels on their inputs. Order sensitive components, when presented with a sealed input stream with incompatible labels, behave the same as they would when presented with an *Async* input stream.

For example, the Storm word-count dataflow (which involves only a simple sequence of application rules) can be reduced in the following way once we supply an *Async* label for its input interface (replacing the “??” in Figure 12):

```

numbers
1 | (apply (C W ()) (apply (O W (word batch)) (apply (C R ()) Async))) [Confluent]
2 | (apply (C W ()) (apply (O W (word batch)) Async)) [ND Witness]
3 | (apply (C W ()) Taint) [Propagate Taint]
4 | Taint

```

5.2.2 Combination Rules. *Combination* rules model what happens when multiple input streams contribute to a single output stream in a dataflow. *Taint* is the strongest label: if any of the paths contributing to an output stream can exhibit permanent nondeterministic contents, then so can that output stream (*Taint Trumps*1–4). *Async* is the weakest, and will be dominated by any labels with which it combines (*Async Yields*1–2). When multiple paths with transient nondeterministic contents combine enroute to an output, that output stream has transient nondeterministic contents over the union of the attribute labels in both inputs (*Merge NDRead*). Sealed inputs are combined and propagated (*Combined Seals*).

The last four rules characterize what occurs when a sealed stream and a stream with transiently nondeterministic contents combine. If the seal annotation is compatible with the nondeterministic stream annotation (a notion we make precise in Section 5.4), then a local coordination strategy exists that delays emitting partitions of the input streams until a matching punctuation has arrived, preventing nondeterministic contents from exiting the component. Otherwise, the output stream may continue to exhibit transient nondeterministic contents.

The combination rules can be more compactly represented by the lattice shown in Figure 15. When streams with different labels combine enroute to an output interface, the more severe label dominates. The subtle case occurs when a *NDRead_{gate}* label combines with a *Seal_{key}*. If *key* and *gate* are compatible, then they merge to *Seal_{key}*; otherwise, they merge to *Taint*.

5.3 Confluence of the Reduction System

The reduction relation (Figure 14), together with the evaluation context *e-cxt* shown in Figure 10, establish that BLAZES’s reduction system is *strongly confluent*: every expression has a unique normal form [22]. A simple pairwise analysis of the rewriting rules (omitted for space considerations) shows that every BLAZES expression is either a primitive stream or has a *unique* reduction to a strictly simpler expression.

5.4 Transitivity of seals

When input streams are sealed, the reduction procedures tests whether the seal keys are compatible with the annotations of the component paths into which they flow.

Sealed streams can enable efficient, localized coordination strategies³ 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 $\text{injectivefd}(A, B)$, 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 A s, then we have also seen all the $f(A)$ for some injective f .

We may now define the predicate *compatible*:

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

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 6, 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.

5.5 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.

5.5.1 Sealing Strategies. If the programmer has provided a seal annotation **Seal_{key}** that is compatible with the (non-confluent) component annotation, then we may use a synchronization strategy that avoids global coordination. Consider a component representing a reporting server executing the query *WINDOW* from Section 1.2. Its label is $OR_{\text{id}, \text{window}}$. We know that *WINDOW* will produce deterministic output contents 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

³Of course, BLAZES can only exploit these strategies when they are available in the language under test. In our case studies, Bloom supports barrier synchronization natively, and Storm is modified to support it.

protocol must allow stream producers to communicate when a stream partition is sealed and what it contains, so 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 “stake holders” 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 (henceforth) immutable, it may process the partition without any further synchronization.

5.5.2 Ordering Strategies. If sealing strategies are not available, then 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 overcommits.

6 AUTOMATIC INFERENCE OF COMPONENT ANNOTATIONS

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.

Given an appropriately constrained language, however, 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.

6.1 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.

6.2 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” [42] the injective functional dependencies described in Section 5.4 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.

6.2.1 Confluence. As we described in Section 3.3, 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. In a SQL- or Datalog-like language, we could simply blacklist nonmonotonic operations (antijoin and group-by in SQL, negation in Datalog); any component whose implementation uses a blacklisted operation is marked as 0. Because Bloom extends Datalog with a

number of constructs borrowed from the Ruby language, we chose to ensure soundness by instead *whitelisting* the known monotonic operations (such as select, project, and join, including extended projections with arbitrary UDFs). Components implemented using only whitelisted operators are marked C (or O, otherwise). This static analysis is implemented as a query against the built-in Bloom catalog, and hence required no modifications to the interpreter.

6.2.2 Component State. Bloom distinguishes syntactically between transient event streams and stored state. A simple flow analysis—again, implemented as a Bloom query against the catalog—automatically determines if a component is stateful.

Let $p \rightarrow q$ if p and q are relations, and p appears in the body of a rule in which q appears in the head. For example, given the query `CREATE VIEW t AS SELECT * from r, s WHERE r.id=s.id`, we would have $r \rightarrow t$ and $s \rightarrow t$. Let $\rightarrow^*$ be the transitive closure of $\rightarrow$.

We give a component path from input i to output o the annotation W if and only if there exists a relation s such that

- (1) s is a persistent relation, storing state across requests.
- (2) $i \rightarrow^* s$ and $s \rightarrow^* o$.

Otherwise, it is given the annotation R.

Together, these analyses are sufficient to determine annotations (except for the subscripts, which we describe next) for every Bloom statement in a given module.

6.2.3 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 4.1.1 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.⁴ Applying this reasoning, BLAZES selects subscripts in the following way:

- (1) If the Bloom statement is an aggregation (*group by*), then the subscript is the set of grouping columns.
- (2) If the statement is an antijoin (*not in*), then the subscript is the set of columns occurring in the theta clause that are bound to the antijoin 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*($R.a, S.a$).

These tests are also implemented as queries against the system catalog.

7 CASE STUDIES

In this section, we apply BLAZES to the examples introduced in Section 1.2. In the context of these practical examples, we describe how programmers can reason about the appropriate annotations for dataflow components. We then discuss how BLAZES identifies the coordination requirements

⁴Intuitively, 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 $y = \text{'Yahoo!'}$, or (b) there will *never* be a corresponding $S.x$.

and, where relevant, the appropriate locations in these programs for coordination placement. In the next section (Section 8), we will 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 word-count 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 an adapter that can be applied to arbitrary Storm programs; 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. As we discuss in Section 6, BLAZES automatically extracts all the relevant annotations for programs written in Bloom.

For each dataflow, we present the BLAZES configuration file, containing the programmer-supplied annotations, and a visualization of the dataflow graph it implies. We then show the BLAZES metalanguage expression generated from the configuration file, and the sequence of reduction steps that simplify it to a primitive stream label.

7.1 Storm Word count

Recall that the analysis of the Storm word-count expression (Section 5.2) reduced to Taint. 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*, then BLAZES instead produces this reduction:

```

numbers
1 | (apply (C W) (apply (O W (word batch)) (apply (C R) (Seal (batch))))) [Propagate Seal 1]
2 | (apply (C W) (apply (O W (word batch)) (Seal (batch)))) [Lose Seal]
3 | (apply (C W) Async) [Confluent]
4 | Async

```

Because a batch is atomic (its contents may be completely determined once a punctuation 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.

7.2 Ad Reporting System

Next, we describe how we might annotate the various components of the ad reporting system. As we discuss in Section 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 6.

7.2.1 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 (from click to response) simply appends clicks and impressions to a log, we annotate this path *CW* (it modifies state, but that state is nevertheless convergent) also.


```

Declarations:
Cache:
- [ "request_i", "response_o", "CR" ]
- [ "response_i", "response_o", "CW" ]
- [ "request_i", "request_o", "CR" ]

Report:
- [ "click", "response", "CW" ]
POOR: [ "request", "response", "OR", ["id"]]
THRESH: [ "request", "response", "CR" ]
WINDOW: [ "request", "response", "OR", ["id", "window"]]
CAMPAIGN: [ "request", "response", "OR", ["id", "campaign"]]

Wiring:
- ["Cache", "request_o", "Report", "request"]
- ["Report", "response", "Cache", "response_i"]

Labels:
- ["Cache", "request_i", "Async"]
NOSEAL: ["Report", "click", "Async"]
WINDOW: ["Report", "click", "window"]
CAMPAIGN: ["Report", "click", "campaign"]

```

Fig. 16. The BLAZES configuration file for the ad reporting system.

Fig. 17. Logical dataflow graph for the Ad Reporting system. BLAZES uses Algorithm 13 to derive the following expression: (combine “Cache” (apply (CR) Async) (apply (CW) (combine “Report” (apply (CW) Async) (apply (OR (“id”)) (apply (CR) Async)))))).

All that remains is to annotate the path through the reporting component corresponding to the various continuous queries enumerated in Section 3.4. 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.

Because the reporting server queries are written in SQL, choosing component annotations is particularly straightforward. *THRESH* is a *monotonic* query. The value of the aggregate count() only increases over time, as records in clicks are not deleted. The predicate having count() > 1000 is, therefore, also confluent—it begins false, and if it becomes true it always remains true [23, 35].

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*.

The complete configuration file is shown in Figure 16. In addition to enumerating and labeling the component paths, the programmer also needs to indicate how the components interact using the wirings directive. The cache invokes the reporting server component by connecting its output interface *request_o* to the cache’s input interface *request*. Values returned by the reporting server flow from its output interface *response* to the cache’s input interface *response_i*.

Figure 17 shows a visualization of this dataflow. Note the correspondence with the manually-drawn representation of the same dataflow in Figure 4. The diamond labeled “S” represents the global dataflow *source* and is connected to all otherwise unconnected input interfaces (in this case,

clicks and request_i). The single unconnected output interface response_o is connected to the dataflow *sink* “T”—data flows from “S” to “T.”

7.2.2 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 (shown in Figure 17). If we supply *THRESH*, then BLAZES performs the following reductions, deriving a final label of **Async** for the output path from cache to sink:

```

numbers
1 | (combine Cache (apply (C R) Async) (apply (C W) (combine Report (apply (C W) Async) (apply (C R) (apply (C
   | R) Async)))))) [Confluent]
2 | (combine Cache Async (apply (C W) (combine Report (apply (C W) Async) (apply (C R) (apply (C R) Async))))))
   | [Confluent]
3 | (combine Cache Async (apply (C W) (combine Report Async (apply (C R) (apply (C R) Async)))))) [Confluent]
4 | (combine Cache Async (apply (C W) (combine Report Async (apply (C R) Async)))) [Confluent]
5 | (combine Cache Async (apply (C W) (combine Report Async Async))) [Async Yields 1]
6 | (combine Cache Async (apply (C W) Async)) [Confluent]
7 | (combine Cache Async Async) [Async Yields 1]
8 | Async

```

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

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

```

numbers
1 | (combine Cache (apply (C R) Async) (apply (C W) (combine Report (apply (C W) Async) (apply (O R (id))
   | (apply (C R) Async)))))) [Confluent]
2 | (combine Cache Async (apply (C W) (combine Report (apply (C W) Async) (apply (O R (id)) (apply (C R)
   | Async)))))) [Confluent]
3 | (combine Cache Async (apply (C W) (combine Report Async (apply (O R (id)) (apply (C R) Async))))))
   | [Confluent]
4 | (combine Cache Async (apply (C W) (combine Report Async (apply (O R (id)) Async)))) [Transient ND]
5 | (combine Cache Async (apply (C W) (combine Report Async (NDRead (id))))) [Async Yields 1]
6 | (combine Cache Async (apply (C W) (NDRead (id)))) [Long Taint]
7 | (combine Cache Async Taint) [Taint Trumps 2]
8 | (Taint Cache)

```

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 replica divergence will require a coordination strategy that controls message delivery order.

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

```

numbers
1 | (combine Cache (apply (C R) Async) (apply (C W) (combine Report (apply (C W) (Seal (campaign))) (apply (O R
   | (id campaign)) (apply (C R) Async)))))) [Confluent]
2 | (combine Cache Async (apply (C W) (combine Report (apply (C W) (Seal (campaign))) (apply (O R (id
   | campaign)) (apply (C R) Async)))))) [Propagate Seal 1]
3 | (combine Cache Async (apply (C W) (combine Report (Seal (campaign)) (apply (O R (id campaign)) (apply (C R)
   | Async)))))) [Confluent]
4 | (combine Cache Async (apply (C W) (combine Report (Seal (campaign)) (apply (O R (id campaign)) Async))))
   | [Transient ND]
5 | (combine Cache Async (apply (C W) (combine Report (Seal (campaign)) (NDRead (id campaign))))) [Key/Gate
   | Pass 1]
6 | (combine Cache Async (apply (C W) (Seal (campaign)))) [Propagate Seal 1]
7 | (combine Cache Async (Seal (campaign))) [Async Yields 1]
8 | (Seal (campaign))

```

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 5.5.

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

7.3 Bloom Shopping Cart

Finally, we revisit the replicated shopping cart implementations presented in Alvaro et al. [8]. In the first, a “destructive cart” is presented that makes use of a mutable key-value store (KVS) to store the state of a session. When a client session updates their cart, the value associated with the key (represented as an array) in the KVS is replaced. In the second, a “disorderly cart” instead simply appends to a log of add and delete operations, computing the contents of the cart via set difference on checkout [8]. The authors show how Bloom’s static analysis capabilities identify the coordination points in both implementations, and recognize that the destructive cart requires coordination at every operation, the disorderly cart requires it only at checkout.

We now 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. Better still, BLAZES synthesizes the required coordination code automatically.

The initial implementation of a shopping cart in Bloom given by Alvaro et al. reused a predefined component (the ReplicatedKVS module) 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. Alvaro et al. reimplement 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, they paid a software engineering penalty: they 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.

7.3.1 Component Annotations. BLAZES extracts the annotations shown in Figure 18 from its analysis of DestructiveCart. 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**).

7.3.2 Analysis. In the absence of sealing information, BLAZES warns us that replicas of the cart can diverge:

```

numbers
1 | (apply (C R) (combine KVS (apply (O W (key)) (apply (C R) Async)) (apply (O R (key)) (apply (C R) Async))))
   | [Confluent]
2 | (apply (C R) (combine KVS (apply (O W (key)) Async) (apply (O R (key)) (apply (C R) Async)))) [ND Witness]
3 | (apply (C R) (combine KVS Taint (apply (O R (key)) (apply (C R) Async)))) [Confluent]
4 | (apply (C R) (combine KVS Taint (apply (O R (key)) Async))) [Transient ND]
5 | (apply (C R) (combine KVS Taint (NDRead (key)))) [TaintTrumps I]
6 | (apply (C R) (Taint KVS)) [Propagate Taint]
7 | (Taint KVS)

```

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

```

Declarations:
  Cart:
    - ["action", "put", "CR"]
    - ["action", "get", "CR"]
    - ["get_response", "response", "CR"]
  KVS:
    - ["kvput", "kvget_response", "OW", ["key"]]
    - ["kvget", "kvget_response", "OR", ["key"]]
  Wiring:
    - ["Cart", "put", "KVS", "kvput"]
    - ["Cart", "get", "KVS", "kvget"]
    - ["KVS", "kvget_response", "Cart", "get_response"]
  Labels:
    - ["Cart", "action", "Async"]

```

Fig. 18. The complete configuration file for the Bloom shopping cart.

Fig. 19. Logical dataflow graph for the Bloom shopping cart. Expression: (apply (CR) (combine KVS (apply (OW (key)) (apply (CR) Async)) (apply (OR (key)) (apply (CR) Async))))).

updates. Analysis of the composed program reveals that `kvput.key` (the “gate” for the path from `kvput` to `kvget_response`) is injectively functionally determined by `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.

```

numbers
1| (apply (C R ()) (combine KVS (apply (O W (key)) (apply (C R ()) (Seal (key))))) (apply (O R (key)) (apply (C
| R ()) (Seal (key))))) [Propagate Seal 2]
2| (apply (C R ()) (combine KVS (apply (O W (key)) (Seal (key))) (apply (O R (key)) (apply (C R ()) (Seal
| (key))))) [Propagate Seal 1]
3| (apply (C R ()) (combine KVS (Seal (key)) (apply (O R (key)) (apply (C R ()) (Seal (key))))) [Propagate Seal 2]
4| (apply (C R ()) (combine KVS (Seal (key)) (apply (O R (key)) (Seal (key))))) [Propagate Seal 1]
5| (apply (C R ()) (combine KVS (Seal (key)) (Seal (key))) [Compatible Seals]
6| (apply (C R ()) (Seal (key))) [Propagate Seal 2]
7| (Seal (key))

```

7.4 Discussion

This last example demonstrates how BLAZES improves upon the analysis capabilities provided by Bloom. Monotonicity analysis [8] focused the attention of the programmer on program locations that may require coordination; Alvaro et al. showed how programmers could exploit this guidance to rewrite the destructive cart program to reduce its implicit coordination requirements. In exchange for improved performance and availability, they 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

Fig. 20. The effect of coordination on throughput for a Storm topology computing a streaming word count.

Fig. 21. Log records processed over time, 5 ad servers.

anomalies described in Section 3.1 can be rewritten into a program that achieves the required coordination using localized, barrier-based synchronization rather than enforcing a global ordering.

8 EVALUATION

In Section 3, 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.

8.1 Storm Word count

To evaluate the potential savings of avoiding unnecessary synchronization, we implemented two versions of the streaming word-count query described in Section 1.2. 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 Commi t components use coordination to ensure that outputs are committed to the backing store in a serial order.⁵ The second—which BLAZES has ensured will produce deterministic outcomes without any global coordination—is a “non-transactional 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 10min.

Figure 20 plots the throughput of the coordinated and uncoordinated implementations of the word-count 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 3×.

8.2 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

⁵Storm uses Zookeeper for coordination.

Fig. 22. Log records processed over time, 10 ad servers.

Fig. 23. Seal-based strategies, 10 ad servers.

introduced in Section 1.2. For ad servers, which simply generate click logs and forward them to reporting servers, we used 10 micro instances. We created three reporting servers using medium instances. Our Zookeeper cluster consisted of three 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 sleeping 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.

8.2.1 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 21 and 22 shows the log records processed over time for the uncoordinated run, for systems with 5 and 10 ad servers, respectively.

8.2.2 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 21 and 22 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.

8.2.3 Sealing Strategies. For the last experiments, we provided the input annotation *Seal_{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 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 21 and 22 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 ad-server, 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 system throughput.

Figure 23 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 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 10.

9 RELATED WORK

Distributed systems are a mature research area, but the study of distributed consistency remains extremely active. In this section, we cover foundational as well as emerging work related to BLAZES.

The state machine replication approach ensures consistency of arbitrary replicated systems by establishing a total order of message delivery [47]; each component implements a deterministic state machine, and a global coordination service, such as atomic broadcast [16, 33], MultiPaxos [36], or Raft [45], decides the message order.

To avoid the latency, throughput and availability penalties fundamental to approaches based on global coordination, much recent research has focused on identifying when consistency guarantees can be maintained even under loose ordering. Alvaro et al. [4] categorize these approaches into three classes. *Object-level* consistency exploits the algebraic properties of abstract datatypes to provide guarantees about component-local state, while *language-level* consistency leverages application-specific semantics to provide global guarantees. *Flow-level consistency*—of which BLAZES is an example—provides a middle ground.

Examples of object-level consistency approaches include convergent replicated data types (CRDTs) [48]. Any CRDT could be treated as a dataflow component annotated as CW in BLAZES. While CRDTs provide a guarantee of convergence (discussed Section 3), more recent work [14, 29] has focused on enforcing arbitrary constraints over object-local replicated state.

The Bloom language discussed in section 6 is one of many language-level approaches to distributed consistency. λ_{par} [35] 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.” Lasp [43] provides a functional language for confluent edge computation over CRDTs. Quelea [49]—much like BLAZES when used in concert

with the Bloom language—uses program analysis to identify the *weakest* consistency guarantees that are sufficient to ensure that application-specific invariants are upheld.

The object- and language-level approaches represent different points in the tradeoff space between generality and expressivity. Language-level strategies can provide system-wide correctness guarantees by exploiting application-specific semantics, but require “boiling the ocean”—rewriting large-scale systems in experimental languages. By contrast, object-level strategies can be incrementally added to existing systems, but composing global guarantees from object-local properties remains challenging [4, 23, 43]. Flow-level consistency approaches such as BLAZES occupy a sweet spot between these extremes.

Like reactive distributed systems, streaming databases [2, 10, 21] must operate over unbounded inputs—we have borrowed much of our stream formalism from this tradition. The CQL language distinguishes between monotonic and non-monotonic 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.

Similar to our work, the Gemini system [38] 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.

This article extends prior work on coordination analysis [7]. We present the complete term rewriting system (Section 5) used by BLAZES to identify when coordination is required to rule out stream anomalies. For each of the case study programs and configurations (Section 7), we show the complete derivation of the output labels, to illustrate how the term rewriting system obtained its result. We also present a new case study (Section 7.3), which connects BLAZES to prior work on Bloom [8].

10 CONCLUSION

In this article, we showed how the BLAZES framework enables the composition of globally consistent data processing pipelines out of heterogeneous components. We demonstrated how a simple language of annotations allows us to capture the relevant semantics of “black box” components and how emerging languages such as Bloom make it possible to infer these annotations automatically.

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 8.2.3, 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), then 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.⁶ More challenging and compelling is the possibility of again pushing these design principles into a compiler and automatically rewriting dataflows.

REFERENCES

- [1] Daniel Abadi. 2012. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *Computer* 45, 2 (2012), 37–42.
- [2] Daniel J. Abadi, Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. 2003. Aurora: A new model and architecture for data stream management. *VLDB J.* 12, 2 (Aug. 2003), 120–139.
- [3] Rakesh Agrawal et al. 2008. The Claremont report on database research. *SIGMOD Rec.* 37 (2008), 9–19.
- [4] Peter Alvaro, Peter Bailis, Neil Conway, and Joseph M. Hellerstein. 2013. Consistency without borders. In *Proceedings of the SoCC*.
- [5] Peter Alvaro, Tyson Condie, Neil Conway, Khaled Elmeleegy, Joseph M. Hellerstein, and Russell Sears. 2010a. BOOM analytics: Exploring Data-centric, Declarative Programming for the Cloud. In *Proceedings of the EuroSys*.
- [6] Peter Alvaro, Tyson Condie, Neil Conway, Joseph M. Hellerstein, and Russell Sears. 2010b. I do declare: Consensus in a logic language. *ACM SIGOPS Operating Systems Review* 43, 4 (2010), 25–30.
- [7] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and David Maier. 2014. Blazes: Coordination Analysis for Distributed Programs. In *Proceedings of the ICDE*.
- [8] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. 2011. Consistency analysis in bloom: A CALM and collected approach. In *Proceedings of the CIDR*.
- [9] Tom J. Ameloot, Frank Neven, and Jan Van den Bussche. 2011. Relational transducers for declarative networking. In *Proceedings of the PODS*.
- [10] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2006. The CQL continuous query language: Semantic Foundations and query execution. *VLDB J.* 15, 2 (June 2006).
- [11] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2014. Coordination avoidance in database systems. *Proceedings of the VLDB Endowment* (2014).
- [12] Jason Baker, Chris Bond, James C. Corbett, J. J. Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. 2011. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the CIDR*.
- [13] Magdalena Balazinska, Jeong-Hyon Hwang, and Mehul A. Shah. 2009. Fault-tolerance and high availability in data stream management systems. In *Encyclopedia of Database Systems*, Ling Liu and M. Tamer Özsu (Eds.). Springer U.S., 1109–1115.
- [14] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, and Marc Shapiro. 2015. Putting consistency back into eventual consistency. In *Proceedings of the EuroSys*.
- [15] Ken Birman, Gregory Chockler, and Robbert van Renesse. 2009. Toward a cloud computing research agenda. *SIGACT News* 40, 2 (June 2009), 68–80.
- [16] Kenneth P. Birman and Robert V. Renesse. 1994. *Reliable Distributed Computing with the ISIS Toolkit*. IEEE Computer Society Press, Los Alamitos, CA.
- [17] Bloom. 2012. Bloom programming language. Retrieved from <http://www.bloom-lang.org>.
- [18] Eric Brewer. 2012. CAP twelve years later: How the “rules” have changed. *IEEE Comput.* 45 (2012), 23–29.
- [19] Mike Burrows. 2006. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the OSDI*.
- [20] Rick Cattell. 2010. Scalable SQL and NoSQL data stores. *SIGMOD Rec.* (2010), 12–27.
- [21] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Sam Madden, Vijayshankar Raman, Fred Reiss, and Mehul Shah. 2003. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proceedings of the CIDR*.
- [22] A. Church and J. B. Rosser. 1936. Some properties of conversion. *Trans. Amer. Math. Soc.* (1936).

⁶Distributed garbage collection, based on sealing, is an avenue of future research.

- [23] Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. 2012. Logic and lattices for distributed programming. In *Proceedings of the SoCC*.
- [24] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's globally-distributed database. In *Proceedings of the OSDI*.
- [25] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Voshall, and Werner Vogels. 2007. Dynamo: Amazon's highly available key-value store. In *Proceedings of the SOSP*.
- [26] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. 2009. *Semantics Engineering with PLT Redex* (1st ed.). The MIT Press.
- [27] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of distributed consensus with one faulty process. *J. ACM* 32, 2 (April 1985), 374–382.
- [28] Message P. Forum. 1994. *MPI: A Message-Passing Interface Standard*. Technical Report. Knoxville, TN.
- [29] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 2016. 'Cause I'M strong enough: Reasoning about consistency choices in distributed systems. In *Proceedings of the POPL*.
- [30] Pat Helland. 2007. Life beyond distributed transactions: an apostate's opinion. In *Proceedings of the CIDR*.
- [31] Joseph M. Hellerstein. 2010. The declarative imperative: Experiences and conjectures in distributed logic. *SIGMOD Rec.* 39, 1 (2010), 5–19.
- [32] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the USENIX ATC*.
- [33] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. 2011. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the DSN*.
- [34] G. Kahn. 1974. The semantics of a simple language for parallel programming. In *Information Processing*, J. L. Rosenfeld (Ed.). North Holland, Amsterdam, Stockholm, Sweden, 471–475.
- [35] Lindsey Kuper and Ryan R. Newton. 2012. *A Lattice-Theoretical Approach to Deterministic Parallelism with Shared State*. Technical Report TR702. Indiana University. Retrieved from <http://www.cs.indiana.edu/cgi-bin/techreports/TRNNN.cgi?trnum=TR702>.
- [36] Leslie Lamport. 1998. The part-time parliament. *ACM Trans. Comput. Syst.* 16, 2 (May 1998), 133–169.
- [37] Jonathan Leibiusky, Gabriel Eisbruch, and Dario Simonassi. 2012. *Getting Started with Storm—Continuous Streaming Computation with Twitter's Cluster Technology*. O'Reilly.
- [38] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. 2012. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the OSDI*.
- [39] Jin Li, Kristin Tufte, Vladislav Shkapenyuk, Vassilis Papadimos, Theodore Johnson, and David Maier. 2008. Out-of-order Processing: A new architecture for high-performance stream systems. *PVLDB* 1, 1 (2008), 274–288.
- [40] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. 2006. Declarative networking: Language, execution and optimization. In *Proceedings of the SIGMOD*.
- [41] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. 2012. Distributed graphlab: A framework for machine learning and data mining in the cloud. In *Proceedings of the VLDB*.
- [42] David Maier, Alberto O. Mendelzon, and Yehoshua Sagiv. 1979. Testing implications of data dependencies. *ACM Trans. Database Syst.* 4 (1979), 455–469.
- [43] Christopher Meiklejohn and Peter Van Roy. 2015. Lasp: A language for distributed, coordination-free programming. In *Proceedings of the PPDP*.
- [44] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. 2010. S4: Distributed stream computing platform. In *Proceedings of the ICDMW*.
- [45] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *Proceedings of the USENIX ATC*.
- [46] Benjamin Recht, Christopher Re, Stephen J. Wright, and Feng Niu. 2011. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Proceedings of the NIPS*.
- [47] Fred B. Schneider. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.* 22, 4 (Dec. 1990).
- [48] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. *A comprehensive study of Convergent and Commutative Replicated Data Types*. Research report. INRIA.
- [49] K. C. Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. 2015. Declarative programming over eventually consistent data stores. In *Proceedings of the PLDI*.

- [50] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. 2003. Exploiting punctuation semantics in continuous data streams. *TKDE* 15, 3 (2003), 555–568.
- [51] Leslie G. Valiant. 1999. A bridging model for parallel computation. *Commun. ACM* (1999).
- [52] Werner Vogels. 2009. Eventually consistent. *CACM* 52, 1 (Jan. 2009), 40–44. <http://doi.acm.org/10.1145/1435417.1435432>.
- [53] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. 2012. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. In *Proceedings of the HotCloud*.
- [54] Daniel Zinn, Todd J. Green, and Bertram Ludäscher. 2012. Win-move is coordination-free (Sometimes). In *Proceedings of the ICDT*.

Received March 2016; revised February 2017; accepted June 2017