# Blazes: Coordination Analysis for Distributed Programs

Peter Alvaro #1, Neil Conway #2, Joseph M. Hellerstein #3, David Maier *4

# *UC Berkeley*

[1] palvaro@cs.berkeley.edu
[2] nrc@cs.berkeley.edu
[3] hellerstein@cs.berkeley.edu

* *Portland State University*

[4] maier@cs.pdx.edu

***Abstract*—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 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.**

## I. INTRODUCTION

*The first principle of successful scalability is to batter the consistency mechanisms down to a minimum.*
– James Hamilton, as transcribed in [1].

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

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.

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 [4], [5], [6], [7]. 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 paper, 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 [1], [8], [9]. Coordination services for distributed consistency, such as Chubby [10] and Zookeeper [11], are in wide use. At the same time, there have been various efforts to address consistency in system-specific ways, including NoSQL systems [12], Internet services infrastructure [10], [13], [14], [15] and even large-scale machine learning systems [16], [17]. The reason for the interest is clear: for many practitioners distributed consistency is the most critical issue for system performance and manageability at scale [1].

### A. Blazes

Recent work has highlighted promising connections between distributed consistency and database theory surrounding monotonicity [6], [18], [19], [20]. In this paper we move beyond theory and language design to develop practical techniques that have direct utility for popular distributed programming platforms like Twitter Storm [21], while providing even more power for the declarative languages like Bloom [22] 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 [21], Apache S4 [23], and Spark Streaming [24]. Programmers of stream processing engines interact with BLAZES in a "grey box" manner: they provide simple semantic *annotations* to the black-box components in their dataflows, and BLAZES performs the analysis of all dataflow paths through the program. BLAZES can also take advantage of the richer analyzability of declarative languages like Bloom. Bloom programmers are freed from the need to supply annotations, since Bloom's language semantics allow BLAZES to infer component properties automatically.

We make the following contributions in this paper:

- **Consistency Anomalies and Properties.** We present a spectrum of consistency anomalies that arise in distributed dataflows. We identify key properties of both streams and components that affect consistency.
- **Composition of Properties.** We show how to analyze the composition of consistency properties in complex programs via a term-rewriting technique over dataflow paths, which translates local component properties into end-to-end stream properties.
- **Custom Coordination Code.** We distinguish two alternative coordination strategies, *ordering* and *sealing*, and show how we can automatically generate application-aware coordination code that uses the cheaper sealing technique in many cases.

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

### B. Running Examples

We consider two running examples: a streaming analytic query implemented using the Storm stream processing system and a distributed 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 continuous 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—which divides tweets into their constituent words—via random partitioning. The words 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 a backing store.

Storm ensures fault-tolerance via replay: if component instances fail or time out, stream sources redeliver their inputs.

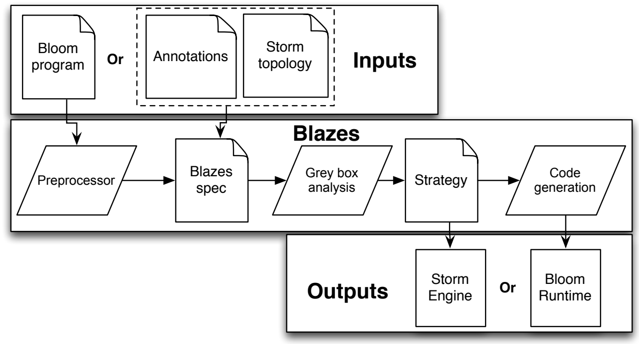


Fig. 1: The BLAZES framework. In the "grey box" system, programmers supply a configuration file representing an annotated dataflow. In the "white box" system, this file is automatically generated via static analysis.

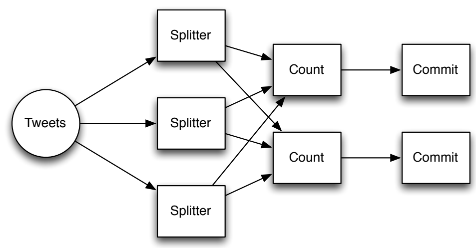


Fig. 2: Physical architecture of a Storm word count topology.

It is up to the programmer to ensure that accurate counts are committed to the store despite these at-least-once delivery semantics. One approach is to make the Storm topology *transactional*—i.e., one that processes tuples in atomic batches, ensuring that certain components (called *committers*) emit the batches in a total order. By recording the last successfully processed batch identifier, a programmer may ensure at-most-once processing in the face of possible replay by incurring the extra overhead of synchronizing the processing of batches.

Note that batches are independent in the word counting application; because the streaming query groups outputs by batch id, there is no need to order batches with respect to each other. BLAZES can aid a topology designer in avoiding unnecessarily conservative ordering constraints, which (as we will see in Section VIII) results in up to a $3\times$ improvement in throughput in our experiments.

**Ad-tracking with Bloom:** Figure 3 depicts an ad-tracking network, in which a collection of *ad servers* deliver advertisements to users (not shown) and send click logs (edges labeled "*c*") to a set of *reporting server* replicas. 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 common 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

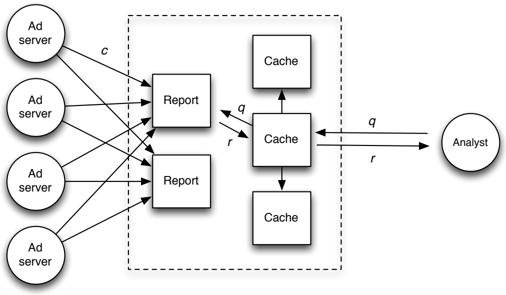


Fig. 3: Physical architecture of an ad-tracking network.

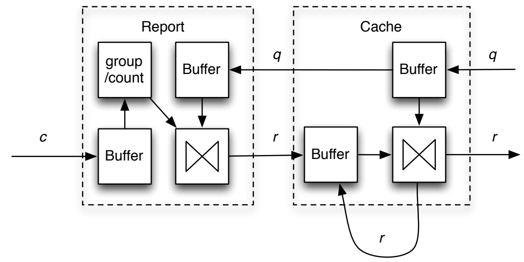


Fig. 4: Dataflow representations of the ad-tracking network's `Report` and `Cache` components.

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*—each partition of which is generated by a single ad server—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 ad servers, network delivery order is nondeterministic. As we shall see, different continuous queries have different sensitivities to network nondeterminism. BLAZES can help determine how much coordination is required to ensure that network behavior does not cause inconsistent results.

## II. System Model

The BLAZES API is based on a simple "black box" model of component-based distributed services. We use dataflow graphs [25] 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.

A *logical dataflow* (e.g., the representation of the ad tracking network depicted in Figure 4) captures a *software architecture*, describing how components interact via API calls. By contrast, a *physical dataflow* (like the one shown in Figure 3) extends a software architecture into a *system architecture*, mapping software components to the physical resources on which they will execute. We choose to focus our analysis on logical dataflows, which abstract away details like the multiplicity of physical resources but are sufficient—when properly annotated—to characterize the consistency semantics of distributed services.

### A. 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 [26] 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 from *q* to *r*). We assume that components are deterministic: two instances of a given component that receive the same inputs in the same order produce the same outputs and reach the same state.

A *component instance* is a binding between a component and a physical resource—with its own clock and (potentially mutable) state—on which the component executes. In the ad system, the reporting server is a single logical component in Figure 4, but corresponds to two distinct (replicated) component instances in Figure 3. Similarly, we differentiate between logical streams (which characterize the types of the messages that flow between components) and *stream instances*, which correspond to physical channels between component instances. Individual components may execute on different machines as separate component instances, consuming stream instances with potentially different contents and orderings.

While streams are unbounded, in practice they are often divided into batches [21], [24], [27] to enable replay-based fault-tolerance. *Runs* are (possibly repeated) executions over finite stream batches.

A stream producer can optionally embed *punctuations* [28] 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, 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 V-B, we show how punctuations can enable efficient, localized coordination strategies based on *sealing*.

## III. Dataflow Consistency

In this section, we develop consistency criteria and mechanisms appropriate to distributed, fault-tolerant dataflows. We begin by describing undesirable behaviors that can arise due to the interaction between nondeterministic message orders and fault-tolerance mechanisms. We review common strategies for preventing such anomalies by exploiting semantic properties of components (Section III-B) or by enforcing constraints on message delivery (Section III-C). We then generalize delivery mechanisms into two classes: message *ordering* and partition

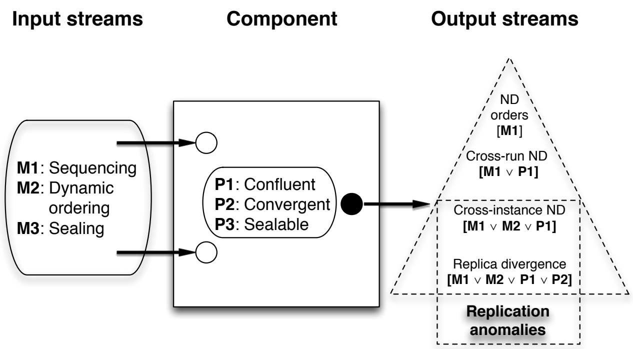


Fig. 5: The relationship between potential stream anomalies (right) and remediation strategies based on component properties (center) and delivery mechanisms (left). For each anomaly, we list the properties and mechanisms that *prevent* it. For example, convergent components (**P2**) prevent only replica divergence while a dynamic message ordering mechanism (**M2**) prevents all replication anomalies.

*sealing*. Finally, we consider a collection of queries that we could install at the reporting server in the ad tracking example presented in Section I-B. 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.

### *A. 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 I-B) and *replay* (employed by Storm and Spark) [27]. 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.

Figure 5—which depicts a component with two input streams and a single output stream—captures the relationship between delivery mechanisms, component properties and output stream anomalies. The spectrum of behaviors that can arise as a result of nondeterministic message ordering are listed on the right side of the figure. Because it is difficult to control the order in which a component's inputs appear, the first (and least severe) anomaly is nondeterministic orderings of (otherwise deterministic) output contents (**Async**). In this paper, we focus on the three remaining classes of anomalies, all of which have direct consequences on the fault-tolerance mechanism:

1) *Cross-run nondeterminism* (**Run**), in which a component produces different output stream *contents* in different runs over the same inputs. Systems that exhibit cross-run nondeterminism do not support replay-based fault-tolerance. For obvious reasons, nondeterminism across runs makes such systems difficult to test and debug.
2) *Cross-instance nondeterminism* (**Inst**), in which replicated instances of the same components produce different output contents in the *same* execution over the same inputs. Cross-instance nondeterminism can lead to inconsistencies across queries.
3) *Replica divergence* (**Diverge**), 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.

### *B. Monotonicity, confluence and convergence*

Distributed programs that produce the same outcome for all message delivery orders exhibit none of the anomalies listed in Section III-A, regardless of the choice of fault-tolerance or delivery mechanisms. In recent work, we proposed the *CALM theorem*, which observes that programs expressed in *monotonic* logic produce deterministic results despite nondeterminism in delivery orders [6], [18], [19]. Intuitively, monotonic programs compute a continually growing result, never retracting an earlier output given new inputs. Hence replicas running monotonic code always eventually agree, and replaying monotonic code produces the same result in every run.

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

Confluence is similar to the notion of replica *convergence* common in distributed systems. A system is convergent or "eventually consistent" if, when all messages have been delivered, all replicas agree on the set of stored values [29]. Convergent components never exhibit replica divergence. Convergence is a local guarantee about component *state*; by contrast, confluence provides guarantees about component *outputs*, which (because they become the inputs to downstream components) compose into global guarantees about dataflows.

Confluence implies convergence but the converse does not hold. Convergent replicated components are guaranteed to eventually reach the same state, but this final state may not be uniquely determined by component inputs. As Figure 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 read-only outputs of a convergent component (e.g., GETs posed to a key/value store) flow into a replicated stateful component (e.g., a replicated cache). If the caches record different stream contents, the result is replica divergence.

| *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).

### C. Coordination Strategies

Confluent components produce deterministic outputs and convergent replicated state. How can we achieve these properties for components that are not confluent? We assume that components are deterministic, so we can prevent inconsistent outputs within or across program runs simply by removing the nondeterminism from component input orderings. Two extreme approaches include (a) establishing a single total order in which all instances of a given component receive messages (a *sequencing* strategy) and (b) disallowing components from producing outputs until all of their inputs have arrived (a *sealing* strategy). The former—which enforces a total order of inputs—resembles state machine replication from the distributed systems literature [30], 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 [31] in the database literature, as well as barrier synchronization mechanisms used in systems like MapReduce [32].

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

Fortunately, both coordination strategies have a dynamic variant that allows systems to make incremental progress over time. To prevent replica divergence, it is sufficient to use a dynamic ordering service (e.g., Paxos) that decides a global order of messages *within a particular run*. As Figure 5 shows, a nondeterministic choice of message ordering can prevent cross-instance nondeterminism but not cross-run nondeterminism since the choice is dependent on arrival orders at the coordination service. Similarly, strategies based on sealing inputs can be applied to infinite streams as long as the streams can be partitioned into finite partitions that exhibit temporal locality, like windows with "slack" [33]. Sealing strategies—applicable when input stream partitioning is *compatible* with component semantics—can rule out all nondeterminism anomalies.[1] Note that sealing is significantly less constrained than ordering: it enforces an output barrier per partition, but allows asynchrony both in the arrival of a batch's inputs and in interleaving across batches.

[1]That is, **M3** and **P3** together are semantically equivalent to **P1** in Figure 5. The notion of compatibility is defined in Section V.

### D. Example Queries

The ad reporting system presented in Section I-B 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 [34].

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

In practice, streaming query systems often address the problem of blocking operators via *windowing*, which constrains blocking queries to operate over bounded inputs [33], [35], [36]. 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

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

Fig. 7: The **C.O.W.R.** 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.

indicating the termination of a campaign, the nonmonotonic query *CAMPAIGN* can produce deterministic outputs.

## IV. 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 by presenting a *grey box* model in which programmers provide simple annotations about the semantic properties of components. In Section V, we show how BLAZES can use these annotations to automatically derive the consistency properties of entire dataflow graphs.

### A. Annotations and Labels

In this section, we describe a language of *annotations* and *labels* that enriches the "black box" model (Section II) 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.

*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.

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 its inputs do not modify the component's state. $CW$ denotes a path that is confluent and stateful.

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

Consider a reporting server component implementing the query *WINDOW*. When it receives a request referencing a particular advertisement and window, it returns a response if the advertisement has fewer than 1000 clicks *within that window*. An appropriate label for the path from request inputs to outputs as $OR_{id,window}$—a stateless order-sensitive path operating over partitions with composite key *id,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. 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 *WINDOW* is compatible with click streams partitioned (and emitting appropriate punctuations) on *id* or *window*.

*2) Stream Annotations:* 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**$_{session}$, to indicate that clients will send messages indicating that sessions are complete. To ensure progress, there must be a punctuation for every session identifier that appears in any message.

Programmers can use the **Rep** annotation to indicate that a stream is *replicated*. A replicated stream connects a producer component instance (or instances) to more than one consumer component instance, and 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 divergence. **Rep** is an optional Boolean flag that may be combined with other annotations and labels.

## V. Coordination Analysis and Synthesis

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

### A. Analysis

To derive labels for the output streams in a dataflow graph, BLAZES starts by enumerating all paths between pairs of sources and sinks. To rule out infinite paths, it reduces each cycle in the graph to a single node with a collapsed label by selecting the label of highest severity among the cycle

[2] This rules out races by ensuring (without enforcing an ordering on message delivery) that the query comes *after* all relevant click records.

members.[3]

For each component whose input streams are labeled (beginning with the components with unconnected inputs), BLAZES performs an inference procedure to derive a stream label for each of the component's output streams. These output stream labels becomes input stream labels for the next components in the dataflow, and so on until all output streams are labeled. The inference procedure is implemented as a term rewriting system; for space reasons, we describe its details in the technical report [37].

For non-confluent components with sealed input streams, the inference procedure must test whether the component preserves the independence of the sealed partitions—if it does, BLAZES can ensure deterministic outcomes by delaying processing of partitions until when their complete contents are known. For example, given the queries in Figure 6, 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.

A stream sealed on key $key$ is compatible with a component with annotation $OR_{gate}$ or $OW_{gate}$ 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 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.

### B. Coordination Selection

BLAZES will automatically repair dataflows that are not confluent or convergent by constraining how messages are delivered to certain 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 to those components.

*1) Sealing Strategies:* If the programmer has provided a seal annotation $\textbf{Seal}_{key}$ that is compatible with the (non-confluent) component annotation, we may use a synchronization strategy that avoids global coordination. The intuition is that if the component never combines inputs from different (punctuated) partitions, then the order in which it learns about the partitions, their contents and their corresponding seals has no effect on its outputs. Consider a component representing a reporting server executing the query *WINDOW* from Section I-B. 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 a given 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 one-way coordination, limited to the "stakeholders" contributing to or consuming individual stream 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.

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

## VI. Case Studies

In this section, we apply BLAZES to the examples introduced in Section I-B. 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 Section VIII we will show concrete performance benefits of the BLAZES coordination choices as compared to a conservative use of a coordination service such as 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. The interested reader should refer to the technical report [37] for details on the derivation of each output stream label using the BLAZES analyzer.

### A. 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

[3]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 $q$.

unnecessary to ensure deterministic replay, and hence consistent outcomes.

*1) Component annotations:* To annotate the three components of the Storm word count query, we provide the following file to BLAZES:

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

`Splitter` is a stateless, confluent component: we give it the annotation $CR$. We annotate `Count` as $OW_{word,batch}$—it is stateful (accumulating counts over time) and order-sensitive, but potentially sealable on word or batch (or both). Lastly, `Commit` is also stateful (the backing store to which it stores the final counts is persistent), but since it is append-only and does not record the order of appends, we annotate it $CW$.

*2) Analysis:* In the absence of any seal annotations, BLAZES derives an output label of **Run** for the wordcount dataflow. Without coordination, nondeterministic input orders may produce nondeterministic output contents due to the order-sensitive nature of the `Count` component. 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 recognizes the compatibility between the stream punctuations and the `Count` component, which operates over grouping sets of *word, batch*. 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 under all interleavings.

### B. Ad-reporting system

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

*1) Component annotations:* Below is the BLAZES annotation file for the ad serving network:

```
Cache:
  annotation:
    - { from: request, to: response, label: CR }
    - { from: response, to: response, label: CW }
    - { from: request, to: request, label: CR }
Report:
  Rep: true
  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] }
```

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

All that remains is to annotate the read-only path through the reporting component corresponding to the various continuous queries listed in Figure 6. `Report` is a replicated component, so we supply the **Rep** annotation for all instances. We annotate the query path corresponding to *THRESH*—which is confluent because it never emits a record until the ad impressions reach the given threshold—$CR$. We annotate queries *POOR* and *CAMPAIGN* $OR_{id}$ and $OR_{id,campaign}$, respectively. These queries 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*, making it compatible with an input stream sealed on *window*.

*2) Analysis:* Having annotated all of the instances of the reporting server component for different queries, we may now consider how BLAZES automatically derives output stream labels for the global dataflow. If we supply *THRESH*, BLAZES derives a final label of **Async** for the output path from cache to sink. 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 $CW$.

Given query *POOR* with no input stream annotations, BLAZES derives a label of **Diverge**. The poor performers query is not confluent: it produces nondeterministic output contents. 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 to the reporting server.

If, however, the input stream is sealed on *campaign*, BLAZES recognizes the compatibility between the stream partitioning and the component path annotation $OR_{id,campaign}$, synthesizes a protocol that allows the partition to be processed when it has stopped changing, and gives the dataflow the label **Async**. Implementing this sealing strategy does not require global coordination, but merely some synchronization between stream producers and consumers.

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

## VII. 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 IV. As we saw in Section VI-A1, this process is often quite natural; unfortunately, as we learned in Section VI-B1, 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 transparent "white box" system, in which unadorned programs are submitted, analyzed and—if necessary to ensure consistent outcomes—automatically rewritten. By applying techniques from database theory and logic programming, BLAZES and Bloom allow programmers to shift their focus from individual component behaviors to program outcomes—a significant step towards truly declarative programming for distributed systems.

### A. 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.

### B. 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" [38] the injective functional dependencies described in Section V transitively across compositions.

*1) Confluence and state:* As we described in Section III-B, 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. Any component free of such operations is provably order-insensitive. Similarly, Bloom's type system distinguishes syntactically between transient event streams and stored tables. A simple flow analysis automatically determines if a component accumulates state over time. Together, these analyses are sufficient to determine annotations (except for the subscripts, which we describe next) for every Bloom statement in a given module.

*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 detecting injective functional dependencies.

Recall that in Section IV-A1 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*), 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 detect injective functional dependencies (in a sound but incomplete way), we 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 $S \equiv \pi_a \pi_{ab} \pi_{abc} R$, $S.a$ is injectively functionally determined by $R.a$.

## VIII. EVALUATION

In Section III, 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.

### A. Storm wordcount

To evaluate the potential savings of avoiding unnecessary synchronization in Storm, we implemented two versions of the streaming wordcount query described in Section I-B. 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 coordinate 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 "nontransactional topology." We optimized the batch size and cluster configurations of both implementations to maximize throughput.

[4] To see this, note that 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.

[5] Storm uses Zookeeper for coordination.

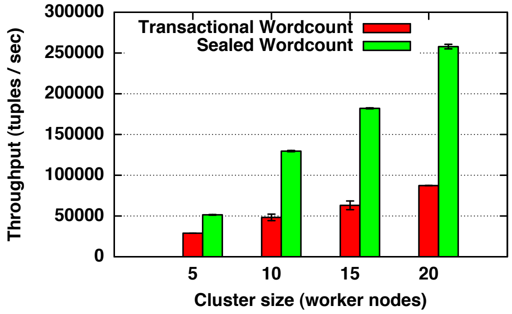


Fig. 8: The effect of coordination on throughput for a Storm topology computing a streaming wordcount.

We used a single dedicated node (as the documentation recommends) for the Storm master 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 8 plots the throughput of the coordinated and uncoordinated implementations of the wordcount dataflow as a function of the cluster size. The overhead of conservatively deploying a transactional topology is considerable. The uncoordinated dataflow has a peak throughput roughly 1.8 times that of its coordinated counterpart in a 5-node deployment. As we scale up the cluster to 20 nodes, the difference in throughput grows to $3\times$.

### B. 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 ad tracking network introduced in Section I-B. 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.

Ad servers generate a workload of 1000 log entries per server, dispatching 50 click log messages in batch and sleeping periodically. During the workload, we pose a number of requests to the reporting servers, all of which implement the continuous query *CAMPAIGN*.

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

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

*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 9 and 10 plots the records processed over time for this strategy.

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

*3) Sealing Strategies:* 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 the sealed stream and the aggregate query in *CAMPAIGN* (a "group-by" on *id, campaign*), BLAZES synthesized a seal-based coordination strategy.

Using the seal-based strategy, reporting servers do not need to wait until events are globally ordered; instead, they are processed as soon as a reporting server can determine that they belong to a sealed partition. The reporting servers use Zookeeper only to determine the set of ad servers responsible for each campaign—that is, one call to Zookeeper per campaign. When a reporting server has received seal messages from all producers for a given campaign, it emits the partition for processing.

In Figures 9 and 10 we evaluate the sealing strategy for two alternative partitionings of click records: in "Independent seal" each campaign is mastered at exactly one adserver, while in "Seal," all ad servers produce click records for all campaigns. Note that both seal-based runs closely track the performance of the uncoordinated run; doubling the number of ad servers effectively doubles system throughput.

To highlight the differences between the two seal-based runs, Figure 11 plots the 10-server run but omits the ordering strategy. As we would expect, "independent seals" result in 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, conflicting with the coordination strategy. We revisit the notion of "coordination locality" in Section X.

## IX. Related Work

Our approach to automatically coordinating distributed services draws inspiration from the literature on both distributed

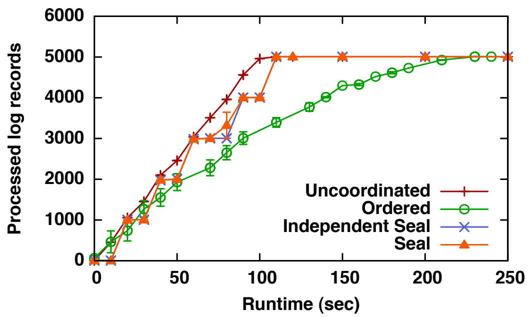


Fig. 9: Log records processed over time, 5 ad servers.

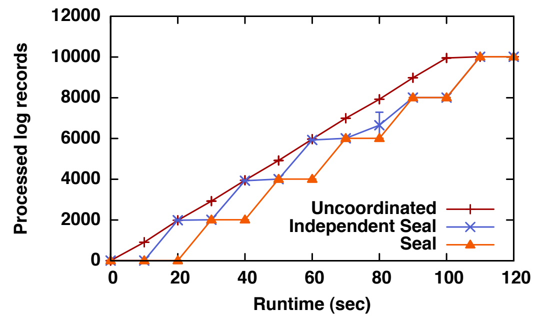


Fig. 11: Seal-based strategies, 10 ad servers.

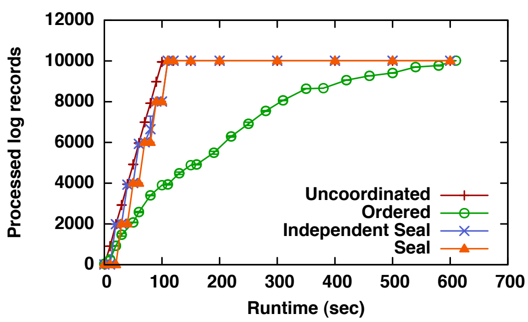


Fig. 10: Log records processed over time, 10 ad servers.

systems and databases. Ensuring consistent replica state by establishing a total order of message delivery is the technique adopted by state machine replication [30]; each component implements a deterministic state machine, and a global coordination service such as atomic broadcast or Multipaxos decides the message order.

Marczak et al. draw a connection between stratified evaluation of conventional logic programming languages and distributed protocols to ensure consistency [39]. They describe a program rewrite that ensures deterministic executions by preventing any node from performing a nonmonotonic operation until that operation's inputs have stopped changing. 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 [40] 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. CRDTs [41] are convergent replicated data structures; CRDTs can be modeled in BLAZES as components whose update API calls are labeled $CW$. LVar data structures [42] ensure determinism for shared-memory parallel programs by restricting modifications and observations of shared state according to a user-specified lattice. Like confluent components, CRDTs and LVars ensure monotone growth of state.

Like reactive distributed systems, streaming databases [33], [35], [36] 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 [43] attempts to efficiently and correctly evaluate a workload with heterogeneous consistency requirements, ensuring replica convergence while taking advantage of cheaper strategies for operations that require only weak orderings. By contrast, BLAZES makes guarantees about composed services, which requires reasoning about the properties of streams as well as component state.

## X. CONCLUSIONS

BLAZES relieves programmers of 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 VIII-B3, 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. Problems emerge if coordination locality is in conflict with spatial locality. For example, clustering ads likely to be served together (the non-independent seal topology in Figure 11) caused campaigns (the seal key) to be distributed across multiple nodes, increasing coordination latency.

Given a dataflow, 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 input orders, 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. More challenging and compelling is the possibility of capturing these design principles into a compiler and automatically rewriting dataflows.

## Acknowledgments

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 FA95500810352), the Natural Sciences and Engineering Research Council of Canada, the National Science Foundation (grants CNS-0722077, IIS-0713661, IIS-0803690, and IIS-0917349), and gifts from EMC, Microsoft Research, and NTT Multimedia Communications Laboratories.

## References

[1] K. Birman, G. Chockler, and R. van Renesse, "Toward a Cloud Computing Research Agenda," *SIGACT News*, 2009.

[2] "Appalachian trail conservancy guide: trail markings," http://www.appalachiantrail.org/hiking/hiking-basics/trail-markings, Accessed: 2013-11-25.

[3] R. Agrawal *et al.*, "The Claremont Report on Database Research," *SIGMOD Record*, vol. 37, pp. 9–19, 2008.

[4] P. Alvaro, T. Condie, N. Conway, J. M. Hellerstein, and R. Sears, "I Do Declare: Consensus in a Logic Language," *ACM SIGOPS Operating Systems Review*, 2010.

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

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

[7] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica, "Declarative networking: language, execution and optimization," in *SIGMOD*, 2006.

[8] P. Helland, "Life beyond distributed transactions: an apostate's opinion," in *CIDR*, 2007.

[9] E. Brewer, "CAP twelve years later: How the "rules" have changed," *IEEE Computer*, vol. 45, pp. 23–29, 2012.

[10] M. Burrows, "The Chubby lock service for loosely-coupled distributed systems," in *OSDI*, 2006.

[11] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free Coordination for Internet-scale Systems," in *USENIX ATC*, 2010.

[12] R. Cattell, "Scalable SQL and NoSQL Data Stores," *SIGMOD Record*, pp. 12–27, 2010.

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

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

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

[16] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed GraphLab: a framework for machine learning and data mining in the cloud," in *VLDB*, 2012.

[17] B. Recht, C. Re, S. J. Wright, and F. Niu, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent," in *NIPS*, 2011.

[18] J. M. Hellerstein, "The Declarative Imperative: Experiences and conjectures in distributed logic," *SIGMOD Record*, 2010.

[19] T. J. Ameloot, F. Neven, and J. Van den Bussche, "Relational Transducers for Declarative Networking," in *PODS*, 2011.

[20] D. Zinn, T. J. Green, and B. Ludäscher, "Win-move is Coordination-free (Sometimes)," in *ICDT*, 2012.

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

[22] "Bloom programming language," http://www.bloom-lang.org.

[23] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed Stream Computing Platform," in *ICDMW*, 2010.

[24] 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.

[25] G. Kahn, "The semantics of a simple language for parallel programming," in *Information Processing*, Aug 1974, pp. 471–475.

[26] 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*, vol. 1, no. 1, pp. 274–288, 2008.

[27] M. Balazinska, J.-H. Hwang, and M. A. Shah, "Fault-Tolerance and High Availability in Data Stream Management Systems," in *Encyclopedia of Database Systems*. Springer US, 2009, pp. 1109–1115.

[28] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras, "Exploiting Punctuation Semantics in Continuous Data Streams," *TKDE*, vol. 15, no. 3, pp. 555–568, 2003.

[29] W. Vogels, "Eventually Consistent," *CACM*, vol. 52, no. 1, pp. 40–44, Jan. 2009.

[30] F. B. Schneider, "Implementing Fault-tolerant Services Using the State Machine Approach: a Tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, Dec. 1990.

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

[32] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *OSDI*, 2004.

[33] 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*, vol. 12, no. 2, pp. 120–139, Aug. 2003.

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

[35] A. Arasu, S. Babu, and J. Widom, "The CQL Continuous Query Language: Semantic Foundations and Query Execution," *VLDB Journal*, vol. 15, no. 2, Jun. 2006.

[36] 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.

[37] P. Alvaro, N. Conway, J. M. Hellerstein, and D. Maier, "Blazes: coordination analysis for distributed programs," http://arxiv.org/abs/1309.3324, 2013, Technical Report.

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

[39] 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.

[40] M. C. Rinard and P. C. Diniz, "Commutativity Analysis: a New Analysis Technique for Parallelizing Compilers," *ACM Trans. Program. Lang. Syst.*, vol. 19, no. 6, pp. 942–991, Nov. 1997.

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

[42] L. Kuper and R. R. Newton, "LVars: Lattice-based Data Structures for Deterministic Parallelism," in *FHPC '13*.

[43] 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.