Blazes: Coordination Analysis for Distributed Programs

Peter Alvaro
Neil Conway
Joseph M. Hellerstein
David Maier

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2013-133
http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-133.html

July 16, 2013

Copyright © 2013, by the author(s).
All rights reserved.

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. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Blazes: Coordination analysis for distributed programs

Peter Alvaro
UC Berkeley
palvaro@cs.berkeley.edu

Neil Conway
UC Berkeley
nrc@cs.berkeley.edu

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

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

ABSTRACT

Fault tolerance is an essential feature of scalable software services. For a fault tolerance mechanism to be transparent, it must provide *consistency* across the results of redundant computations. Coordination protocols can ensure this consistency, but in practice they cause undesirable performance unless used judiciously. This raises significant challenges for distributed system architects and developers.

In this paper 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.

1. INTRODUCTION

> *The first principle of successful scalability is to batter the consistency mechanisms down to a minimum.*
> – James Hamilton, as transcribed in [5].

Scalable software services are made up of multiple independent components and must tolerate component failure gracefully. Many fault-tolerance techniques exist, but all of them—replicated databases, process pairs, log replay—share a basic strategy: *redundancy* of state and computation. The use of redundancy mechanisms brings with it a need to consider issues of *consistency*: when a service instance becomes unavailable, will the redundancy technique that replaces it produce output equivalent to that of the original? When this equivalence fails, the anomalies that arise can be significant and extremely difficult to debug.

The standard approach to ensuring consistency across nodes is to employ a distributed *coordination protocol* such as Paxos, atomic broadcast, or two-phase commit. However, these protocols need to be used very carefully, as they are associated with increased latency and reduced availability [5, 6]. Developers are thus faced with a difficult design decision: too little coordination allows hard-to-reproduce consistency anomalies, but too much coordination hinders performance and manageability. This work aims to provide programmers with program analysis tools to address these challenges.

1.1 Follow the Data

Consistency of program results depends on consistency of the data driving the computation. There has been significant work on consistency of replicated databases, but databases are only one of many services in a typical application. Programmers need to reason about the consistency of data as it transits across multiple services, potentially affecting persistent service state along the way.

Ideally, a programmer could submit arbitrary distributed code—including any services being used—to a program analysis tool that would guarantee consistency via just enough coordination in all the right places. This seems difficult or impossible in today's popular general-purpose programming languages. But perhaps it could be done in a more restricted programming abstraction?

Such abstractions are being developed in both research and industry. *Distributed stream processing* has recently emerged as a popular abstraction in the field, exemplified by Twitter's open source Storm system, Apache S4, and Spark Streaming. In these systems, a streaming *dataflow* of messages captures the interactions between arbitrary "black box" services. While it is not possible to automatically analyze the semantics of the black-box services, these systems make it easy to extract a program's data dependencies *across* services—a step in the right direction.

The adoption of stream processing hints at the potential for richer analysis via even higher-level abstractions. Bloom is a recent example of such an abstraction: a declarative language for distributed programming [2]. Bloom's roots are in logic programming, so its "components" are well-known relational algebra operators, and the dataflows between them are specified using a language rooted in Datalog with declarative additions for state update and asynchronous messaging. In Bloom programs, not only is it easy to follow the data through the components, but database theory makes it possible to reason about semantic properties of the components themselves.

1.2 Blazes

In this paper we present Blazes, a program-analysis framework that provides developers of distributed applications with judiciously chosen, *application-specific* coordination code. First, Blazes identifies code that may cause consistency anomalies by starting with properties of individual components, including order-sensitivity, statefulness, and replication; it reasons transitively about compositions of these properties across dataflows that span components. Second, Blazes generates consistency-preserving code to prevent anomalies with a minimum of coordination. The key intuition exploited by Blazes is that even when components are order-sensitive,

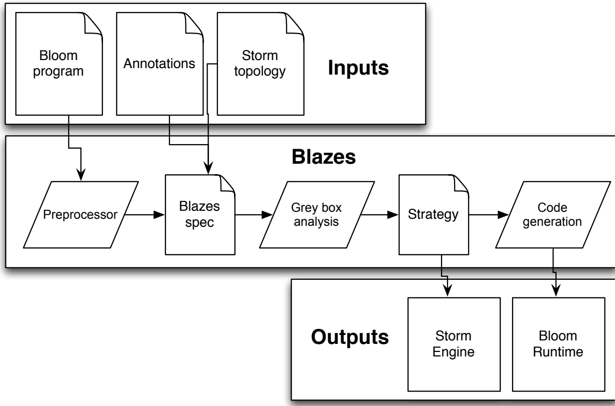


Figure 1: The BLAZES framework. In the "grey box" system, programmers supply a configuration file representing an annotated dataflow. In the "white box" system, this file is automatically generated via static analysis.

expensive global coordination (the conservative default in many systems) can often be avoided. In many cases, BLAZES can ensure deterministic outcomes via asynchronous point-to-point communication between producers and consumers—called *sealing*—that simply indicates when partitions of a stream have stopped changing. These partitions can be identified and "chased" through a dataflow via techniques from functional dependency analysis.

BLAZES can be used with existing stream-processing engines services, but it can also take advantage of the richer analyzability of declarative languages. 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. Bloom programmers are freed from the responsibility of annotations, since Bloom's formal language enables complete "white box" transparency for the component properties required by BLAZES. The BLAZES architecture is depicted in Figure 1.

In this paper we make the following contributions:

- We identify properties of dataflow components and streams that affect consistency, and introduce a term-rewriting technique over dataflow paths to translate component properties into end-to-end stream properties.
- We distinguish two alternative strategies for coordination: *ordering* and *sealing*, and show how we can take advantage of the cheaper sealing technique when possible.
- We present the BLAZES framework, and demonstrate its use with both a widely used stream processing system (Storm) and a forward-looking declarative DSL for distributed systems (Bloom).

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. Our experiments also reveal the subtle influence of data placement on application-specific coordination strategies.

1.3 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.

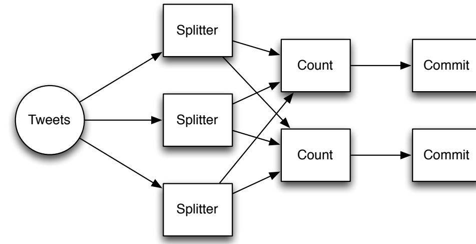


Figure 2: Physical architecture of a Storm word count topology

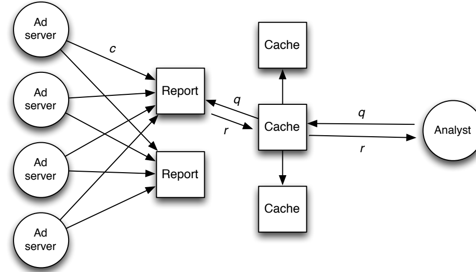


Figure 3: Physical architecture of an ad-tracking network

Streaming analytics with Storm: Figure 2 shows the architecture of a Storm topology that computes a streaming word count over the Twitter stream. Each "tweet" is associated with a numbered batch (the unit of replay) and is sent to exactly one `Splitter` component via random partitioning. The `Splitter` component divides tweets into their constituent words. These are hash partitioned to the `Count` component, which tallies the number of occurrences of each word in the current batch. When a batch ends, the `Commit` component records the batch number and frequency for each word in the batch in a backing store.

Storm ensures fault-tolerance via replay: if component instances fail or time out, stream sources redeliver their inputs. As a result, messages may be delivered more than once. Hence we are not concerned in this example with consistency of replicated state, but with ensuring that accurate counts are committed to the store despite the at-least-once delivery semantics. When implementing a Storm topology, the programmer must decide whether to make it *transactional*—i.e., one that processes tuples in atomic batches, ensuring that certain components (called *committers*) emit the batches in a total order. A programmer may, by recording the last successfully processed batch identifier, ensure at-most-once processing in the face of possible replay by incurring the extra overhead of synchronized batch processing.

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× 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

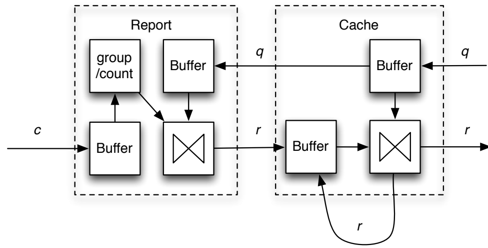


Figure 4: Dataflow representations of the ad-tracking network and its `Report` and `Cache` components.

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

2. SYSTEM MODEL

In this section, we revisit the familiar "black box" model of component-based distributed services, which provides the basis of the Blazes API. We use dataflow graphs [10] to represent distributed services. Nodes in the dataflow graph correspond to components, which expose input and output *interfaces*, corresponding to service calls or other message events. We can concisely represent both the data- and control-flow of the ad server network using the dataflow diagrams in Figure 4, in which nodes represent components while arcs represent streams.

The *logical dataflow* in Figure 4 captures the *software architecture* of the ad tracking network, describing how components interact via API calls. By contrast, the *physical dataflow* shown in Figure 3 captures the *system architecture*, mapping software components to the physical resources on which they will execute. Physical dataflows are descriptive but specific to a particular deployment, which is likely to evolve over time. We choose to focus our analysis on logical dataflows, which abstract away details like the multiplicity of component and stream instances. As we shall see, a properly annotated logical dataflow is sufficient to characterize the consistency semantics of distributed services.

2.1 Components

A *component* is a logical unit of computation and storage. Over time, a component processes a stream of inputs and produces a stream of outputs. To understand how inputs are transformed into outputs, we consider all the *paths* that connect a component's 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 from *q* to *r*).We assume that components are deterministic: two components that receive the same inputs in the same order produce the same outputs and reach the same state.

A *component instance* is a binding between a component and a physical resource on which the component executes. A component instance associates a component with a unique computational resource, with its own (potentially mutable) state and logical clock. In the ad system, the reporting server is a single logical component in Figure 4, but corresponds to two distinct (replicated) component instances in Figure 3.

Components may be stateful or stateless. A stateless path through a component does not change the component's internal state; a component is stateless if all paths through it are stateless.

2.2 Streams

Components are connected by *streams*, which are unbounded, unordered [14] collections of timestamped messages. A stream associates an output interface of one component with an input interface of another.

As we did with components, we differentiate between logical streams (which characterize the messages that flow between components and are uniquely defined by the component interfaces that they bridge) and *stream instances*, which correspond to physical channels between component instances. In an actual execution, individual components may execute on different machines as separate component instances, consuming stream instances with potentially different contents and orderings.

While streams (and hence dataflow executions) are theoretically unbounded, in practice they are often subdivided into partitions [4, 12, 23] to enable replay-based fault-tolerance. We will sometimes describe the properties of *runs*, or repeated executions over finite stream partitions.

A stream producer can optionally embed *punctuations* [20] into the stream. A punctuation guarantees that the producer will generate no more messages within a particular logical partition of the stream. For example, in Figure 3, a client might send an *end-of-session* punctuation on the *q* stream to indicate that they will generate no more requests. An ad server might indicate that they will henceforth produce no new records for a particular time window or advertising campaign via the *c* stream. When provided, these punctuations can enable efficient, local coordination strategies based on *sealing* (Section 6). Punctuations must contain metadata describing the contents of the partition that they seal, because (given our weak assumptions regarding stream order) a punctuation for a partition may arrive before some of the contents of that partition.

3. DATAFLOW CONSISTENCY

In this section, we develop consistency criteria appropriate to distributed dataflows. We begin by describing undesirable behaviors that can arise in streaming systems. We review common mechanisms for preventing such anomalies, generalizing these mechanisms into two classes of coordination strategies: message *ordering* and partition *sealing*. Finally, we consider a collection of queries that we could install at the reporting server in the ad tracking example presented in Section 1.3. We show how slight differences in the queries can lead to different distributed anomalies, and how practical variants of the ordering and sealing strategies can be used to prevent these anomalies.

3.1 Anomalies

Nondeterministic messaging interacts with fault-tolerance mechanisms in subtle ways. Two standard schemes exist for fault-tolerant dataflows: *replication* (used in the ad reporting system described in Section 1.3) and *replay* (employed by Storm and Spark) [4].

In both mechanisms, nondeterministic message order can cause inconsistencies—disagreement regarding stream contents among replicas or across replays, respectively. This disagreement undermines the transparency that fault tolerance mechanisms are meant to achieve.

We focus on three classes of anomalies:

1. *Cross-run nondeterminism*, in which nondeterminism in the execution causes components to produce, given the same inputs, different output stream *contents* in different runs. Systems that do not exhibit cross-run nondeterminism are *replayable*, and support efficient replay-based fault-tolerance. For obvious reasons, replayable systems are also easier to test and debug.

2. *Cross-instance nondeterminism*, in which multiple replicated components produce, given the same input, different output contents in the *same* execution. Cross-instance nondeterminism can lead to inconsistencies across queries.

3. *Split brain* phenomena, in which divergent replicated state leads to persistent inconsistency. 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 anomalies are *witnesses* of asynchronous execution; nondeterminism in message ordering has "leaked" into program outputs.

3.2 Monotonicity, confluence and convergence

Fortunately, a class of programs can be proven immune to the consistency anomalies described above. These "eventually consistent" programs produce a fixed final outcome regardless of any nondeterminism in message delivery ordering, and hence require no coordination. In recent work, we proposed the *CALM theorem*, which observes that *monotonic* programs produce consistent results regardless of nondeterminism in delivery orders [9]. Intuitively, monotonic programs compute a continually growing result, never retracting an earlier result given new inputs. Hence replicas running monotonic code always eventually agree, and replaying monotonic code produces the same result in every run.

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

Distributed systems commonly adopt a storage-centric notion of consistency. Replicated storage is *eventually consistent* or *convergent* if, when all messages have been delivered, all replicas agree on the set of stored values [22].

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, and the components may take "detours" along the way. Hence we must take care when reading "snapshots" of the state—which may exhibit cross-instance nondeterminism—while it is still changing. Consider what happens when the read-only outputs of a convergent component (e.g., GETs posed to a key/value store) flow into a replicated stateful component (e.g., a cache). If the replicas record different stream contents, the result is a split brain phenomenon.

Storage-centric consistency criteria focus on the properties of data at rest; reasoning about the overall properties of a composed dataflow requires following the data as it moves.

3.3 Coordination Strategies

Confluent components invariably produce deterministic outputs and convergent replicated state. How can we achieve these desirable properties for components that cannot be shown to be confluent? To prevent inconsistent outputs within or across program runs we need only address nondeterministic message orders, since we assume that components are deterministic. Hence coordination of non-confluent components can be achieved by removing the nondeterminism from their input orderings. Two extreme approaches include (a) establishing a single total order in which all instances of a given component receive messages (an *ordering* strategy) and (b) disallowing components from producing outputs until all of their inputs have arrived (a *sealing* strategy). The former—which enforces a total order of inputs—resembles state machine replication from the distributed systems literature [18], a technique for implementing consistent replicated services. The latter—which instead controls the order of evaluation at a coarse grain—resembles stratified evaluation of logic programs [21] in the database literature.

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

Fortunately, both coordination strategies have a dynamic variant that supports *live* systems that make incremental progress over time. To prevent replica divergence, it is sufficient to use a dynamic ordering service (e.g., Paxos) that decides a global order of messages *within a particular run*. This nondeterministic choice of message ordering prevents cross-instance nondeterminism but cannot prevent cross-run nondeterminism since the choice is dependent on arrival orders at the coordination service. Similarly, strategies based on sealing inputs can be applied to infinite streams as long as the streams can be partitioned into finite partitions that exhibit temporal locality, like windows with "slack" [1]. In cases where finite partitions are specified, sealing strategies can rule out all nondeterminism anomalies. Note that sealing is significantly less constrained than ordering: it enforces an output barrier per batch, but allows nondeterminism both in the arrival of a batch's inputs and in interleaving across batches.

3.4 Examples

The ad reporting system presented in Section 1.3 involves a collection of components interacting in a dataflow network. In this section, we focus on the `Report` component, which accumulates click logs and continually evaluates a standing query against them. Figure 5 presents a variety of simple queries that we might install at the reporting server; perhaps surprisingly, these queries have substantially different coordination requirements if we demand that they return deterministic answers. In Section 5, we will use Blazes to determine these requirements by treating each query as a separate instance of the `Report` component.

We consider first a threshold query *THRESH*, which computes the unique identifiers of any ads that have at least 1000 impressions. Although the click messages may arrive in different orders at different replicas or in different executions, *THRESH* returns results only when a count of messages exceeds a threshold. *THRESH* is confluent: we expect it to produce a deterministic result set without

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

Figure 5: Reporting server queries (shown in SQL syntax for familiarity).

need for coordination, since the value of the count monotonically increases in a manner insensitive to message arrival order [8].

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). This query is nonmonotonic: as more clicks are observed, the set of poorly performing ads might shrink. Because this query 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; if we install *POOR* at multiple reporting server replicas, they may report different answers in the same execution. To avoid such anomalies, a coordination service like Zookeeper can be used to ensure global message delivery order, and ensure that replicas remain in sync throughout execution. Unfortunately, enforcing a global delivery order 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 [1, 3, 7]. 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* before deciding if a particular advertisement is a poor performer. Query *WINDOW* returns, for each one hour window, those advertisement identifiers that have fewer than 100 clicks within that window.

The windowing strategy—a special case of *sealing*—ensures deterministic results by delaying the nonmonotonic query from processing a logical partition of the input stream until it is completely determined. This sealing technique may also be applied to partitions that are not explicitly temporal. 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 indeed may overlap or contain other campaigns. Nevertheless, once a campaign is over we can safely say things about it that we could not say while it was still "live."

4. ANNOTATED DATAFLOW GRAPHS

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

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 derives labels for each component's output streams.

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

Figure 6: The **C.R.O.W.** component annotations. A component path is either **C**onfluent or **O**rder-sensitive, and either changes component state (a **W**rite path) or does not (a **R**ead-only path). Component paths with higher *severity* annotations can produce more stream anomalies.

S	*Label*	*ND order*	*ND contents*	*Transient replica divergence*	*Persistent replica divergence*
0	$\textbf{NDRead}_{gate}$	**X**	**X**		
0	**Taint**	**X**	**X**		
1	$\textbf{Seal}_{key}$	**X**			
2	**Async**	**X**			
3	**Run**	**X**	**X**		
4	**Inst**	**X**	**X**	**X**	
5	**Split**	**X**	**X**	**X**	**X**

Figure 7: Stream labels, ranked by severity (S). $\textbf{NDRead}_{gate}$ and **Taint** are internal labels, used by the analysis system but never output. **Run, Inst** and **Split** correspond to the stream anomalies enumerated in Section 3.1: cross-run nondeterminism, cross-instance nondeterminism and split brain, respectively.

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 6 lists the component annotations supported by BLAZES. Each annotation applies to a path from an input interface to an output interface; if a component has multiple input or output interfaces, each path can have a different annotation.

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

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

that each record belongs to a different partition.

Consider a reporting server component implementing the query *WINDOW*. When it receives a request referencing a particular advertisement and window, it returns a response if the advertisement has fewer than 1000 clicks *within that window*. We would label the path from request inputs to outputs as $OR_{ad,window}$—a stateless non-confluent path operating over partitions with composite key *ad,window*. Requests do not affect the internal state of the component, but they do return potentially nondeterministic results that depend on the outcomes of races between queries and click records (assuming the inputs are nondeterministically ordered). Note however that if we were to delay the results of queries until we were certain that there would be no new records for a particular advertisement *or* a particular window,[1] the output would be deterministic. Hence *WINDOW* is "compatible" with click streams partitioned (and emitting appropriate punctuations) on *ad* or by *window*—this notion of compatibility will be made precise in Section 4.2.

4.1.2 Stream Annotations

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

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

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

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

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

4.1.3 Derived Stream Labels

Given an annotated component with labeled input streams, BLAZES can derive a label for each of its output streams. Figure 7 lists the derived stream labels—each corresponds to a class of anomalies that may occur in a given stream instance. 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 **Run** may exhibit cross-run nondeterminism, having different contents in different runs. Those labeled **Inst** may also exhibit cross-instance nondeterminism on different replicas within a single run. Finally, streams labeled **Split** may have split-brain behaviors (persistent replica divergence).

4.2 Analysis

We now describe how BLAZES automatically derives labels for the output streams of an annotated dataflow graph. If all output labels are **Async**, the service is guaranteed to produce deterministic outcomes. Otherwise, the analysis system identifies dataflow locations where adding coordination logic would achieve deterministic outcomes. It records these locations to assist in coordination selection (Section 6).

To derive labels for the output streams in a dataflow graph, BLAZES starts by enumerating all paths between pairs of sources and sinks. To rule out infinite paths, it reduces each cycle in the graph to a single node with a collapsed label by selecting the label of highest severity among the cycle members. Note that in the ad-tracking network dataflow shown in Figure 4, `Cache` participates in a cycle (the self-edge, corresponding to communication with other cache instances), but `Cache` and `Report` form no cycle, because `Cache` provides no path from r to q.

For each component whose input streams are determined (beginning with the components with unconnected inputs), BLAZES first performs an *inference* step, shown in Figure 8, for every path through the component. When it has done so, each of the output interfaces of the component is associated with a set of derived stream labels (at least one for each distinct path from an input interface, as well as the intermediate labels introduced by the *inference* rules). BLAZES then performs the second analysis step, the *reconciliation* procedure (described in Figure 9), which may add additional labels. Finally, the labels for each output interface are merged into a single label. This output stream becomes an input stream of the next component in the dataflow, and so on.

4.2.1 Transitivity of seals

When input streams are sealed, the inference and reconciliation procedures test whether the seal keys are compatible with the annotations of the component paths into which they flow. Sealed streams can enable efficient, localized coordination strategies when the sealed partitions are *independent*—a property not just of the streams themselves but of the components that process them. For example, given the queries in Figure 5, an input stream sealed on *campaign* is only 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. 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. For example, given a stream sealed on key *key* (with annotation $\textbf{Seal}_{key}$) flowing into a component with annotation OW_{gate}, under what circumstances are deterministic outputs guaranteed? To answer this question, we must formalize the transitivity of sealed partitions.

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. A trivial (and ubiquitous) example of an injective function between input and output attributes is the identity function, which is applied whenever we project an attribute without transformation—we will focus our discussion on this example.

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

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

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

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

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

Figure 8: Reduction rules for component paths. Each rule takes an input stream label and a component annotation, and produces a new (internal) stream label. Rules may be read as implications: if the premises (expressions above the line) hold, then the conclusion (below) should be added to the `Labels` list.

We may now define the predicate *compatible*:

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

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

4.2.2 Inference

At each reduction step, we apply the rules in Figure 8 to derive additional intermediate stream labels for a component path. An intermediate stream label may be any of the labels in Figure 7.

Rules 1 and 2 of Figure 8 reflect the consequences of providing nondeterministically ordered inputs to order-sensitive components. **Taint** indicates that the internal state of the component may become corrupted by unordered inputs. $\textbf{NDRead}_{gate}$ indicates that the output stream may have transient nondeterministic contents. Rule 3 captures the interaction between cross-instance nondeterminism and split brain: transient disagreement among replicated streams can lead to permanent replica divergence if the streams modify component state downstream. Rules 4 tests whether the predicate *compatible* (defined in the previous section) holds, in order to determine when sealed input streams are compatible with stateful, non-confluent components.

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

4.2.3 Reconciliation

Given an output interface associated with a set of labels, Blazes derives additional labels by using the *reconciliation* procedure shown in Figure 9.

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

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

4.2.4 Notation

$$\text{protected}(\textbf{NDRead}_{gate}) \equiv \forall l \in \texttt{Labels}\ \ l = \textbf{NDRead}_{gate} \lor \exists key\ l = \textbf{Seal}_{key} \land \text{compatible}(gate, key)$$

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

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

Figure 9: The *reconciliation* procedure applies the rules above to the set `Labels`, possibly adding additional labels. "*Rep* ? **A** : **B**" means 'if *Rep*, add **A** to `Labels`, otherwise add **B**.' Finally, *reconciliation* returns the elements in `Labels` with highest severity.

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

$$\frac{(R_1)\ \dfrac{SL_1 \qquad CA_1}{SL_2} \qquad (R_2)\ \dfrac{SL_3 \qquad CA_2}{SL_4} \qquad [\ldots]}{SL_5}\ CN_1$$

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

5. CASE STUDIES

In this section, we apply Blazes to the examples introduced in Section 1.3. We describe how programmers can manually annotate dataflow components. We then discuss how Blazes identifies the coordination requirements and, where relevant, the appropriate locations in these programs for coordination placement. In the next section we will consider how Blazes chooses safe yet relatively inexpensive coordination protocols. Finally, in Section 8 we show concrete performance benefits of the Blazes coordination choices as compared to a conservative use of a coordination service like Zookeeper.

We implemented the Storm wordcount dataflow, which consists of three "bolts" (components) and two distinct "spouts" (stream sources, which differ for the coordinated and uncoordinated implementations) in roughly 400 lines of Java. We extracted the dataflow metadata from Storm into Blazes via a reusable adapter; we describe below the output that Blazes produced and the annotations we added manually. We implemented the ad reporting system entirely in Bloom, in roughly 125 LOC. As discussed in the previous section, Blazes automatically extracted all the relevant annotations.

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

5.1 Storm wordcount

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

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

5.1.1 Component annotations

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

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

5.1.2 Analysis

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

```
         (p) Async      CR
             -------------
Splitter      Async
             -------
              Async        OW_word,batch
         (2) ---------------------------
              Count   Taint
                     -------
                  (p)  Run               CW
                     -----------------------
              Committer   Run
                         -----
                          Run
```

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

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

```
         (p) Seal_batch    CR
             ----------------
Splitter      Seal_batch
             ------------
              Seal_batch       OW_word,batch
         (p) -------------------------------
              Count   Async
                     -------
                  (p) Async               CW
                     ------------------------
              Committer   Async
                         -------
                          Async
```

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

5.2 Ad-reporting system

Next we describe how we might annotate the various components of the ad-reporting system. As we discuss in Section 7, 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 5.

5.2.1 Component annotations

The cache is clearly a stateful component, but since its state is append-only and order-independent (this extremely simple cache provides no facility for deletion or modification of cache entries) we may annotate it *CW*. Because the data-collection path through the reporting server simply appends clicks and impressions to a log, we annotate this path *CW* (it modifies state, but in a confluent way) also.

All that remains is to annotate the path through the reporting component corresponding to the various continuous queries enumerated in Section 3.4. `Report` is a replicated component, so we supply the **Rep** annotation for all instantiations. We annotate the query path corresponding to *THRESH CR*—it is both read-only (in the context of the reporting module) and confluent. It never emits a record until the ad impressions have reached the given threshold, and since they can never again drop below the threshold, the query answer "ad X has > 1000 impressions" holds forever.

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

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

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

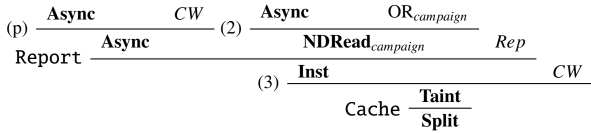
5.2.2 Analysis

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. If we supply *THRESH*, BLAZES performs the following reductions, deriving a final label of **Async** for the output path from cache to sink:

```
     (p) Async        CW   (p) Async        CW
         ---------------       ---------------
           Async                 Async          Rep
Report   ----------------------------------------
                           Async                    CW
                       (p) --------------------------
                           Cache   Async
                                  -------
                                   Async
```

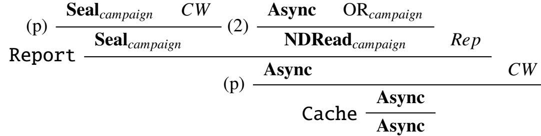
All components are confluent, so the complete dataflow produces deterministic outputs without coordination. If we chose, we could encapsulate the service as a single component with annotation *CW*.

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



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

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



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 6.

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

6. 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.

6.1 Ordering Strategies

If the programmer has not provided any seal annotations, BLAZES achieves replica convergence 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 builtin support for "transactional" topologies, which enforces a total order over commits.

6.2 Sealing Strategies

If the programmer has provided a seal annotation $\mathbf{Seal}_{key}$ that is compatible with the (non-confluent) component annotation, we may use a synchronization strategy that avoids global coordination. Consider a component representing a reporting server executing the query *WINDOW* from Section 1.3. Its label is $OR_{id,window}$. 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 protocol must allow stream producers to communicate when a stream partition is sealed and what it contains, so that consumers can determine when the complete contents of a partition are known.

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

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

7. BLOOM INTEGRATION

To provide input for the "grey box" functionality of BLAZES, programmers must convert their intuitions about component behavior and execution topology into the annotations introduced in Section 4. As we saw in Section 5.1.1, this process is often quite natural; unfortunately, as we learned in Section 5.2.1, it becomes increasingly burdensome as component complexity increases.

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

7.1 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.

7.2 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" [15] the injective functional dependencies described in Section 4.2.1 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.

7.2.1 Confluence and state

As we described in Section 3.2, the CALM theorem establishes that all *monotonic* programs are confluent. The Bloom runtime includes analysis capabilities to identify—at the granularity of program statements—nonmonotonic operations, which can be conservatively identified with a syntactic test. A component free of such operations is provably order-insensitive. Similarly, Bloom distinguishes syntactically between transient event streams and stored state. A simple flow analysis automatically determines if a component is stateful. Together, these analyses are sufficient to determine annotations (except for the subscripts, which we describe next) for every Bloom statement in a given module.

7.2.2 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 effectively computable 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.[3] Applying this reasoning, Blazes selects subscripts in the following way:

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

2. If the statement is an antijoin (*not in*), the subscript is the set of columns occurring in the theta clause.

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 $injectivefd(R.a, S.a)$.

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 wordcount

To evaluate the potential savings of avoiding unnecessary synchronization, we implemented two versions of the streaming wordcount query described in Section 1.3. Both process an identical stream of tweets and produce the same outputs. They differ in that the first implementation is a "transactional topology," in which the `Commit` components use coordination to ensure that outputs are committed to the backing store in a serial order. [4] The second—which Blazes has ensured will produce deterministic outcomes without any global coordination—is a "nontransactional topology." We optimized the batch size and cluster configurations of both implementations to maximize throughput.

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

Figure 10 plots the throughput of the coordinated and uncoordinated implementations of the wordcount dataflow as a function of the cluster size. The overhead of conservatively deploying a transactional topology is considerable. The uncoordinated dataflow has a peak throughput roughly 1.8 times that of its coordinated counterpart in a 5-node deployment. As we scale up the cluster to 20 nodes, the difference in throughput grows to 3X.

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

[4]Storm uses Zookeeper for coordination.

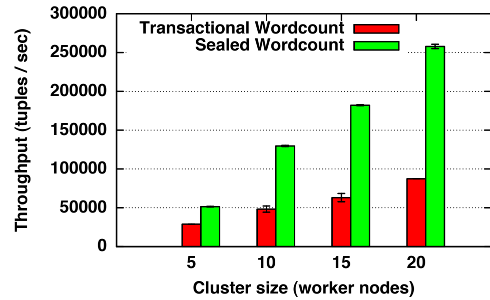


Figure 10: The effect of coordination on throughput for a Storm topology computing a streaming wordcount.

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 introduced in Section 1.3. For ad servers, which simply generate click logs and forward them to reporting servers, we used 10 `micro` instances. We created 3 reporting servers using `medium` instances. Our Zookeeper cluster consisted of 3 `small` instances. All instances were located in the same AWS availability zone.

Ad servers generate a workload of 1000 log entries per server. Servers batch messages, dispatching 50 click log messages at a time, and 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 11 and 12 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 11 and 12 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*

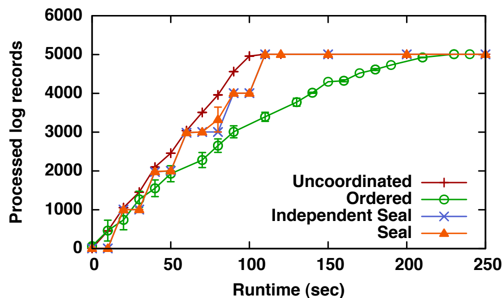


Figure 11: Log records processed over time, 5 ad servers

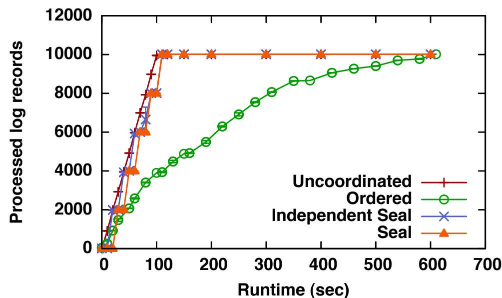


Figure 12: Log records processed over time, 10 ad servers

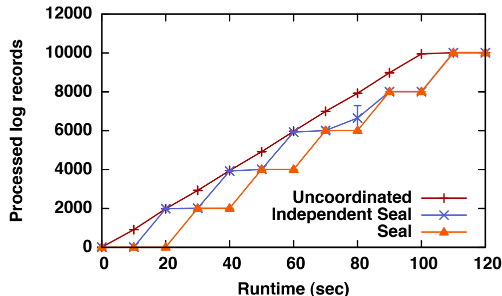


Figure 13: Seal-based strategies, 10 ad servers

For the last experiments we provided the input annotation $\textbf{Seal}_{campaign}$ 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 11 and 12 compare the performance of seal-based strategies to ordered and uncoordinated runs. We plot two topologies: "Independent seal" corresponds to a partitioning in which each campaign is mastered at exactly one adserver, while in "Seal," all ad servers produce click records for all campaigns. Note that both runs that used seals closely track the performance of the uncoordinated run; doubling the number of ad servers has little effect on the system throughput.

Figure 13 plots the 10-server run but omits the ordering strategy, to highlight the differences between the two seal-based topologies. As we would expect, "independent seals" result in executions with slightly lower latencies because reporting servers may process partitions as soon as a single seal message appears (since each partition has a single producer). By contrast, the step-like shape of the non-independent seal strategy reflects the fact that reporting servers delay processing input partitions until they have received a seal record from every producer. Partitioning the data across ad servers so as to place advertisement content close to consumers (i.e., partitioning by ad id) caused campaigns to be spread across ad servers. This partitioning conflicted with the coordination strategy, which would have performed better had it associated each campaign with a unique producer. We revisit the notion of "coordination locality" in Section 10.

9. RELATED WORK

Our approach to automatically coordinating distributed services draws inspiration from the literature on both distributed systems and databases. Ensuring consistent replica state by establishing a total order of message delivery is the technique adopted by state machine replication [18]; each component implements a deterministic state machine, and a global coordination service such as atomic broadcast or Multipaxos decides the message order.

In the context of Dedalus, Marczak et al. draw a connection between stratified evaluation of conventional logic programming languages and distributed protocols to ensure consistency [16]. They describe a program rewrite that ensures deterministic executions by preventing any node from performing a nonmonotonic operation until that operation's inputs are "determined." Agents processing or contributing to a distributed relation carry out a voting-based protocol to agree when the contents of the relation are completely determined. This rewrite—essentially a restricted version of the sealing construct defined in this paper—treats entire input collections as sealable partitions, and hence is not defined for unbounded input relations.

Commutativity of concurrent operations is a subject of interest for parallel as well as distributed programming languages. Commutativity analysis [17] uses symbolic analysis to test whether different method-invocation orders always lead to the same result; when they do, lock-free parallel executions are possible. λ_{par} [11] is a parallel functional language in which program state is constrained to grow according to a partial order and queries are restricted, enabling the creation of programs that are "deterministic by construction." CRDTs [19] are convergent replicated data structures; any CRDT could be treated as a dataflow component annotated as *CW*.

Like reactive distributed systems, streaming databases [1, 3, 7] must operate over unbounded inputs—we have borrowed much of our stream formalism from this tradition. The CQL language distinguishes between monotonic and nonmonotonic operations; the former support efficient strategies for converting between streams and relations due to their pipelineability. The Aurora system also distinguishes between "order-agnostic" and "order-sensitive" relational operators.

Similarly to our work, the Gemini system [13] 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.

10. CONCLUSIONS

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 rough measure being the number of nodes that must communicate to deterministically process a segment of data. If coordination locality is in conflict with spatial locality (e.g., the non-independent partitioning strategy that clusters ads likely to be served together at the cost of distributing campaigns across multiple nodes), problems emerge.

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

11. REFERENCES

[1] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. B. Zdonik. Aurora: a New Model and Architecture for Data Stream Management. *VLDB Journal*, 12(2):120–139, Aug. 2003.

[2] P. Alvaro, N. Conway, J. M. Hellerstein, and W. R. Marczak. Consistency Analysis in Bloom: a CALM and Collected Approach. In *CIDR*, 2011.

[3] A. Arasu, S. Babu, and J. Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *VLDB Journal*, 15(2), June 2006.

[4] M. Balazinska, J.-H. Hwang, and M. A. Shah. Fault-Tolerance and High Availability in Data Stream Management Systems. In L. Liu and M. T. Özsu, editors, *Encyclopedia of Database Systems*, pages 1109–1115. Springer US.

[5] K. Birman, G. Chockler, and R. van Renesse. Toward a Cloud Computing Research Agenda. *SIGACT News*, 40(2):68–80, June 2009.

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

[7] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[8] N. Conway, W. R. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier. Logic and Lattices for Distributed Programming. In *SoCC*, 2012.

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

[10] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, Aug 1974. North Holland, Amsterdam.

[11] L. Kuper and R. R. Newton. A Lattice-Theoretical Approach to Deterministic Parallelism with Shared State. Technical Report TR702, Indiana University, Oct. 2012.

[12] J. Leibiusky, G. Eisbruch, and D. Simonassi. *Getting Started with Storm - Continuous Streaming Computation with Twitter's Cluster Technology*. O'Reilly, 2012.

[13] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making Geo-replicated Systems Fast as Possible, Consistent when Necessary. In *OSDI*, 2012.

[14] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, and D. Maier. Out-of-order Processing: a New Architecture for High-performance Stream Systems. *PVLDB*, 1(1):274–288, 2008.

[15] D. Maier, A. O. Mendelzon, and Y. Sagiv. Testing Implications of Data Dependencies. *ACM Transactions on Database Systems*, 4:455–469, 1979.

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

[17] M. C. Rinard and P. C. Diniz. Commutativity Analysis: a New Analysis Technique for Parallelizing Compilers. *ACM Trans. Program. Lang. Syst.*, 19(6):942–991, Nov. 1997.

[18] F. B. Schneider. Implementing Fault-tolerant Services Using the State Machine Approach: a Tutorial. *ACM Comput. Surv.*, 22(4), Dec. 1990.

[19] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Research report, INRIA, 2011.

[20] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting Punctuation Semantics in Continuous Data Streams. *TKDE*, 15(3):555–568, 2003.

[21] J. D. Ullman. *Principles of Database and Knowledge-Base Systems: Volume II: The New Technologies*. W. H. Freeman & Co., 1990.

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

[23] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized Streams: an Efficient and Fault-tolerant Model for Stream Processing on Large Clusters. In *HotCloud*, 2012.

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