Bulletin of the Technical Committee on

Data Engineering

December 2015 Vol. 38 No. 4

IEEE Computer Society

Letters

Special Issue on Next-Generation Stream Processing

Conference and Journal Notices

The TC on Data Engineering

Membership in the TC on Data Engineering is open to all current members of the IEEE Computer Society who are interested in database systems. The TCDE web page is http://tab.computer.org/tcde/index.html.

The Data Engineering Bulletin

The Bulletin of the Technical Committee on Data Engineering is published quarterly and is distributed to all TC members. Its scope includes the design, implementation, modelling, theory and application of database systems and their technology.

Letters, conference information, and news should be sent to the Editor-in-Chief. Papers for each issue are solicited by and should be sent to the Associate Editor responsible for the issue.

Opinions expressed in contributions are those of the authors and do not necessarily reflect the positions of the TC on Data Engineering, the IEEE Computer Society, or the authors' organizations.

The Data Engineering Bulletin web site is at http://tab.computer.org/tcde/bull_about.html.

Letter from the Editor-in-Chief

Delayed Publication

This December, 2015 issue of the Bulletin is, as some of you may notice, being published in July of 2016, after the March and June, 2016 issues have been published. Put simply, the issue is late, and the March and June issues were published in their correct time slots. The formatting of the issue, and the surrounding editorial material, e.g. the inside front cover and copyright notice, are set to the December, 2015 timeframe. Indeed, the only mention of this inverted ording of issues is in this paragraph. Things do not always go as planned. However, I am delighted that the current issue is being published, and I have high confidence that you will enjoy reading about next-generation stream processing, the topic of the issue.

The Current Issue

At one point a few years ago, the research community had lost interest in stream processing. The first streaming systems had been built and these early systems demonstrated their feasibility. Commercial interest had been generated, with a number of start-ups and major vendors entering the market. Even using a declarative database-style query language had become an accepted part of the technology landscape. Job done, right? Actually, wrong!

As we have seen with the database field itself, innovation and a changing technological environment can lead to an "encore" of interest in a field. Such is the case with stream processing. The issue title: "Next-Generation Stream Processing" captures that. The issue itself captures a whole lot more about the state of the field. Streaming systems have evolved, sometimes in revolutionary ways. Applications of streaming technology have exploded, both in number and in importance. As much as at any time in the past, the streams area is a hive of activity. New technology is opening new application areas, while new application areas create a pull for new technology.

David Maier has worked with Badrish Chandramouli to assemble this current issue devoted to presenting the diversity of work now in progress in the streaming area. Streaming technology is at the core of much of their recent research. This makes them ideal editors for the current issue. They have brought together papers that not only provide insights into new streaming technology, but also illustrate where technology might be taking us in its enabling of new applications. Streams are here as a permanent part of the technology environment in a way similar to databases. Thanks to both David and Badrish for bringing this issue together on a topic that will, I am convinced, become a fixture of both the research and the application environment of our field.

David Lomet
Microsoft Corporation

Letter from the Special Issue Editors

The precursors of data-stream systems began to show up in the late 1980s and early 1990s in the form of "reactive" extensions to data management systems. With such extensions, there was a reversal of sorts between the roles of data and query. Database requests – in the form of continuous queries, materialized views, event-condition-action rules, subscriptions, and so forth – became persistent entities that responded to newly arriving data.

The initial generation of purpose-built stream systems addressed many issues: appropriate languages, dealing with unbounded input, handling delay and disorder, dealing with high data rates, load balancing and shedding, resiliency, and, to some extent, distribution and parallelism. However, integration with other system components, such as persistent storage and messaging middleware, was often rudimentary or left to the application programmer.

The most recent generation of stream systems have the benefit of a better understanding of application requirements and execution platforms, by virtue of lessons learned through experimentation with earlier systems. Scaling, in cloud, fog, and cluster environments, has been at the forefront of design considerations. Systems need to scale not just in terms of stream rate and number of streams, but also to large numbers of queries. Application tuning, operation, and maintenance have also come to the forefront. Support for tradeoffs among throughput, latency, accuracy, and availability is important for application requirements, such as meeting service-level agreements. Resource management at run time is needed to enable elasticity of applications as well as for managing multi-tenancy both with other stream tasks and other application components. Many stream applications require long-term deployment, possibly on the order of years. Thus, the ability to maintain the underlying stream systems as well as evolve applications that run on them is critical. State management is also a concern, both within stream operators and in interactions with other state managers, such as transactional storage. There has also been a focus on broadening the use of stream-processing systems, but through programming models for non-specialists and by supporting more complex analyses over streams, such as machine-learning techniques.

This issue is devoted to this next generation of stream-processing, looking at particular systems, specific optimization and evaluation techniques, and programming models.

The first three papers discuss frameworks that support composing reliable and distributed stream (and batch) processing networks out of individual operators, but are somewhat agnostic about what the particular operators are. Samza (Kleppman, et al.) is a stream-processing framework developed initially at LinkedIn that supports stream operators loosely coupled using the Kafka message broker. The use of Kafka reduces dependencies between stream stages, and provides replicated logs that support multiple consumers running at different rates. The next paper (Fu, et al.) introduces Heron, whose API is compatible with Twitters early streaming platform, Storm. Heron features support sustained deployment and maintenance, such as resource reservations and task isolation. The paper discusses alternative back-pressure mechanisms, and how Heron supports at-least-once and at-most-once messaging semantics. Apache Flink (Carbone, et al.) is a framework that supports a general pipelined dataflow architecture that handles both live stream and historical batch data (and combinations) for simple queries as well as complex iterative scripts as found in machine-learning. The paper discusses mechanisms for trading latency with throughput; the use of in-stream control events to help checkpointing, track progress and coordinate iterations; and low-interference fault-tolerance taking consistent snapshots across operators without pausing execution.

The next three papers deal with complete systems that include specific query languages. In Connected Streaming Analytics (CSA) from Cisco (Shen, et al.), stream-processing components can be embedded in network elements such as routers and switches to support Internet-of-Things applications. Given this execution environment, it is important that stream queries not interfere with high-priority network tasks. CSA uses a container mechanism to constrain resources and promote portability. The language is SQL with window extensions. CSA supports different kinds of window joins: best-effort join combines data immediately on receipt, whereas coordinated join matches items based on application time, which may require buffering. Trill (Chandramouli, et

al.) shares goals with Flink in seeking a single engine that can work for online, incremental and offline processing, and supports latency-throughput tradeoffs as appropriate for different contexts. It takes a library approach that allows mutual embedding with applications written in high-level languages. Trill queries are written in a LINQ-based language that supports tempo-relational operations, along with timestamp manipulation capabilities. For performance, it uses a columnar in-memory representation of data batches. The subsequent paper looks at language runtime support for the IBM Stream Processing Language (SPL) (Schneider, et al.). The SPL runtime provides certain execution guarantees, such as isolation of operator state and in-order delivery, and satisfies performance goals such as long-term query execution without degradation and efficient parallel execution. Performance optimizations include both "fusion" (combining operators into a single Processing Element) and "fission" (replicating a portion of the query graph).

The next three papers consider stream-processing optimizations and guarantees. While several of the systems in the foregoing papers provide a means to make performance tradeoffs, in practice it can be difficult for a user to determine the best way to adjust the control knobs. The FUGU stream-processing system (Heinze, et al.) employs strategies that automate the adjustment of these parameters, based on on-line profiling of query execution and user-provided latency specifications. The paper from Worchester Polytechnic Institute (Rundensteiner, et al.) looks at several methods to improve performance of pattern-matching queries, using a variety of sharing strategies. Examples are Event-Sequence Pattern Sharing, which determines temporal correlations between sub-patterns in order to decide whether sharing is beneficial, and Shared Event-Pattern Aggregation, which looks for shared aggregation opportunities at the sub-pattern level. Several early stream systems had the ability to access stored data in some form, for example, to augment stream events with information from a look-up table. However, these systems gave limited consistency guarantees, either between the stream and the stored data, or between shared access to stored data across stream operators. The S-Store system (Tatbul, et al.) develops a stream-processing model that provides several correctness guarantees, such as traditional ACID semantics, order-of-execution conditions and exactly-once semantics.

The last two papers are oriented towards application development. Most stream systems require queries to be written in a special request language or a general-purpose programming language, either of which is a hurdle for non-CS experts. The Event Model (TEM) (Etzion, et al.) allows a user to specify an event-driven application by concentrating on application logic, expressed in diagrams and associated condition tables. The TEM environment can fill in low-level details and manage the conversion to a particular stream-processing system. "Live" analytics are a major driver of next-generation stream systems. Our final paper looks at mining for events in a text stream (Grossniklaus, et al.). It adopts a tool-kit approach that allows easy implementation of many of the published approaches in this domain. In addition, it describes an evaluation platform for comparing alternative event-detection techniques.

David Maier, Badrish Chandramouli
Portland State University (Maier), Microsoft Corporation (Chandramouli)

Kafka, Samza and the Unix Philosophy of Distributed Data

Martin Kleppmann
University of Cambridge
Computer Laboratory

Jay Kreps
Confluent, Inc.

Abstract

Apache Kafka is a scalable message broker, and Apache Samza is a stream processing framework built upon Kafka. They are widely used as infrastructure for implementing personalized online services and real-time predictive analytics. Besides providing high throughput and low latency, Kafka and Samza are designed with operational robustness and long-term maintenance of applications in mind. In this paper we explain the reasoning behind the design of Kafka and Samza, which allow complex applications to be built by composing a small number of simple primitives – replicated logs and stream operators. We draw parallels between the design of Kafka and Samza, batch processing pipelines, database architecture, and the design philosophy of Unix.

1 Introduction

In recent years, online services have become increasingly personalized. For example, in a service such as LinkedIn there are many activity-based feedback loops, automatically adapting the site to make it more relevant to individual users: recommendation systems such as "people you may know" or "jobs you might be interested in" [30], collaborative filtering [33] or ranking of search results [23, 26] are personalized based on analyses of user behavior (e.g. click-through rates of links) and user-profile information. Other feedback loops include abuse prevention (e.g. blocking spammers, fraudsters and other users who violate the terms of service), A/B tests and user-facing analytics (e.g. "who viewed your profile").

Such personalization makes a service better for users, as they are likely to find what they need faster than if the service presented them with static information. However, personalization has also opened new challenges: a huge amount of data about user activity needs to be collected, aggregated and analyzed [8]. Timeliness is important: after the service learns a new fact, the personalized recommendations and rankings should be swiftly updated to reflect the new fact, otherwise their utility is diminished.

In this paper we describe Kafka and Samza, two related projects that were originally developed at LinkedIn as infrastructure for solving these data collection and processing problems. The projects are now open source, and maintained within the Apache Software Foundation as Apache Kafka[1] and Apache Samza[2], respectively.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

[1] `http://kafka.apache.org/`
[2] `http://samza.apache.org/`

1.1 Implementing Large-Scale Personalized Services

In a large-scale service with many features, the maintainability and the operational robustness of an implementation are of paramount importance. The system should have the following properties:

System scalability: Supporting an online service with hundreds of millions of registered users, handling millions of requests per second.

Organizational scalability: Allowing hundreds or even thousands of software engineers to work on the system without excessive coordination overhead.

Operational robustness: If one part of the system is slow or unavailable, the rest of the system should continue working normally as much as possible.

Large-scale personalized services have been successfully implemented as batch jobs [30], for example using MapReduce [6]. Performing a recommendation system's computations in offline batch jobs decouples them from the online systems that serve user requests, making them easier to maintain and less operationally sensitive.

The main downside of batch jobs is that they introduce a delay between the time the data is collected and the time its effects are visible. The length of the delay depends on the frequency with which the job is run, but it is often on the order of hours or days.

Even though MapReduce is a lowest-common-denominator programming model, and has fairly poor performance compared to specialized massively parallel database engines [2], it has been a remarkably successful tool for implementing recommendation systems [30]. Systems such as Spark [34] overcome some of the performance problems of MapReduce, although they remain batch-oriented.

1.2 Batch Workflows

A recommendation and personalization system can be built as a *workflow*, a directed graph of MapReduce jobs [30]. Each job reads one or more input datasets (typically directories on the Hadoop Distributed Filesystem, HDFS), and produces one or more output datasets (in other directories). A job treats its input as immutable and completely replaces its output. Jobs are chained by directory name: the same name is configured as output directory for the first job and input directory for the second job.

This method of chaining jobs by directory name is simple, and is expensive in terms of I/O, but it provides several important benefits:

Multi-consumer. Several different jobs can read the same input directory without affecting each other. Adding a slow or unreliable consumer affects neither the producer of the dataset, nor other consumers.

Visibility. Every job's input and output can be inspected by ad-hoc debugging jobs for tracking down the cause of an error. Inspection of inputs and outputs is also valuable for audit and capacity planning purposes, and monitoring whether jobs are providing the required level of service.

Team interface. A job operated by one team of people can produce a dataset, and jobs operated by other teams can consume the dataset. The directory name thus acts as interface between the teams, and it can be reinforced with a contract (e.g. prescribing the data format, schema, field semantics, partitioning scheme, and frequency of updates). This arrangement helps organizational scalability.

Loose coupling. Different jobs can be written in different programming languages, using different libraries, but they can still communicate as long as they can read and write the same file format for inputs and outputs. A job does not need to know which jobs produce its inputs and consume its outputs. Different jobs can be run on different schedules, at different priorities, by different users.

Data provenance. With explicitly named inputs and outputs for each job, the flow of data can be tracked through the system. A producer can identify the consumers of its dataset (e.g. when making forward-incompatible changes), and a consumer can identify its transitive data sources (e.g. in order to ensure regulatory compliance).

Failure recovery. If the 46th job in a chain of 50 jobs failed due to a bug in the code, it can be fixed and restarted at the 46th job. There is no need to re-run the entire workflow.

Friendly to experimentation. Most jobs modify only to their designated output directories, and have no other externally visible side-effects such as writing to external databases. Therefore, a new version of a job can easily be run with a temporary output directory for testing purposes, without affecting the rest of the system.

1.3 From Batch to Streaming

When moving from a high-latency batch system to a low-latency streaming system, we wish to preserve the attractive properties listed in Section 1.2.

By analogy, consider how Unix tools are composed into complex programs using shell scripts [21]. A workflow of batch jobs is comparable to a shell script in which there is no pipe operator, so each program must read its input from a file on disk, and write its output to a different (temporary) file on disk. In this scenario, one program must finish writing its output file before another program can start reading that file.

To move from a batch workflow to a streaming data pipeline, the temporary files would need to be replaced with something more like Unix pipes, which support incrementally passing one program's output to another program's input without fully materializing the intermediate result [1]. However, Unix pipes do not have all the properties we want: they connect exactly one output to exactly one input (not multi-consumer), and they cannot be repaired if one of the processes crashes and restarts (no failure recovery).

Kafka and Samza provide infrastructure for low-latency *distributed stream processing* in a style that resembles a chain of Unix tools connected by pipes, while also preserving the aforementioned benefits of chained batch jobs. In the following sections we will discuss the design decisions that this approach entails.

1.4 Relationship of Kafka and Samza

Kafka and Samza are two separate projects with a symbiotic relationship. Kafka provides a message broker service, and Samza provides a framework for processing messages. A Samza job uses the Kafka client library to consume input streams from the Kafka message broker, and to produce output streams back to Kafka. Although either system can be used without the other, they work best together. We introduce Kafka in more detail in Section 2, and Samza in Section 3.

At the time of writing, there is an effort underway to add a feature called *Kafka Streams* to the Kafka client library [31]. This feature provides a stream processing capability similar to Samza, but it differs in that Kafka Streams does not prescribe a deployment mechanism, whereas Samza currently relies on Hadoop YARN. Most other high-level architecture choices are similar in Samza and Kafka Streams; for purposes of this paper, they can be regarded as equivalent.

2 Apache Kafka

Kafka has been described in detail in prior work [8, 16, 19, 32]. In this section we present a brief high-level overview of the principles behind Kafka's design.

Kafka provides a publish-subscribe messaging service, as illustrated in Figure 1. Producer (publisher) clients write messages to a named *topic*, and consumer (subscriber) clients read messages in a topic.

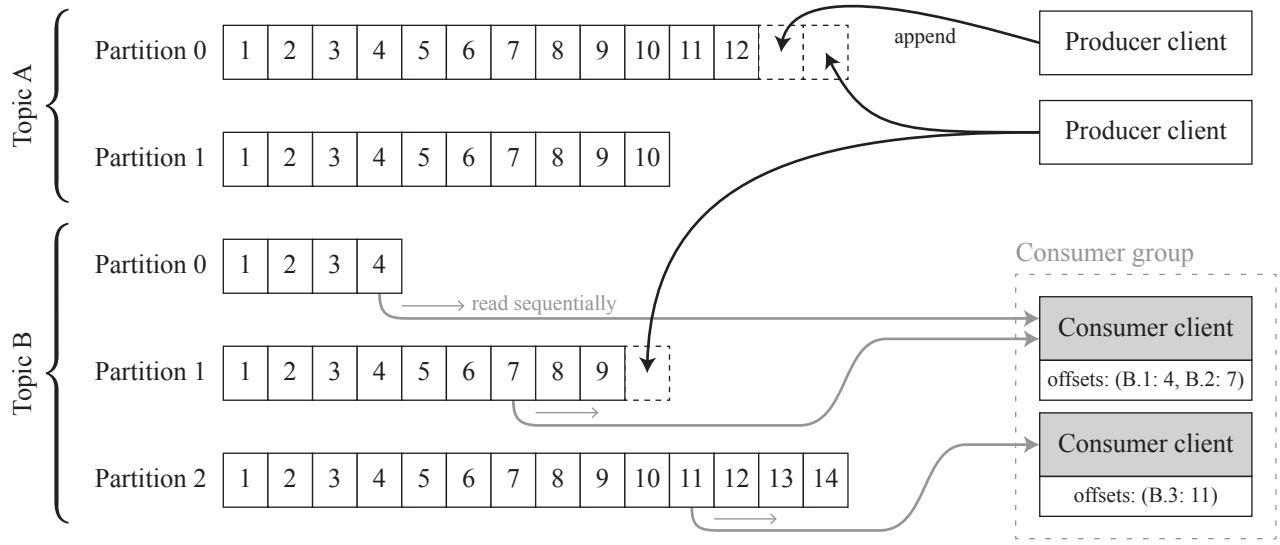


Figure 1: A Kafka *topic* is divided into *partitions*, and each partition is a totally ordered sequence of *messages*.

A topic is divided into *partitions*, and messages within a partition are totally ordered. There is no ordering guarantee across different partitions. The purpose of partitioning is to provide horizontal scalability: different partitions can reside on different machines, and no coordination across partitions is required. The assignment of messages to partitions may be random, or it may deterministic based on a key, as described in Section 3.2.

Broker nodes (Kafka servers) store all messages on disk. Each partition is physically stored as a series of segment files that are written in an append-only manner. A Kafka partition is also known as a *log*, since it resembles a database's transaction commit log [12]: whenever a new message is published to a topic, it is appended to the end of the log. The Kafka broker assigns an *offset* to the message, which is a per-partition monotonically increasing sequence number.

A message in Kafka consists of a key and a value, which are untyped variable-length byte strings. For richer datatypes, any encoding can be used. A common choice is Apache Avro,[3] a binary encoding that uses explicit schemas to describe the structure of messages in a topic, providing a statically typed (but evolvable) interface between producers and consumers [10, 15].

A Kafka consumer client reads all messages in a topic-partition sequentially. For each partition, the client tracks the offset up to which it has seen messages, and it polls the brokers to await the arrival of messages with a greater offset (akin to the Unix tool `tail -f`, which watches a file for appended data). The offset is periodically checkpointed to stable storage; if a consumer client crashes and restarts, it resumes reading from its most recently checkpointed offset.

Each partition is replicated across multiple Kafka broker nodes, so that the system can tolerate the failure of nodes without unavailability or data loss. One of a partition's replicas is chosen as *leader*, and the leader handles all reads and writes of messages in that partition. Writes are serialized by the leader and synchronously replicated to a configurable number of replicas. On leader failure, one of the in-sync replicas is chosen as the new leader.

2.1 Performance and Scalability

Kafka can write millions of messages per second on modest commodity hardware [14], and the deployment at LinkedIn handles over 1 trillion unique messages per day [20]. Message length is typically low hundreds of

[3] `http://avro.apache.org/`

bytes, although smaller or larger messages are also supported.

In many deployments, Kafka is configured to retain messages for a week or longer, limited only by available disk space. Segments of the log are discarded when they are older than a configurable threshold. Alternatively, Kafka supports a *log compaction* mode, in which the latest message with a given key is retained indefinitely, but earlier messages with the same key are garbage-collected. Similar ideas are found in log-structured filesystems [25] and database storage engines [18].

When multiple producers write to the same topic-partition, their messages are interleaved, so there is no inherent limit to the number of producers. The throughput of a single topic-partition is limited by the computing resources of a single broker node – the bottleneck is usually either its NIC bandwidth or the sequential write throughput of the broker's disks. Higher throughput can be achieved by creating more partitions and assigning them to different broker nodes. As there is no coordination between partitions, Kafka scales linearly.

It is common to configure a Kafka cluster with approximately 100 topic-partitions per broker node [22]. When adding nodes to a Kafka cluster, some partitions can be reassigned to the new nodes, without changing the number of partitions in a topic. This rebalancing technique allows the cluster's computing resources to be increased or decreased without affecting partitioning semantics.

On the consumer side, the work of consuming a topic can be shared between a group of consumer clients (illustrated in Figure 1). One consumer client can read several topic-partitions, but any one topic-partition must be read sequentially by a consumer process – it is not possible to consume only a subset of messages in a partition. Thus, the maximum number of processes in a consumer group equals the number of partitions of the topic being consumed.

Different consumer groups maintain their offsets independently, so they can each read the messages at their own pace. Thus, like multiple batch jobs reading the same input directory, multiple groups of consumers can independently read the same Kafka topic without affecting each other.

3 Apache Samza

Samza is a framework that helps application developers write code to consume streams, process messages, and produce derived output streams. In essence, a Samza job consists of a Kafka consumer, an event loop that calls application code to process incoming messages, and a Kafka producer that sends output messages back to Kafka. In addition, the framework provides packaging, cluster deployment (using Hadoop YARN), automatically restarting failed processes, state management (Section 3.1), metrics and monitoring.

For processing messages, Samza provides a Java interface `StreamTask` that is implemented by application code. Figure 2 shows how to implement a streaming word counter with Samza: the first operator splits every input string into words, and the second operator counts how many times each word has been seen.

For a Samza job with one input topic, the framework instantiates one `StreamTask` for each partition of the input topic. Each task instance independently consumes one partition, no matter whether the instances are running in the same process, or distributed across multiple machines. As processing is always logically partitioned by input partition, even if several partitions are physically processed on the same node, a job's allocated computing resources can be scaled up or down without affecting partitioning semantics.

The framework calls the `process()` method for each input message, and the application code may emit any number of output messages as a result. Output messages can be sent to any partition, which allows re-partitioning data between jobs. For example, Figure 3 illustrates the use of partitions in the word-count example: by using the word as message key, the `SplitWords` task ensures that all occurrences of the same word are routed to the same partition of the `words` topic (analogous to the shuffle phase of MapReduce [6]).

Unlike many other stream-processing frameworks, Samza does not implement its own network protocol for transporting messages from one operator to another. Instead, a job usually uses one or more named Kafka topics as input, and other named Kafka topics as output. We discuss the implications of this design in Section 4.

```
class SplitWords implements StreamTask {

  static final SystemStream WORD_STREAM =
    new SystemStream("kafka", "words");

  public void process(
        IncomingMessageEnvelope in,
        MessageCollector out,
        TaskCoordinator _) {

    String str = (String) in.getMessage();

    for (String word : str.split(" ")) {
      out.send(
        new OutgoingMessageEnvelope(
          WORD_STREAM, word, 1));
    }
  }
}
```

```
class CountWords implements StreamTask,
                            InitableTask {

  private KeyValueStore<String, Integer> store;

  public void init(Config config,
                   TaskContext context) {
    store = (KeyValueStore<String, Integer>)
      context.getStore("word-counts");
  }

  public void process(
        IncomingMessageEnvelope in,
        MessageCollector out,
        TaskCoordinator _) {

    String word = (String) in.getKey();
    Integer inc = (Integer) in.getMessage();

    Integer count = store.get(word);
    if (count == null) count = 0;
    store.put(word, count + inc);
  }
}
```

Figure 2: The two operators of a streaming word-frequency counter using Samza's *StreamTask* API.

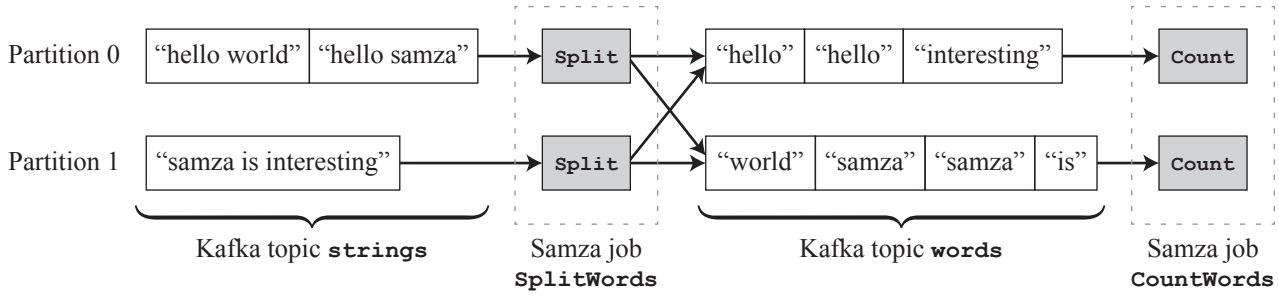


Figure 3: An instance of a Samza task consumes input from one partition, but can send output to any partition.

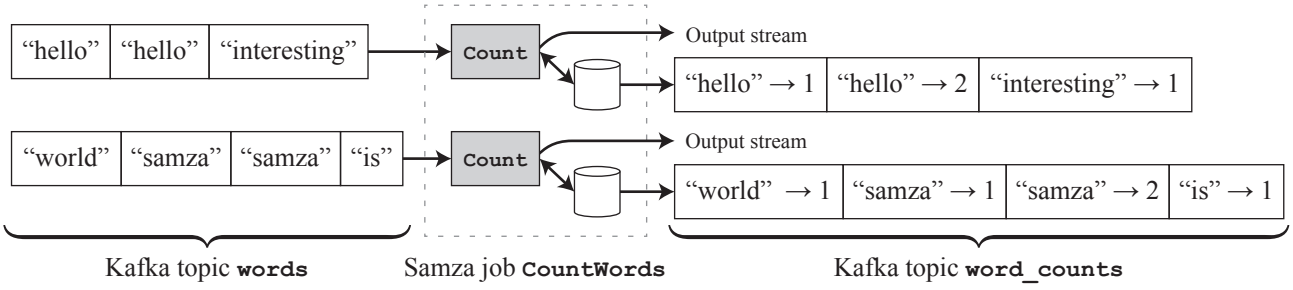


Figure 4: A task's local state is made durable by emitting a changelog to Kafka.

3.1 State Management

Many stream-processing jobs need to maintain state, e.g. in order to perform joins (Section 3.2) or aggregations (such as the counters in `CountWords`, Figure 2). Any transient state can simply be maintained in instance variables of the StreamTask; since messages of a partition are processed sequentially on a single thread, these data structures need not be thread-safe. However, any state that must survive the crash of a stream processor must be written to durable storage.

Samza implements durable state through the `KeyValueStore` abstraction, exemplified in Figure 2. Each StreamTask instance has a separate store that it can read and write as required. Samza uses the RocksDB[4] embedded key-value store, which provides low-latency, high-throughput access to data on local disk. To make the embedded store durable in the face of disk and node failures, every write to the store is also sent to a dedicated topic-partition in Kafka, as illustrated in Figure 4.

This changelog topic acts as a durable replication log for the store: when recovering after a failure, a task can rebuild its store contents by replaying its partition of the changelog from the beginning. Kafka's log compaction mode (see Section 2.1) prevents unbounded growth of the changelog topic: if the same key is repeatedly overwritten (as with a counter), Kafka eventually garbage-collects overwritten values, and retains the most recent value for any given key indefinitely. Rebuilding a store from the log is only necessary if the RocksDB database is lost or corrupted.

Writing the changelog to Kafka is not merely an efficient way of achieving durability, it can also be a useful feature for applications: other stream processing jobs can consume the changelog topic like any other stream, and use it to perform further computations. For example, the `word_counts` topic of Figure 4 could be consumed by another job to determine trending keywords (in this case, the changelog stream is also the `CountWords` operator's output – no separate output topic is required).

3.2 Stream Joins

One characteristic form of stateful processing is a join of two or more input streams, most commonly an equi-join on a key (e.g. user ID). One type of join is a *window join*, in which messages from input streams A and B are matched if they have the same key, and occur within some time interval Δt of one another. Alternatively, a stream may be joined against *tabular data*: for example, user clickstream events could be joined with user profile data, producing a stream of clickstream events with embedded information about the user.

Stream-table joins can be implemented by querying an external database within a StreamTask, but the network round-trip time for database queries soon becomes a bottleneck, and this approach can easily overload the external database [13]. A better option is to make the table data available in the form of a log-compacted stream. Processing tasks can consume this stream to build an in-process replica of a database table partition, using the same approach as the recovery of durable local state (Section 3.1), and then query it with low latency.

For example, in the case of a database of user profiles, the log-compacted stream would contain a snapshot of all user profiles as of some point in time, and an update message every time a user subsequently changes their profile information. Such a stream can be extracted from an existing database using change data capture [5, 32].

When joining partitioned streams, Samza expects that all input streams are partitioned in the same way, with the same number of partitions n, and deterministic assignment of messages to partitions based on the same join key. The Samza job then *co-partitions* its input streams: for any partition k (with $0 \leq k < n$), messages from partition k of input stream A and from partition k of input stream B are delivered to the same StreamTask instance. The task can then use local state to maintain the data that is required to perform the join.

Multi-way joins on several different keys may require different partitioning for each join. Such joins can be implemented with a multi-stage pipeline, where the output of each job partitions messages according to the next stage's join key. The same approach is used in MapReduce workflows.

[4] `http://rocksdb.org/`

4 Discussion

In Sections 2 and 3 we outlined the architecture of Kafka and Samza. We now examine the design decisions behind that architecture in the light of our goals discussed in Section 1, namely creating large-scale personalized services in a way that is scalable, maintainable and operationally robust.

4.1 Use of Replicated Logs

Stream processing with Samza relies heavily on fault-tolerant, partitioned logs as implemented by Kafka. Kafka topics are used for input, output, messaging between operators, durability of local state, replicating database tables, checkpointing consumer offsets, collecting metrics, and disseminating configuration information.

An append-only log with optional compaction is one of the simplest data structures that is useful in practice [12]. Kafka focuses on implementing logs in a fault-tolerant and scalable way. Since the only access methods supported by a log are an appending write and a sequential read from a given offset, Kafka avoids the complexity of implementing random-access indexes. By doing less work, Kafka is able to provide much better performance than systems with richer access methods [14, 16]. Kafka's focus on the log abstraction is reminiscent of the Unix philosophy [17]: "Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new 'features'."

Real systems do require indexes and caches, but these can be derived from the log by a Kafka consumer that writes messages to an indexed store, either in-process (for local access) or to a remote database (for access by other applications). Because all consumers see messages in the same partition in the same order, deterministic consumers can independently construct views that are consistent with each other – an approach known as state machine replication [27]. The truth is in the log, and a database is a cached subset of the log [9].

4.2 Composing Stream Operators

Each Samza job is structurally simple: it is just one step in a data processing pipeline, with Kafka topics as inputs and outputs. If Kafka is like a streaming version of HDFS, then Samza is like a streaming version of MapReduce. The pipeline is loosely coupled, since a job does not know the identity of the jobs upstream or downstream from it, only the topic names. This principle again evokes a Unix maxim [17]: "Expect the output of every program to become the input to another, as yet unknown, program."

However, there are some key differences between Kafka topics and Unix pipes. In particular, Kafka preserves the advantages of batch workflows discussed in Section 1.2: a topic can have any number of consumers that do not interfere with each other (including consumers operated by different teams, or special consumers for debugging or monitoring), it tolerates failure of producers, consumers or brokers, and a topic is a named entity that can be used for tracing data provenance.

Kafka topics deliberately do not provide backpressure: the on-disk log acts as an almost-unbounded buffer of messages. If a slow consumer falls behind the producer, the producers and other consumers continue operating at full speed. Thus, one faulty process does not disrupt the rest of the system, which improves operational reliability. Since Kafka stores all messages on disk anyway, buffering messages for a slow consumer does not incur additional overhead. The slow consumer can catch up without missing messages, as long as it does not fall behind further than Kafka's retention period of log segments, which is usually on the order of days or weeks.

Moreover, Kafka offers the ability for a consumer to jump back to an earlier point in the log, or to rebuild the entire state of a database replica by consuming from the beginning of a log-compacted topic. This facility makes it feasible to use stream processors not only for ephemeral event data, but also for database-like use cases.

Even though the intermediate state between two Samza stream processing operators is always materialized to disk, Samza is able to provide good performance: a simple stream processing job can process over 1 million messages per second on one machine, and saturate a gigabit Ethernet NIC [7].

4.3 Unix as a Role Model

Unix and databases are both data management systems [24], allowing data to be stored (in files or tables) and processed (through command-line tools or queries). Unix tools are famously well suited for implementing ad-hoc, experimental, short-running data processing tasks [21], whereas databases have traditionally been the tool of choice for building complex, long-lived applications. If our goal is to build stream processing applications that will run reliably for many years, is Unix really a good role model?

The database tradition favors clean high-level semantics (the relational model) and declarative query languages. While this approach has been very successful in many domains, it has not worked well in the context of building large-scale personalized services, because the algorithms required for these use cases (such as statistical machine learning and information retrieval methods) are not amenable to implementation using relational operators [28, 29].

Moreover, different use cases have different access patterns, which require different indexing and storage methods. It may be necessary to store the same data in both a traditional row-oriented fashion with indexes, as well as columnar storage, pre-aggregated OLAP cubes, inverted full-text search indexes, sparse matrices or array storage. Rather than trying to implement everything in a single product, most databases specialize in implementing one of these storage methods well (which is hard enough already).

In the absence of a single database system that can provide all the necessary functionality, application developers are forced to combine several data storage and processing systems that each provide a portion of the required application functionality. However, many traditional database systems are not designed for such composition: they focus on providing strong semantics internally, rather than integration with external systems. Mechanisms for integrating with external systems, such as change data capture, are often ad-hoc and retrofitted [5].

By contrast, the log-oriented model of Kafka and Samza is fundamentally built on the idea of composing heterogeneous systems through the uniform interface of a replicated, partitioned log. Individual systems for data storage and processing are encouraged to do one thing well, and to use logs as input and output. Even though Kafka's logs are not the same as Unix pipes, they encourage composability, and thus Unix-style thinking.

4.4 Limitations

Kafka guarantees a total ordering of messages per partition, even in the face of crashes and network failures. This guarantee is stronger than most "eventually consistent" datastores provide, but not as strong as serializable database transactions.

The stream-processing model of computation is fundamentally asynchronous: if a client issues a write to the log, and then reads from a datastore that is maintained by consuming the log, the read may return a stale value. This decoupling is desirable, as it prevents a slow consumer from disrupting a producer or other consumers (Section 4.2). If linearizable data structures are required, they can fairly easily be implemented on top of a totally ordered log [3].

If a Kafka consumer or Samza job crashes and restarts, it resumes consuming messages from the most recently checkpointed offset. Thus, any messages processed between the last checkpoint and the crash are processed twice, and any non-idempotent operations (such as the counter increment in `CountWords`, Figure 2) may yield non-exact results. There is work in progress to add a multi-partition atomic commit protocol to Kafka [11], which will allow exactly-once semantics to be achieved.

Samza uses a low-level one-message-at-a-time programming model, which is very flexible, but also harder to use, more error-prone and less amenable to automatic optimization than a high-level declarative query language. Work is currently in progress in the Kafka project to implement a high-level dataflow API called *Kafka Streams*, and the Samza project is developing a SQL query interface, with relational operators implemented as stream processing tasks. These higher-level programming models enable easier development of applications that fit the model, while retaining the freedom for applications to use the lower-level APIs when required.

5 Conclusion

We present the design philosophy behind Kafka and Samza, which implement stream processing by composing a small number of general-purpose abstractions. We draw analogies to the design of Unix, and batch processing pipelines. The approach reflects broader trends: the convergence between batch and stream processing [1, 4], and the decomposition of monolithic data infrastructure into a collection of specialized services [12, 28].

In particular, we advocate a style of application development in which each data storage and processing component focuses on "doing one thing well". Heterogeneous systems can be built by composing such specialised tools through the simple, general-purpose interface of a log. Compared to monolithic systems, such composable systems provide better scalability properties thanks to loose coupling, and allow easier adaptation of a system to a wide range of different workloads, such as recommendation systems.

Acknowledgements

Large portions of the development of Kafka and Samza were funded by LinkedIn. Many people have contributed, and the authors would like to thank the committers on both projects: David Arthur, Sriharsha Chintalapani, Yan Fang, Jakob Homan, Joel Koshy, Prashanth Menon, Neha Narkhede, Yi Pan, Navina Ramesh, Jun Rao, Chris Riccomini, Gwen Shapira, Zhijie Shen, Chinmay Soman, Joe Stein, Sriram Subramanian, Garry Turkington, and Guozhang Wang. Thank you to Garry Turkington, Yan Fang and Alastair Beresford for feedback on a draft of this article.

References

[1] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, et al. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8(12):1792–1803, August 2015. doi:10.14778/2824032.2824076.

[2] Shivnath Babu and Herodotos Herodotou. Massively parallel databases and MapReduce systems. *Foundations and Trends in Databases*, 5(1):1–104, November 2013. doi:10.1561/1900000036.

[3] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, et al. Tango: Distributed data structures over a shared log. In *24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 325–340, November 2013. doi:10.1145/2517349.2522732.

[4] Raul Castro Fernandez, Peter Pietzuch, Jay Kreps, Neha Narkhede, et al. Liquid: Unifying nearline and offline big data integration. In *7th Biennial Conference on Innovative Data Systems Research (CIDR)*, January 2015.

[5] Shirshanka Das, Chavdar Botev, Kapil Surlaker, Bhaskar Ghosh, et al. All aboard the Databus! In *3rd ACM Symposium on Cloud Computing (SoCC)*, October 2012.

[6] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *6th USENIX Symposium on Operating System Design and Implementation (OSDI)*, December 2004.

[7] Tao Feng. Benchmarking Apache Samza: 1.2 million messages per second on a single node, August 2015. URL `http://engineering.linkedin.com/performance/benchmarking-apache-samza-12-million-messages-second-single-node`.

[8] Ken Goodhope, Joel Koshy, Jay Kreps, Neha Narkhede, et al. Building LinkedIn's real-time activity data pipeline. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 35(2):33–45, June 2012.

[9] Pat Helland. Immutability changes everything. In *7th Biennial Conference on Innovative Data Systems Research (CIDR)*, January 2015.

[10] Martin Kleppmann. Schema evolution in Avro, Protocol Buffers and Thrift, December 2012. URL `http://martin.kleppmann.com/2012/12/05/schema-evolution-in-avro-protocol-buffers-thrift.html`.

[11] Joel Koshy. Transactional messaging in Kafka, July 2014. URL `https://cwiki.apache.org/confluence/display/KAFKA/Transactional+Messaging+in+Kafka`.

[12] Jay Kreps. *I Heart Logs*. O'Reilly Media, September 2014. ISBN 978-1-4919-0932-4.

[13] Jay Kreps. Why local state is a fundamental primitive in stream processing, July 2014. URL http://radar.oreilly.com/2014/07/why-local-state-is-a-fundamental-primitive-in-stream-processing.html.

[14] Jay Kreps. Benchmarking Apache Kafka: 2 million writes per second (on three cheap machines), April 2014. URL https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines.

[15] Jay Kreps. Putting Apache Kafka to use: a practical guide to building a stream data platform (part 2), February 2015. URL http://blog.confluent.io/2015/02/25/stream-data-platform-2/.

[16] Jay Kreps, Neha Narkhede, and Jun Rao. Kafka: a distributed messaging system for log processing. In *6th International Workshop on Networking Meets Databases (NetDB)*, June 2011.

[17] M D McIlroy, E N Pinson, and B A Tague. UNIX time-sharing system: Foreword. *The Bell System Technical Journal*, 57(6):1899–1904, July 1978.

[18] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (LSM-Tree). *Acta Informatica*, 33(4):351–385, June 1996. doi:10.1007/s002360050048.

[19] Todd Palino. Running Kafka at scale, March 2015. URL https://engineering.linkedin.com/kafka/running-kafka-scale.

[20] Kartik Paramasivam. How we're improving and advancing Kafka at LinkedIn, September 2015. URL http://engineering.linkedin.com/apache-kafka/how-we%E2%80%99re-improving-and-advancing-kafka-linkedin.

[21] Rob Pike and Brian W Kernighan. Program design in the UNIX environment. *AT&T Bell Laboratories Technical Journal*, 63(8):1595–1605, October 1984. doi:10.1002/j.1538-7305.1984.tb00055.x.

[22] Jun Rao. How to choose the number of topics/partitions in a Kafka cluster?, March 2015. URL http://www.confluent.io/blog/how-to-choose-the-number-of-topicspartitions-in-a-kafka-cluster/.

[23] Azarias Reda, Yubin Park, Mitul Tiwari, Christian Posse, and Sam Shah. Metaphor: A system for related search recommendations. In *21st ACM International Conference on Information and Knowledge Management (CIKM)*, October 2012.

[24] Dennis M Ritchie and Ken Thompson. The UNIX time-sharing system. *Communications of the ACM*, 17(7), July 1974. doi:10.1145/361011.361061.

[25] Mendel Rosenblum and John K Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, February 1992. doi:10.1145/146941.146943.

[26] Sriram Sankar. Did you mean "Galene"?, June 2014. URL https://engineering.linkedin.com/search/did-you-mean-galene.

[27] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[28] Margo Seltzer. Beyond relational databases. *Communications of the ACM*, 51(7):52–58, July 2008. doi:10.1145/1364782.1364797.

[29] Michael Stonebraker and Uğur Çetintemel. "One size fits all": An idea whose time has come and gone. In *21st International Conference on Data Engineering (ICDE)*, April 2005.

[30] Roshan Sumbaly, Jay Kreps, and Sam Shah. The "Big Data" ecosystem at LinkedIn. In *ACM International Conference on Management of Data (SIGMOD)*, July 2013.

[31] Guozhang Wang. KIP-28 — add a processor client, July 2015. URL https://cwiki.apache.org/confluence/display/KAFKA/KIP-28+-+Add+a+processor+client.

[32] Guozhang Wang, Joel Koshy, Sriram Subramanian, Kartik Paramasivam, et al. Building a replicated logging system with Apache Kafka. *Proceedings of the VLDB Endowment*, 8(12):1654–1655, August 2015. doi:10.14778/2824032.2824063.

[33] Lili Wu, Sam Shah, Sean Choi, Mitul Tiwari, and Christian Posse. The browsemaps: Collaborative filtering at LinkedIn. In *6th Workshop on Recommender Systems and the Social Web (RSWeb)*, October 2014.

[34] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, April 2012.

Streaming@Twitter

Maosong Fu, Sailesh Mittal, Vikas Kedigehalli, Karthik Ramasamy, Michael Barry,
Andrew Jorgensen, Christopher Kellogg, Neng Lu, Bill Graham, Jingwei Wu

Twitter, Inc.

Abstract

Twitter generates tens of billions of events per hour when users interact with it. Analyzing these events to surface relevant content and to derive insights in real-time is a challenge. To address this, we developed Heron, a new real time distributed streaming engine. In this paper, we first describe the design goals of Heron and show how the Heron architecture achieves task isolation and resource reservation to ease debugging, troubleshooting, and seamless use of shared cluster infrastructure with other critical Twitter services. We subsequently explore how a topology self adjusts using back pressure so that the pace of the topology goes as its slowest component. Finally, we outline how Heron implements at-most-once and at-least-once semantics and we describe a few operational stories based on running Heron in production.

1 Introduction

Stream-processing platforms enable enterprises to extract business value from data in motion, similar to batch processing platforms that facilitated the same with data at rest [42]. The goal of stream processing is to enable real-time or near real-time decision making by providing capabilities to inspect, correlate and analyze data as it flows through data-processing pipelines. There is an emerging trend to transition from predominant batch analytics to streaming analytics driven by a combination of increased data collection in real-time and the need to make decisions instantly. Several scenarios in different industries require stream processing capabilities that can process millions and even hundreds of millions of events per second. Twitter is no exception.

Twitter is synonymous with real-time. When a user tweets, his or her tweet can reach millions of users instantly. Twitter users post several hundred million tweets every day. These tweets vary in diversity of content [28] including but not limited to news, pass along (information or URL sharing), status updates (daily chatter), and real-time conversations surrounding events such as the Super Bowl, and the Oscars. Due to the volume and variety of tweets, it is necessary to surface relevant content in the form of break-out moments and trending #hashtags to users in real time. In addition, there are several real-time use cases including but not limited to analyzing user engagements, extract/transform/load (ETL), and model building.

In order to power the aforementioned crucial use cases, Twitter developed an entirely new real-time distributed stream-processing engine called Heron. Heron is designed to provide

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

- **Ease of Development and Troubleshooting**: Users can easily debug and identify the issues in their topologies (also called standing queries), allowing them to iterate quickly during development. This improvement in visibility is possible because of the fundamental change in architecture in Heron from thread based to process based. Users can easily reason about how their topologies work, and profile and debug their components in isolation.
- **Efficiency and Performance**: Heron is 2-5x more efficient than Storm [40]. This improvement resulted in significant cost savings for Twitter both in capital and operational expenditures.
- **Scalability and Reliability**: Heron is highly scalable both in the ability to execute large numbers of components for each topology and the ability to launch and track large numbers of topologies. This large scale results from the clean separation of topology scheduling and monitoring.
- **Compatibility with Storm**: Heron is API compatible with Storm and hence no code change is required for migration.
- **Simplified and Responsive UI**: The Heron UI gives a visual overview of each topology. The UI uses metrics to show at a glance where the hot spots are and provides detailed counters for tracking progress and troubleshooting.
- **Capacity Allocation and Management**: Users can take a topology from development to production in a shared-cluster infrastructure instantly, since Heron runs as yet another framework of the scheduler that manages capacity allocation.

The remainder of this paper is organized as follows. Section 2 presents related work on streaming systems. The following section, Section 3 describes the Heron data model. Section 4 describes the Heron architecture followed by how the architecture meets the design goals in Section 5. Section 6 discusses some of the operational aspects that we encountered while running Heron at Twitter specifically back-pressure issues in Section 6.1, load shedding in Section 6.2, and Kestrel spout issues in Section 6.3. Finally, Section 7 contains our conclusions and points to a few directions for future work.

2 Related Work

The importance of stream-processing systems was recognized in the late 1990s and early 2000s. From then on, these systems have gone through three generations of evolution. First-generation systems were either main-memory database systems or rule engines that evaluate rules expressed as condition-action pairs when new events arrive. When a rule is triggered, it might produce alerts or modify the internal state, which could trigger other rules. These systems were limited in functionality and also did not scale with large-data-volume streams. Some of the systems in this generation include HiPAC [29], Starburst [43], Postgres [37], Ode [31], and NiagaraCQ [27].

Second-generation systems were focused on extending SQL for processing streams by exploiting the similarities between a stream and a relation. A stream is considered as an instantaneous relation [22] and streams can be processed using relational operators. Furthermore, the stream and stream results can be stored in relations for later querying. TelegraphCQ [25] focused on developing novel techniques for processing streams of continuous queries over large volume of data using Postgres. Stanford stream manager STREAM [21] proposed a data model integrating streams into SQL. Aurora [18] used operator definitions to form a directed acyclic graph (DAG) for processing stream data in a single node system. Borealis [17] extended Aurora for distributed stream processing with a focus on fault tolerance and distribution. Cayuga [30] is a stateful publishe-subscribe system that developed a query language for event processing based on an algebra using non-deterministic finite state automaton.

Because these second-generation systems were not designed to handle incoming data in a distributed fashion, a need for a third generation arose as Internet companies began producing data at a high velocity and volume. These third-generation systems were developed with the key focus on scalable processing of streaming data. Yahoo S4 [3] is one of the earliest distributed streaming systems that is near real-time, scalable and allows for easy implementation of streaming applications. Apache Storm [40] is a widely popular distributed streaming system open sourced by Twitter. It models a streaming analytics job as a DAG and runs each node of the DAG as several tasks distributed across a cluster of machines. MillWheel [19] is a key-value based streaming system that supports exactly once semantics. It uses BigTable [26] for storing state and checkpointing. Apache Samza [4] developed at LinkedIn, is a real-time, asynchronous computational framework for stream processing. It uses several independent single-stage computational tasks for stitching together a topology similar to Storm. Each stage reads one or more streams from Apache Kafka [32] and writes the output stream to Kafka for stitching together a processing DAG.

Apache Spark [5] supports streaming using a high-level abstraction called a discretized stream, Spark runs short tasks to process these discretized streams and output results to other systems. In contrast, Apache Flink [2] uses a distributed streaming dataflow engine and asynchronous snapshots for achieving exactly once semantics. Pulsar [35] is a real time analytics engine open sourced by eBay and its unique feature is its SQL interface. Some of the other notable systems include S-Store [34] Akka [1], Photon [20], and Reactive Streams [11]. In addition to these platforms, several commercial streaming systems are available in the market [7], [8], [9], [12], [13]i, [14], and [15].

3 Heron Data Model

Heron uses a directed acyclic graph (DAG) for representing a real-time computation. The graph is referred to as a *topology*. Each node in the topology contains processing logic, and the links between the nodes indicate how the data flows between them. These data flows are called *streams*. A stream is an unbounded sequence of tuples. Nodes take one or more streams and transform them into one or more new new streams. There are two types of nodes: *spouts* and *bolts*. Spouts are the sources of streams. For example, a Kafka [32] spout can tap into a Kafka queue and emit it as a stream. A bolt consumes tuples from streams, applies its processing logic and emits tuples in outgoing streams. Typical processing logic includes filtering, joining and aggregation of streams. An example topology is shown in Figure 1.

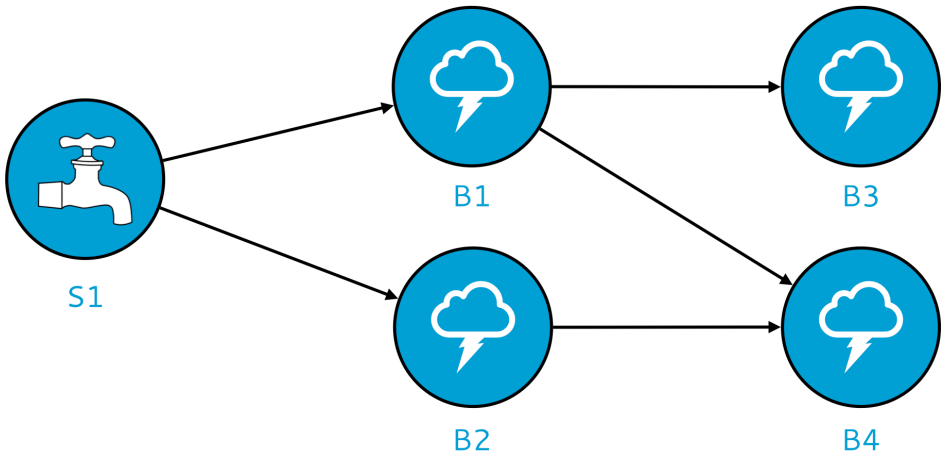


Figure 1: Heron Topology

In this topology, the spouts S1 taps into its data source and emits two streams consumed by the first stage

bolts B1, and B2. These bolts transform the streams and emit three new streams feeding bolts B3 and B4. Since the incoming data rate might be higher than the processing capability of a single process or even a single machine, each spout and bolt of the topology is run as multiple tasks. The number of tasks for each spout and bolt is specified in the topology configuration by the programmer. Such a task specification is referred to as the degree of parallelism. The topology shown in Figure 1, when instantiated at run time is illustrated in Figure 2. The topology, the task parallelism for each node and the specification about how data should be routed form the physical execution plan of the topology.

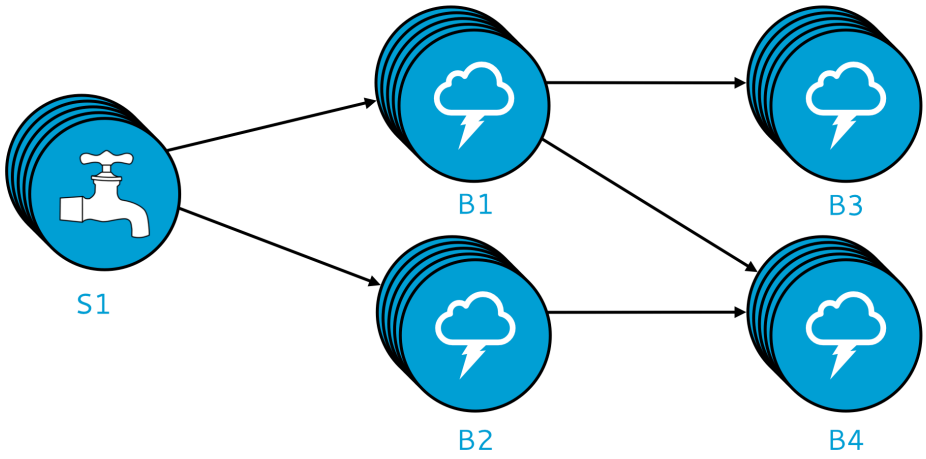


Figure 2: Physical Execution of a Heron Topology

4 Heron Architecture

The design goals for Heron are multifold. First, the spout and bolt tasks need to be executed in isolation. Such isolation will provide the ability to debug and profile a task when needed. Second, the resources allocated to the topology should not be exceeded during the execution of the topology. This requirement enables Heron topologies to be run in a shared cluster environment alongside other critical services. Third, the Heron API should be backward compatible with Storm and a migrated topology should run unchanged. Fourth, Heron topologies should adjust themselves automatically when some of their components are executing slowly. Fifth, Heron should be able to provide high throughput and low latency. While these goals are often mutually exclusive, Heron should expose the appropriate knobs so that users can balance throughput and latency needs. Sixth, Heron should support the processing semantics of at most once and at least once. Finally, Heron should be able to achieve high throughput and/or low latency while consuming a minimal amount of resources.

To meet the aforementioned design goals, Heron uses the architecture as shown in Figure 3. A user writes his or her topology using the Heron API and submits to a scheduler. The scheduler acquires the resources (CPU and RAM) as specified by the topology and spawns multiple containers on different nodes. The first container, referred to as the *master container*, runs the *topology master*. The other containers each run a *stream manager*, a *metrics manager* and several processes called *instances* that execute the processing logic of spouts and bolts.

The topology master is responsible for managing the entire topology. Furthermore, it assigns a role or group based on the user who launched the topology. This role is used to track the resource usage of topologies across different teams and calcuate the cost of running them for reporting. In addition, the topology master acts as the gateway to access the metrics and status of the topology. Once the topology master comes up in the master container, it advertises its location in the form of a host and port via an ephemeral Zookeeper [6] node. This node allows other containers to discover the location of the topology master and also prevents multiple topology

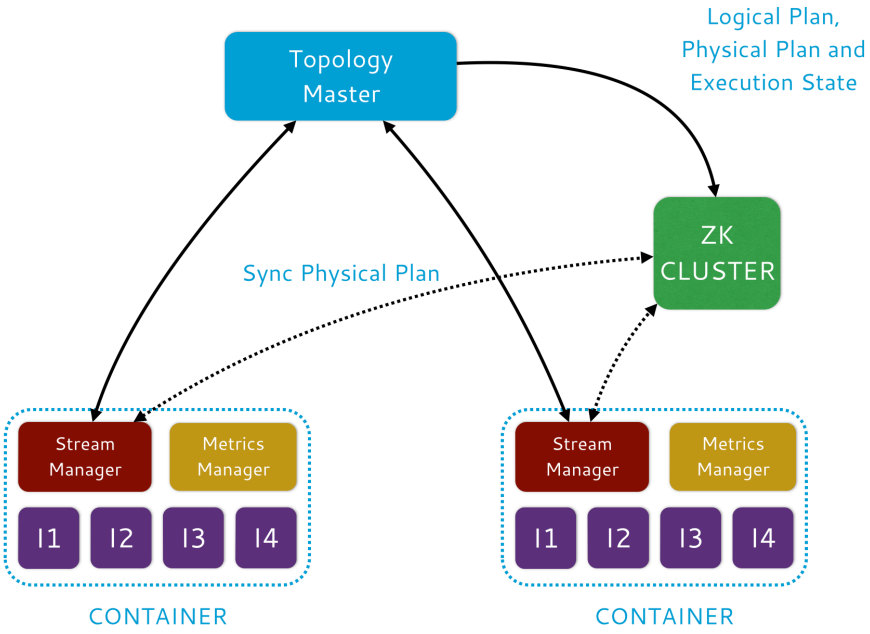


Figure 3: Heron Topology Architecture

masters becoming master during network partitioning. We use an ephemeral node in Zookeeper because when the topology master dies, it detects the loss of session and automatically removes the node.

A network of stream managers route data tuples from one Heron instance to other Heron instances. Each container has a stream manager and the Heron instances in that container send and receive data from it. Even data tuples destined for local Heron instances in a container are routed through the stream manager. When a container is scheduled, the stream manager comes up and discovers where the topology master is running. The stream manager forms a handshake request that includes the host and port on which it is listening and sends it to the topology master. This host and port information allows the topology master to assemble the physical plan and push the plan to all the stream managers. Once stream managers get the physical plan, they connect with other stream managers to form a fully connected graph, as shown in Figure 3.

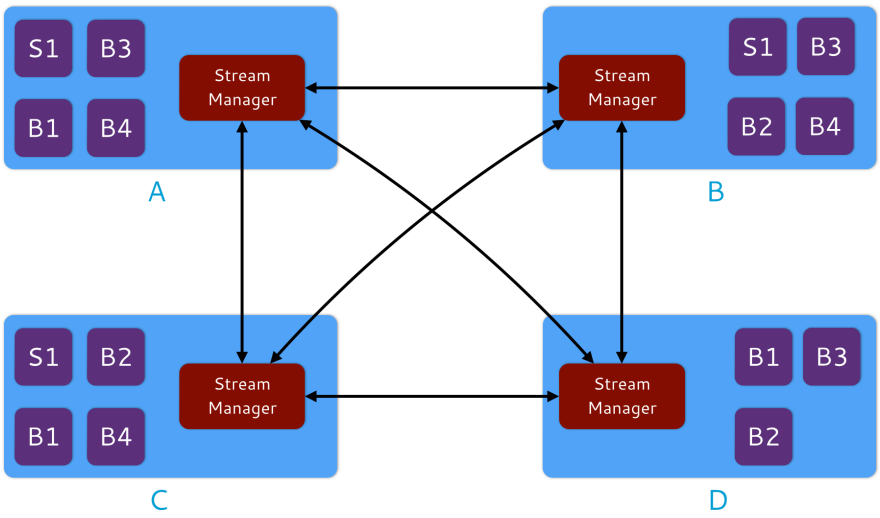


Figure 4: Dataflow in Heron

A Heron instance runs the processing logic in spouts or bolts. Each Heron instance is a process running a single spout task or a bolt task. The instance process runs two threads –the gateway thread and the task-execution thread. The gateway thread communicates with the stream manager to send and receive data tuples from the stream manager. The task-execution thread runs the user code of the spout or bolt. When the gateway thread receives tuples, it passes them to the task-execution thread. The task-execution thread applies the processing logic and emits tuples, if needed. These emitted tuples are sent to the gateway thread, which passes them to the stream manager. In addition to tuples, the task-execution thread collects several metrics. These are passed to the gateway thread, which routes them to the metrics manager.

The metrics manager is responsible for collecting metrics from all instances and exporting them to the metrics-collection system. The metrics-collection system stores those raw metrics and allows for later analysis. Since there are several popular metrics-collection systems, the metrics manager exposes a generic abstraction. This abstraction facilitates ease of implementation for routing metrics to various different metrics-collection systems.

5 Achieving Design Goals

As mentioned in the previous section, Heron was developed with certain design goals in mind. In this section, we examine how we achieved each one of them in detail.

5.1 Task Isolation

Since a Heron instance executes a single task in a dedicated process, it is entirely isolated from other spout and bolt tasks. Such task isolation provides several advantages. First, it is easy to debug an offending task, since the logs from its instance are written to a file of its own providing a time ordered view of events. This ordering helps simplify debugging. Second, one can use performance-tracking tools (such as YourKit [16], etc) to identify the functions consuming substantial time, when a spout or bolt task is running slowly. Third, it allows examination of the memory of the process to identify large objects and provide insights. Finally, it facilitates the examination of execution state of all threads in the process to identify synchronization issues.

5.2 Resource Reservation

In Heron, a topology requests its resources in the form of containers, and the scheduler spawns those containers on the appropriate machines. Each container is assigned the requested number of CPU cores and memory. Once a certain amount of resources (CPU and RAM) are assigned to a topology, Heron ensures that they are not exceeded. This monitoring is needed when Heron topologies are run alongside other critical services in a shared infrastructure. Furthermore, when fragments of multiple topologies are executing in the same machine, resource reservation ensures that one topology does not influence other topologies by consuming more resources temporarily. If resource reservation is not enforced, it would lead to unpredictability in the behavior of other topologies, making it harder to track the underlying performance issues. Each container is mapped to a Linux cgroup. This ensures that the container does not exceed the allocated resources. If there is an attempt to temporarily consume more resources, the container will be throttled, leading to a slowdown of the topology.

5.3 Self Adjustment

A typical problem seen in streaming systems, similar to what is seen in batch systems, is that of stragglers. Since the topology can process data only as fast as its slowest component, stragglers cause lag in the input data to build up. In such scenarios, a streaming system tends to drop data at different stages of the DAG. This dropping of results in either data loss or replay of data multiple times. A topology needs to adjust its pace depending on the

prevailing situations. Some of these situations are data skew, where a bolt instance is receiving more data than it can process, and when a fragment of the topology is scheduled on a slow node.

During such scenarios, some feedback mechanism should be incorporated to slow down the topology temporarily so that the data drops are minimized. Heron implements a full fledged back-pressure mechanism to ensure that the topology is self adjusting. We investigated two back-pressure approaches –TCP-based back pressure and spout-based back pressure.

The TCP protocol uses slow-start and sliding-window mechanisms to ensure that the sender is transmitting at the rate the receiver can consume. Hence it is natural to ask whether Heron could leverage the TCP protocol for back pressure. But due to the multiplexing nature of the stream manager, where multiple logical transport channels are mapped on a single physical channel, TCP-based back pressure could slow upstream or downstream spouts or bolts. To illustrate this possibility, consider the physical execution of the topology in Figure 1 with four containers as shown in Figure 5. Assume that an instance of Bolt B3 in Container A is going slow. As shown in Figure 1, Bolt B3 receives input from Bolt B1 which means all instances of Bolt B3 will receive input from all instances of B1. Hence, the stream manager in Container A will receive input from bolt instances of B1 running in Containers C and D. Since the instance of Bolt B3 in Container A is going slow, its stream manager will not take any additional input from the stream managers of the containers C and D. Since the connection between stream managers use TCP sockets, eventually the socket send buffers in stream managers in Containers C and D will fill up. As a result, the data exchange between Bolt B1 and B2 (shown in green) in containers C and D with bolt B4 (shown in green) in Container A is affected. We found that for some topologies, such situations could eventually drive the throughput to zero.

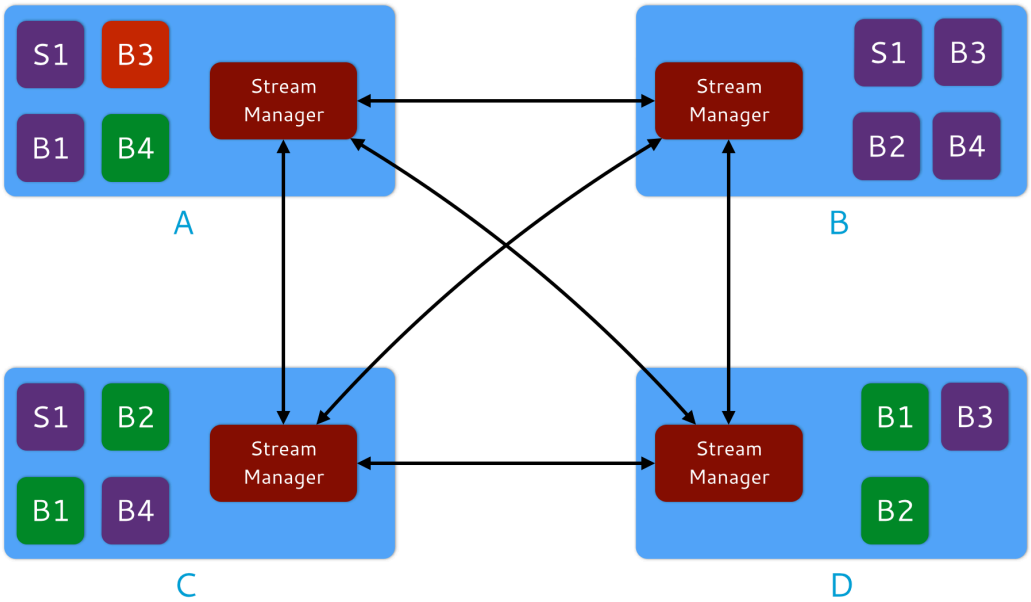


Figure 5: TCP Back Pressure

We considered another approach called spout-based back pressure. This approach is based on the observation that spouts are the sources of data and we can manage when they emit or suspend the injection of data. In other words, whenever a stream manager detects one of the instances is going slow, it will explicitly send an initiate-back-pressure message to all the other stream managers. When a stream manager receives this message, it examines the physical plan and, if there are any spouts running in the container, it will not consume data from them. To illustrate, again consider the physical execution of topology in Figure 1 as shown in Figure 6. When the Bolt B3 in Container A goes slower, its stream manager sends the initiate-back-pressure message to stream

managers of all the containers. Upon receiving this message, the stream managers in Containers B and C do not consume data from their spouts, in this case, Spout S1 (shown in blue). This action reduces the data inflow into the topology thereby self adjusting. Once the Bolt B3 picks up pace, its stream manager sends a relieve-back-pressure message to all other stream managers. They act on this message by starting to consume from their local spouts. More details about the back pressure mechanism can be found in Kulkarni, et al. [33].

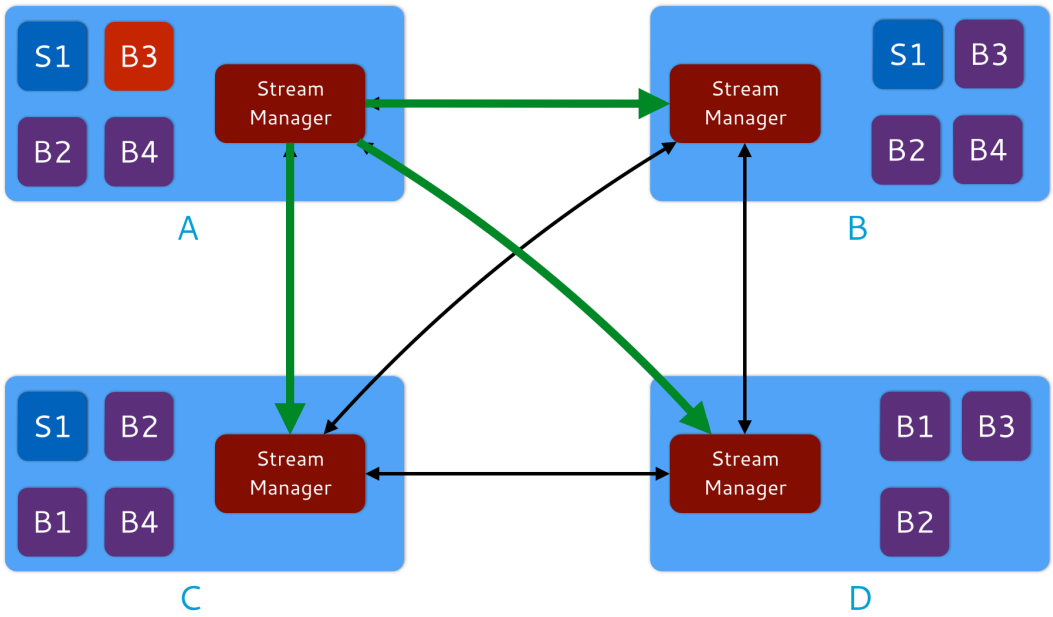


Figure 6: Spout Back Pressure

5.4 Processing Semantics

In order to provide predictability, a stream processing system needs to provide guarantees on the data that passes through it. Heron supports two different types of processing semantics:

- **At most once:** In this semantics, the processing is best effort. In the presence of node or process failures, the data processed by the streaming system could be lost. Hence, the number of data tuples processed might be lower than the actual number of data tuples, which could affect the results.
- **At least once:** In this semantics, the system guarantees that the data is processed at least once. If the data is dropped during node or process failures, it is reprocessed. It is possible that the same data tuple is processed more than once. Hence, the number of data tuples processed might be higher than the actual number of data tuples, again potentially affecting the results.

Incorporating at-most-once semantics in Heron is straight forward. A Heron topology continuously processes data and, during processing, the data moves from instance to stream manager and between stream managers. When an instance in a container fails, the state accumulated by the bolt or spout is lost. After restart, it connects with the stream manager and continues to receive and process data thereby, accumulating new state. Similarly, when a stream manager in a container dies, it restarts and reconnects to other stream managers and resumes processing. If an entire container fails due to node failure, the container is relocated to another node. Once the stream manager and instances in the relocated container come up, the data processing continues. During relocation, the data intended for the failed stream manager from other stream managers could be dropped or if the data is buffered, the buffers could overflow, eventually dropping data.

6 Heron in Practice

Heron has been in production at Twitter for over two years. It is used for diverse use cases such as real-time business intelligence, real-time machine-learning, real-time classification, real-time engagements, computing real-time trends, real-time media, and real-time monitoring. In this section, we will explore some of operational issues that occur in practice and how we solve them.

6.1 Back Pressure

Spout-based back pressure helped us reduce data loss significantly as stragglers are the norm in multi-tenant distributed systems. The Heron back-pressure recovery mechanism allows us to process data at a maximal rate such that the recovery times are very low. Since most topologies are provisioned with extra capacity to handle increased traffic during well-known events (such as the Super Bowl and the Oscars), the recovery rate is usually much higher than the steady state. In cases where the topologies have not been provisioned to handle increased traffic, the back pressure mechanisms act as a shock absorber to handle any temporary spikes. In cases where these spikes are not temporary, back pressure also allows users to add more capacity and restart their topologies with minimal loss of data.

We have encouraged topology writers to test their back pressure (and recovery) mechanism in staging environments by artificially creating traffic spikes (e.g., by reading from older offsets in Kafka). This practice allows them to understand the dynamic behavior of back pressure and measure the recovery time. To monitor this process in real time, several metrics have been exposed on the dashboard. Back pressure also helps topology writers in tuning their topology. Since we do not have auto tuning (yet), users are required to use trial and error to get the correct values for resource allocation and parallelism of the components. By looking at the back pressure metrics, they can identify which of the components are under back pressure and correspondingly increase the resources or parallelism until there is no back pressure in steady state.

In our experience, we have found that in most scenarios, back pressure recovers without manual intervention. However, there are cases where a particular component in topology gets scheduled on a faulty host or goes into irrecoverable garbage-collection cycles (for various reasons). Under such scenarios, users get paged, upon which they usually restart those components to get the problem fixed. While most users see back pressure as a requirement, some users prefer dropping data as they only care about the latest data. To handle such cases, we added the *load-shedding* feature in spouts as decribed in the following section.

6.2 Load Shedding

Load shedding has been studied extensively in the context of second-generation streaming systems [23, 24, 36, 38, 39, 41]. Most of the proposed alternatives fall into two broad categories, sampling-based approaches and data-dropping-based approaches. The idea behind sampling-based approaches is that if the system can automatically downsample an incoming stream in a predictable way, the user can potentially scale up the results of the computation in order to compensate. For example, if a Heron topology is counting widgets and the stream is being downsampled by 50%, the user can simply multiply the widget counts by two for each widget that is present in the stream and therefore still get approximately correct results.

The common theme of sampling approaches is that a more uniformly sampled stream is easier to reason about and a user could also use the information about the sampling rate to scale the output of the computations, which is a very desirable property. However, for sampling to be useful to applications, it would be important that the sampling was done on a global level.

If each spout instance was individually sampling at different times and different rates the value of uniform sampling to applications programmers is pretty much negated. The system would lose the property that it is easy to reason about the sampling that is happening and also the ability to properly scale the output of the

computation based on the sampling rate. Due to these limitations and its considerably higher complexity, we did not implement the sampling-based approach.

On the other hand, the idea behind dropping-based approaches is that the system will simply drop older data and prefer more recent data when the Heron topology is unable to keep up. Heron spouts are modified such that the user can configure a *lag threshold* and a *lag-adjustment value*. The lag threshold will indicate how much lag is tolerable before the spout drops any data. The lag-adjustment value will indicate how much of the old data the system will drop when this threshold is reached.

Given the two values described above, the system will monitor the lag for each individual spout instance and periodically skip ahead by the lag adjustment value whenever the lag is above the threshold value. A key point here is that the decision to drop data is a completely local decision in each spout instance. There will be no attempt made to synchronize amongst different spouts or otherwise coordinate such that the spouts work together in deciding what data to drop. Each spout drops data from its associated Kafka or Eventbus partition and no communication between spouts will occur.

6.3 Kestrel Spout

Kestrel [10] is a simple distributed message-queuing system. Each Kestrel host handles a set of reliable, and ordered, message queues. A Kestrel cluster consists of several such hosts with no communication between them. Whenever a client is interested in enqueuing or dequeuing an item, it randomly picks a host, thereby providing reliable, loosely ordered message queue behavior. An attractive property of Kestrel is its ability to scale, since servers do not communicate with each other and have no need for any coordination.

Unlike Kafka [32], Kestrel is stateful. In order to maintain state, Kestrel replicates data for every consumer. In other words, Kestrel assumes only one consumer per physical queue. An item in the queue is removed only after a client dequeues and then acknowledges it. If two different instances of a consumer are consuming from the same Kestrel queue, it is guaranteed that they will never receive same item, given that they acknowledge their respective items. If the item is not acknowledged within a specified amount of time, it is placed back in the queue for the next instance to receive.

We started with the open source Kestrel spout and it worked reasonably well. However, as traffic grew, Heron topologies using Kestrel spouts faced several issues, such as:

- One or more Kestrel hosts would start accumulating data and not drain. The immediate resolution is to manually mark those servers as read only until they drain, and enable writes once the number of items to be consumed goes below a certain threshold. This approach presents an operational challenge, especially during non-working hours. When a host is not getting drained, it affects the performance of other queues it needs to service as well. One possible solution is to set maxItems (the maximum number of items held in queue) and maxAge (maximum amount of time an item stays in the queue before it is deleted) limits on the queues to be small, so that the size of queue does not grow to affect other queues on the host. But this solution results in data loss for the job consuming this queue.

- A Kestrel spout would pack the Kestrel client (or connection) along with the data in a tuple. This would cause the spout to become stateless, because when the tuple came back to the spout to get acknowledged, it just extracted the client from the tuple and acknowledged it back to Kestrel host to retire the tuple. The problem with this approach was that the tuple size grew, and it carried extra load for no reason, which resulted in extra data transfers, and more serialization and deserialization costs.

- A Kestrel spout would create a new connected client every time it requested the next batch of items from Kestrel. While this behavior has no effect on topologies with low throughput, for more data-heavy topologies, the number of connections to a host grew without bound. Some of the spout-related configurations,

such as maximum spout pending (limits the number of tuples in flight in a topology, so the spouts do not request an unbounded number of tuples) often hid this problem. Furthermore, creation of many connections exacerbated garbage-collection issues.

The root cause for one or more Kestrel hosts not draining was triggered by the use of Zookeeper to discover Kestrel hosts. Specifically, the Kestrel spout used a service factory for creating a connection to one of the Kestrel hosts in the server set, The factory did not provide any guarantees that all the hosts would be connected and read evenly. As a result, some of the servers were occasionally left out, causing items from those servers to not be consumed. Our initial solution was to fetch all the hosts from the Kestrel server set, and read from each server in a round-robin fashion. This practice ensured that no server is left unread, while giving all the hosts equal priority. This approach worked even during times of high load, because it is assumed that to achieve steady state, the read rate has to be higher than the write rate. So even in case of high load, round robin would drain the full queues, and bring the system to steady state.

Soon we saw an issue where instead of one Kestrel host lagging, all of the hosts were backing up. This issue was traced to one host being unable to respond and because of the round robin policy, all the hosts were read at the pace of the slowest one. The actual slow down of the host was due to disk writes for logging. Hence, an approach was needed to decouple a slow host from others temporarily. To solve the issue, each spout instance is assigned a configurable number of Kestrel hosts. These assignments were not mutually exclusive, and had overlaps. The three main properties of these assignments are:

- Each spout instance reads from a subset (more than one) of Kestrel hosts.
- Each Kestrel host is read by a subset (more than one) spout instances.
- If any two Kestrel hosts, A and B, are read by one spout instance, then there exists a spout instance that reads from host A and not B, and another instance that reads from host B and not A.

The last property ensures that if one Kestrel host slows down, the rest of the hosts will be read without any penalties. And using round-robin reads ensures that the slow host will not be left out, and will still be drained.

The issue of passing a Kestrel client was fixed by mapping each tuple to its Kestrel client using a combination of a generated unique identifier and the original item identifier provided by the Kestrel host. This approach also prevented the creation of several client objects by reusing existing Kestrel client objects. Finally, we added configuration parameters to control both the number of connections per Kestrel host from a spout instance and the number of pending items per connection, which helped in playing nice with Kestrel.

7 Conclusion

Heron has become the de-facto real-time streaming system at Twitter. It runs several hundred development and production topologies and been in production for more than two years. Several teams in Twitter use Heron for making real-time data-driven decisions that are business critical. Heron is used for several diverse use cases ranging from ETL to building machine-learning models and is expanding rapidly. These use cases require additional future work to evolve Heron.

First, manual resource assignment for a topology when it goes production currently requires several iterations. Each iteration involves changing the configuration parameters, recompiling and redeploying. For large topologies, each iteration is very expensive. We want to explore an elegant solution for estimating initial resource requirements using a combination of data-source characteristics, sampling and linear regression. Second, the topologies are often overprovisioned to accommodate peak loads during popular events to avoid manual intervention. This policy led to resource wastage and hence we are investigating approaches where the topology can expand automatically and shrink depending on traffic variations. Third, we want to support a declarative

query paradigm that allows users to write queries faster and be more productive. Fourth, in some uses cases, we have to guarantee data processing by the topology is exactly once. The problems of auto-scaling and exactly once will require distributed partitionable state and additional Heron APIs.

8 Acknowledgements

Thanks to David Maier and Kristin Tufte for providing comments on the initial draft of the paper that helped improved its presentation. Thanks to Jeff Naughton, Deep Medhi and Jignesh Patel for reading the pre-final draft and help improve the presentation. Thanks to Arun Kejariwal for help with LaTeX including setting it up, patiently answering several questions and providing several comments on the first draft.

References

[1] Akka. http://akka.io/.

[2] Apache Flink. https://flink.apache.org/.

[3] Apache S4. http://incubator.apache.org/s4/.

[4] Apache Samza. https://samza.apache.org/.

[5] Apache Spark. https://spark.apache.org/.

[6] Apache Zookeeper. http://zookeeper.apache.org/.

[7] Apama Streaming Analytics. http://www.softwareag.com/corporate/products/apama_webmethods/analytics/overview/default.asp.

[8] Informatica Vibe Data Stream. https://www.informatica.com/products/data-integration/real-time-integration/vibe-data-stream.html#fbid=v8VRdfhc8YI.

[9] InfoSphere Streams: Capture and analyze data in motion. http://www-03.ibm.com/software/products/en/infosphere-streams.

[10] Kestrel: A simple, distributed message queue system. http://twitter.github.io/kestrel.

[11] Reactive Streams. http://incubator.apache.org/s4/.

[12] SAP Event Stream Processor. http://www.sap.com/pc/tech/database/software/sybase-complex-event-processing/index.html.

[13] SQLstream Blaze. http://www.sqlstream.com/blaze/.

[14] TIBCO StreamBase. http://www.streambase.com/.

[15] Vitria OI For Streaming Big Data Analytics. http://www.vitria.com/solutions/streaming-big-data-analytics/benefits/.

[16] YourKit. https://www.yourkit.com/.

[17] D. J. Abadi, Y. Ahmad, M. Balazinska, M. Cherniack, J. hyon Hwang, W. Lindner, A. S. Maskey, E. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the Borealis stream processing engine. In *Proceedings of the Conference on Innovative Data Systems Research*, pages 277–289, 2005.

[18] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12(2), Aug. 2003.

[19] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. Millwheel: Fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, Aug. 2013.

[20] R. Ananthanarayanan, V. Basker, S. Das, A. Gupta, H. Jiang, T. Qiu, A. Reznichenko, D. Ryabkov, M. Singh, and S. Venkataraman. Photon: Fault-tolerant and scalable joining of continuous data streams. In *Proceedings of the 2013 International Conference on Management of Data*, pages 577–588, 2013.

[21] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom. STREAM: The stanford stream data manager (demonstration description). In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 665–665, 2003.

[22] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of the Symposium on Principles of Database Systems*, pages 1–16, Madison, Wisconsin, 2002.

[23] B. Babcock, M. Datar, and R. Motwani. Load shedding techniques for data stream systems. In *Proceedings of the 2003 Workshop on Management and Processing of Data Streams MPDS*, 2003.

[24] B. Babcock, M. Datar, and R. Motwani. Load shedding in data stream systems. In C. Aggarwal, editor, *Data Streams*, volume 31 of *Advances in Database Systems*, pages 127–147. Springer US, 2007.

[25] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 668–668, 2003.

[26] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems*, 26(2), June 2008.

[27] J. Chen, D. J. Dewitt, F. Tian, and Y. Wang. Niagara CQ: A scalable continuous query system for internet databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 379–390, 2000.

[28] S. Dann. Twitter content classification. *First Monday*, 15(12), December 2010. `http://firstmonday.org/ojs/index.php/fm/article/view/2745/2681`.

[29] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ledin, D. McCarthy, A. Rosenthal, S. Sarin, M. J. Carey, M. Livny, and R. Jauhari. The HiPAC project: Combining active databases and timing constraints. *SIGMOD Rec.*, 17(1):51–70, March 1988.

[30] A. Demers, J. Gehrke, M. Hong, B. Panda, M. Riedewald, V. Sharma, and W. White. Cayuga: A general purpose event monitoring system. In *Proceedings of the Conference on Innovative Data Systems Research*, 2007.

[31] N. Gehani and H. V. Jagdish. Ode as an active database: Constraints and triggers. In *Proceedings of the 17th International Conference on Very Large Data Bases*, Barcelona, Spain, 1991.

[32] N. N. Jay Kreps and J. Rao. Kafka: A distributed messaging system for log processing. In *SIGMOD Workshop on Networking Meets Databases*, 2011.

[33] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter Heron: Streaming at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, Melbourne, Australia, 2015.

[34] J. Meehan, N. Tatbul, S. Zdonik, C. Aslantas, U. Cetintemel, J. Du, T. Kraska, S. Madden, D. Maier, A. Pavlo, M. Stonebraker, K. Tufte, and H. Wang. S-Store: streaming meets transaction processing. *Proceedings of VLDB Endowment*, 8(13):2134–2145, Sept. 2015.

[35] S. Murthy and T. Ng. Announcing Pulsar: Real-time Analytics at Scale. `http://www.ebaytechblog.com/2015/02/23/announcing-pulsar-real-time-analytics-at-scale`, Feb. 2015.

[36] S. Senthamilarasu and M. Hemalatha. Load shedding using window aggregation queries on data streams. *International Journal of Computer Applications*, 54(9):42–49, September 2012.

[37] M. Stonebraker and G. Kemnitz. The POSTGRES next generation database management system. *Communications of the ACM*, 34(10):78–92, October 1991.

[38] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *Proceedings of the 29th International Conference on Very Large Data Bases*, pages 309–320, 2003.

[39] N. Tatbul and S. Zdonik. Window-aware Load Shedding for Aggregation Queries over Data Streams. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB'06)*.

[40] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy. Storm@twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 147–156, 2014.

[41] Y.-C. Tu, S. Liu, S. Prabhakar, and B. Yao. Load shedding in stream databases: A control-based approach. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 787–798, 2006.

[42] J. Vijayan. Streaming Analytics: Business Value from Real-Time Data. `http://www.datamation.com/data-center/streaming-analytics-business-value-from-real-time-data.html`.

[43] J. Widom. The Starburst rule system: Language design, implementation, and applications. *IEEE Data Engineering Bulletin, Special Issue on Active Databases*, 15:1–4, 1992.

Apache Flink™: Stream and Batch Processing in a Single Engine

Paris Carbone[†]
Asterios Katsifodimos[*]

Stephan Ewen[‡]
Volker Markl[*]

Seif Haridi[†]
Kostas Tzoumas[‡]

[†]KTH & SICS Sweden
parisc,haridi@kth.se

[‡]data Artisans
first@data-artisans.com

[*]TU Berlin & DFKI
first.last@tu-berlin.de

Abstract

Apache Flink[1] *is an open-source system for processing streaming and batch data. Flink is built on the philosophy that many classes of data processing applications, including real-time analytics, continuous data pipelines, historic data processing (batch), and iterative algorithms (machine learning, graph analysis) can be expressed and executed as pipelined fault-tolerant dataflows. In this paper, we present Flink's architecture and expand on how a (seemingly diverse) set of use cases can be unified under a single execution model.*

1 Introduction

Data-stream processing (e.g., as exemplified by complex event processing systems) and static (batch) data processing (e.g., as exemplified by MPP databases and Hadoop) were traditionally considered as two very different types of applications. They were programmed using different programming models and APIs, and were executed by different systems (e.g., dedicated streaming systems such as Apache Storm, IBM Infosphere Streams, Microsoft StreamInsight, or Streambase versus relational databases or execution engines for Hadoop, including Apache Spark and Apache Drill). Traditionally, batch data analysis made up for the lion's share of the use cases, data sizes, and market, while streaming data analysis mostly served specialized applications.

It is becoming more and more apparent, however, that a huge number of today's large-scale data processing use cases handle data that is, in reality, produced continuously over time. These continuous streams of data come for example from web logs, application logs, sensors, or as changes to application state in databases (transaction log records). Rather than treating the streams as streams, today's setups ignore the continuous and timely nature of data production. Instead, data records are (often artificially) batched into static data sets (e.g., hourly, daily, or monthly chunks) and then processed in a time-agnostic fashion. Data collection tools, workflow managers, and schedulers orchestrate the creation and processing of batches, in what is actually a continuous data processing pipeline. Architectural patterns such as the "lambda architecture" [21] combine batch and stream processing systems to implement multiple paths of computation: a streaming fast path for timely approximate results, and a batch offline path for late accurate results. All these approaches suffer from high latency (imposed by batches),

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

[1]The authors of this paper make no claim in being the sole inventors or implementers of the ideas behind Apache Flink, but rather a group of people that attempt to accurately document Flink's concepts and their significance. Consult Section 7 for acknowledgements.

high complexity (connecting and orchestrating several systems, and implementing business logic twice), as well as arbitrary inaccuracy, as the time dimension is not explicitly handled by the application code.

Apache Flink follows a paradigm that embraces data-stream processing as the unifying model for real-time analysis, continuous streams, and batch processing both in the programming model and in the execution engine. In combination with durable message queues that allow quasi-arbitrary replay of data streams (like Apache Kafka or Amazon Kinesis), stream processing programs make no distinction between processing the latest events in real-time, continuously aggregating data periodically in large windows, or processing terabytes of historical data. Instead, these different types of computations simply start their processing at different points in the durable stream, and maintain different forms of state during the computation. Through a highly flexible windowing mechanism, Flink programs can compute both early and approximate, as well as delayed and accurate, results in the same operation, obviating the need to combine different systems for the two use cases. Flink supports different notions of time (event-time, ingestion-time, processing-time) in order to give programmers high flexibility in defining how events should be correlated.

At the same time, Flink acknowledges that there is, and will be, a need for dedicated batch processing (dealing with static data sets). Complex queries over static data are still a good match for a batch processing abstraction. Furthermore, batch processing is still needed both for legacy implementations of streaming use cases, and for analysis applications where no efficient algorithms are yet known that perform this kind of processing on streaming data. Batch programs are special cases of streaming programs, where the stream is finite, and the order and time of records does not matter (all records implicitly belong to one all-encompassing window). However, to support batch use cases with competitive ease and performance, Flink has a specialized API for processing static data sets, uses specialized data structures and algorithms for the batch versions of operators like join or grouping, and uses dedicated scheduling strategies. The result is that Flink presents itself as a full-fledged and efficient batch processor on top of a streaming runtime, including libraries for graph analysis and machine learning. Originating from the Stratosphere project [4], Flink is a top-level project of the Apache Software Foundation that is developed and supported by a large and lively community (consisting of over 180 open-source contributors as of the time of this writing), and is used in production in several companies.

The contributions of this paper are as follows:

- we make the case for a unified architecture of stream and batch data processing, including specific optimizations that are only relevant for static data sets,
- we show how streaming, batch, iterative, and interactive analytics can be represented as fault-tolerant streaming dataflows (in Section 3),
- we discuss how we can build a full-fledged stream analytics system with a flexible windowing mechanism (in Section 4), as well as a full-fledged batch processor (in Section 4.1) on top of these dataflows, by showing how streaming, batch, iterative, and interactive analytics can be represented as streaming dataflows.

2 System Architecture

In this section we lay out the architecture of Flink as a software stack and as a distributed system. While Flink's stack of APIs continues to grow, we can distinguish four main layers: deployment, core, APIs, and libraries.

Flink's Runtime and APIs. Figure 1 shows Flink's software stack. The core of Flink is the distributed dataflow engine, which executes dataflow programs. A Flink runtime program is a DAG of stateful operators connected with data streams. There are two core APIs in Flink: the DataSet API for processing finite data sets (often referred to as *batch processing*), and the DataStream API for processing potentially unbounded data streams (often referred to as *stream processing*). Flink's core runtime engine can be seen as a streaming dataflow engine, and both the DataSet and DataStream APIs create runtime programs executable by the engine. As such, it serves as the common fabric to abstract both bounded (batch) and unbounded (stream) processing. On top of the core

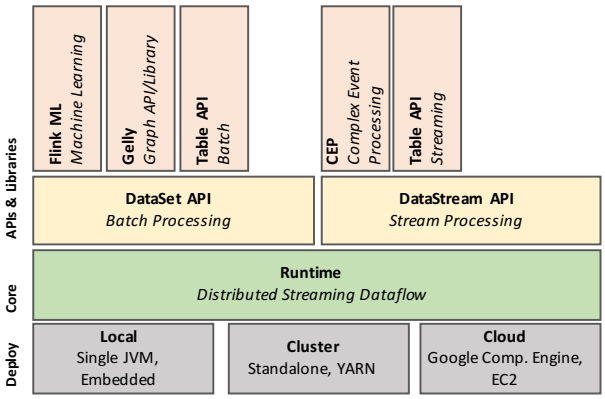


Figure 1: The Flink software stack.

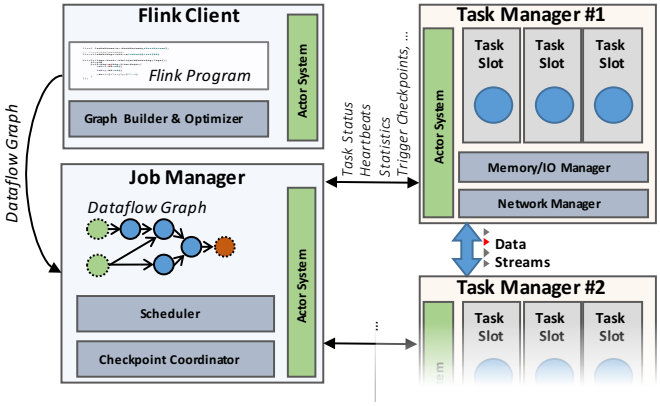


Figure 2: The Flink process model.

APIs, Flink bundles domain-specific libraries and APIs that generate DataSet and DataStream API programs, currently, FlinkML for machine learning, Gelly for graph processing and Table for SQL-like operations.

As depicted in Figure 2, a Flink cluster comprises three types of processes: the client, the Job Manager, and at least one Task Manager. The client takes the program code, transforms it to a dataflow graph, and submits that to the JobManager. This transformation phase also examines the data types (schema) of the data exchanged between operators and creates serializers and other type/schema specific code. DataSet programs additionally go through a cost-based query optimization phase, similar to the physical optimizations performed by relational query optimizers (for more details see Section 4.1).

The JobManager coordinates the distributed execution of the dataflow. It tracks the state and progress of each operator and stream, schedules new operators, and coordinates checkpoints and recovery. In a high-availability setup, the JobManager persists a minimal set of metadata at each checkpoint to a fault-tolerant storage, such that a standby JobManager can reconstruct the checkpoint and recover the dataflow execution from there. The actual data processing takes place in the TaskManagers. A TaskManager executes one or more operators that produce streams, and reports on their status to the JobManager. The TaskManagers maintain the buffer pools to buffer or materialize the streams, and the network connections to exchange the data streams between operators.

3 The Common Fabric: Streaming Dataflows

Although users can write Flink programs using a multitude of APIs, all Flink programs eventually compile down to a common representation: the dataflow graph. The dataflow graph is executed by Flink's runtime engine, the common layer underneath both the batch processing (DataSet) and stream processing (DataStream) APIs.

3.1 Dataflow Graphs

The dataflow graph as depicted in Figure 3 is a directed acyclic graph (DAG) that consists of: (i) stateful operators and (ii) data streams that represent data produced by an operator and are available for consumption by operators. Since dataflow graphs are executed in a data-parallel fashion, operators are parallelized into one or more parallel instances called *subtasks* and streams are split into one or more *stream partitions* (one partition per producing subtask). The stateful operators, which may be stateless as a special case implement all of the processing logic (e.g., filters, hash joins and stream window functions). Many of these operators are implementations of textbook versions of well known algorithms. In Section 4, we provide details on the implementation of windowing operators. Streams distribute data between producing and consuming operators in various patterns, such as point-to-point, broadcast, re-partition, fan-out, and merge.

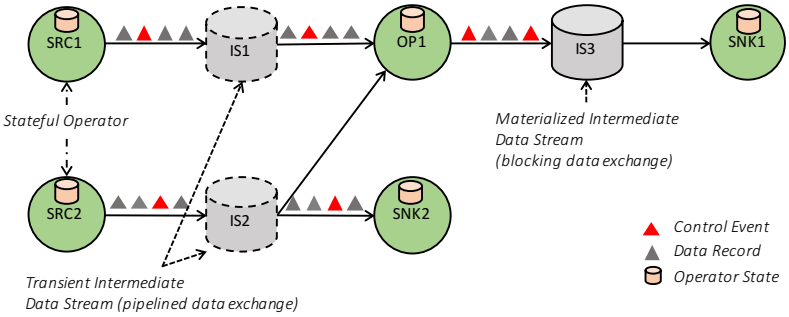


Figure 3: A simple dataflow graph.

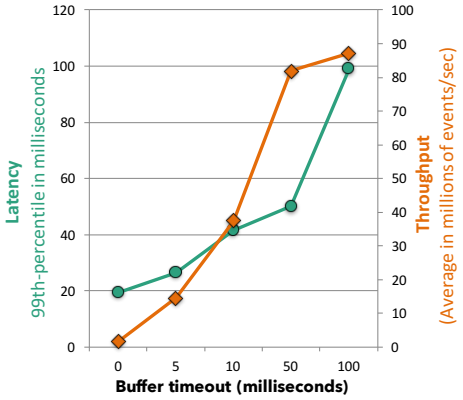


Figure 4: The effect of buffer-timeout in latency and throughput.

3.2 Data Exchange through Intermediate Data Streams

Flink's intermediate data streams are the core abstraction for data-exchange between operators. An intermediate data stream represents a logical handle to the data that is produced by an operator and can be consumed by one or more operators. Intermediate streams are logical in the sense that the data they point to may or may not be materialized on disk. The particular behavior of a data stream is parameterized by the higher layers in Flink (e.g., the program optimizer used by the DataSet API).

Pipelined and Blocking Data Exchange. *Pipelined intermediate streams* exchange data between concurrently running producers and consumers resulting in pipelined execution. As a result, pipelined streams propagate back pressure from consumers to producers, modulo some elasticity via intermediate buffer pools, in order to compensate for short-term throughput fluctuations. Flink uses pipelined streams for continuous streaming programs, as well as for many parts of batch dataflows, in order to avoid materialization when possible. *Blocking streams* on the other hand are applicable to bounded data streams. A blocking stream buffers all of the producing operator's data before making it available for consumption, thereby separating the producing and consuming operators into different execution stages. Blocking streams naturally require more memory, frequently spill to secondary storage, and do not propagate backpressure. They are used to isolate successive operators against each other (where desired) and in situations where plans with pipeline-breaking operators, such as sort-merge joins may cause distributed deadlocks.

Balancing Latency and Throughput. Flink's data-exchange mechanisms are implemented around the exchange of buffers. When a data record is ready on the producer side, it is serialized and split into one or more buffers (a buffer can also fit multiple records) that can be forwarded to consumers. A buffer is sent to a consumer either i) as soon as it is full or ii) when a timeout condition is reached. This enables Flink to achieve high throughput by setting the size of buffers to a high value (e.g., a few kilobytes), as well as low latency by setting the buffer timeout to a low value (e.g., a few milliseconds). Figure 4 shows the effect of buffer-timeouts on the throughput and latency of delivering records in a simple streaming grep job on 30 machines (120 cores). Flink can achieve an observable 99^{th}-percentile latency of 20ms. The corresponding throughput is 1.5 million events per second. As we increase the buffer timeout, we see an increase in latency with an increase in throughput, until full throughput is reached (i.e., buffers fill up faster than the timeout expiration). At a buffer timeout of 50ms, the cluster reaches a throughput of more than 80 million events per second with a 99^{th}-percentile latency of 50ms.

Control Events. Apart from exchanging data, streams in Flink communicate different types of control events. These are special events injected in the data stream by operators, and are delivered in-order along with all other

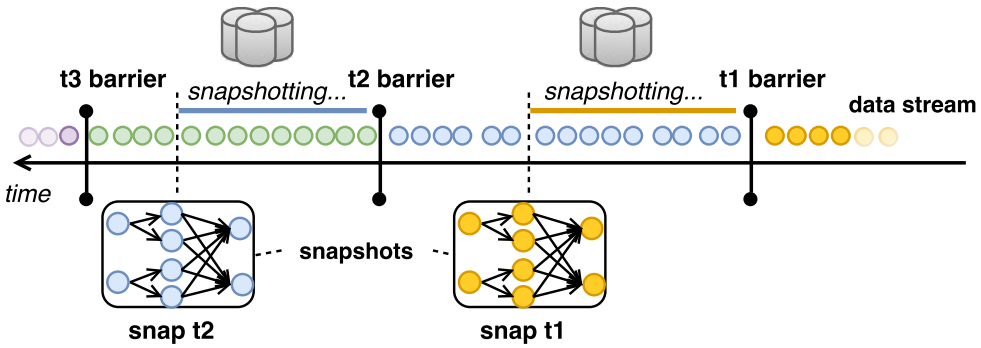


Figure 5: Asynchronous Barrier Snapshotting.

data records and events within a stream partition. The receiving operators react to these events by performing certain actions upon their arrival. Flink uses lots of special types of control events, including:

- *checkpoint barriers* that coordinate checkpoints by dividing the stream into pre-checkpoint and post-checkpoint (discussed in Section 3.3),
- *watermarks* signaling the progress of event-time within a stream partition (discussed in Section 4.1),
- *iteration barriers* signaling that a stream partition has reached the end of a superstep, in Bulk/Stale-Synchronous-Parallel iterative algorithms on top of cyclic dataflows (discussed in Section 5.3).

As mentioned above, control events assume that a stream partition preserves the order of records. To this end, unary operators in Flink that consume a single stream partition, *guarantee a FIFO order of records*. However, operators receiving more than one stream partition merge the streams in arrival order, in order to keep up with the streams' rates and avoid back pressure. As a result, streaming dataflows in Flink do not provide ordering guarantees after any form of repartitioning or broadcasting and the responsibility of dealing with out-of-order records is left to the operator implementation. We found that this arrangement gives the most efficient design, as most operators do not require deterministic order (e.g., hash-joins, maps), and operators that need to compensate for out-of-order arrivals, such as event-time windows can do that more efficiently as part of the operator logic.

3.3 Fault Tolerance

Flink offers reliable execution with strict exactly-once-processing consistency guarantees and deals with failures via checkpointing and partial re-execution. The general assumption the system makes to effectively provide these guarantees is that the data sources are persistent and replayable. Examples of such sources are files and durable message queues (e.g., Apache Kafka). In practice, non-persistent sources can also be incorporated by keeping a write-ahead log within the state of the source operators.

The checkpointing mechanism of Apache Flink builds on the notion of distributed consistent snapshots to achieve exactly-once-processing guarantees. The possibly unbounded nature of a data stream makes re-computation upon recovery impractical, as possibly months of computation will need to be replayed for a long-running job. To bound recovery time, Flink takes a snapshot of the state of operators, including the current position of the input streams at regular intervals.

The core challenge lies in taking a consistent snapshot of all parallel operators without halting the execution of the topology. In essence, the snapshot of all operators should refer to the same logical time in the computation. The mechanism used in Flink is called Asynchronous Barrier Snapshotting (ABS [7]). Barriers are control records injected into the input streams that correspond to a logical time and logically separate the stream to the part whose effects will be included in the current snapshot and the part that will be snapshotted later.

An operator receives barriers from upstream and first performs an alignment phase, making sure that the barriers from all inputs have been received. Then, the operator writes its state (e.g., contents of a sliding window, or custom data structures) to durable storage (e.g., the storage backend can be an external system such as HDFS). Once the state has been backed up, the operator forwards the barrier downstream. Eventually, all operators will

register a snapshot of their state and a global snapshot will be complete. For example, in Figure 5 we show that *snapshot t2* contains all operator states that are the result of consuming all records before *t2 barrier*. ABS bears resemblances to the Chandy-Lamport algorithm for asynchronous distributed snapshots [11]. However, because of the DAG structure of a Flink program, ABS does not need to checkpoint in-flight records, but solely relies on the aligning phase to apply all their effects to the operator states. This guarantees that the data that needs to be written to reliable storage is kept to the theoretical minimum (i.e., only the current state of the operators).

Recovery from failures reverts all operator states to their respective states taken from the last successful snapshot and restarts the input streams starting from the latest barrier for which there is a snapshot. The maximum amount of re-computation needed upon recovery is limited to the amount of input records between two consecutive barriers. Furthermore, partial recovery of a failed subtask is possible by additionally replaying unprocessed records buffered at the immediate upstream subtasks [7].

ABS provides several benefits: i) it guarantees exactly-once state updates without ever pausing the computation ii) it is completely decoupled from other forms of control messages, (e.g., by events that trigger the computation of windows and thereby do not restrict the windowing mechanism to multiples of the checkpoint interval) and iii) it is completely decoupled from the mechanism used for reliable storage, allowing state to be backed up to file systems, databases, etc., depending on the larger environment in which Flink is used.

3.4 Iterative Dataflows

Incremental processing and iterations are crucial for applications, such as graph processing and machine learning. Support for iterations in data-parallel processing platforms typically relies on submitting a new job for each iteration or by adding additional nodes to a running DAG [6, 25] or feedback edges [23]. Iterations in Flink are implemented as *iteration steps*, special operators that themselves can contain an execution graph (Figure 6). To maintain the DAG-based runtime and scheduler, Flink allows for iteration "head" and "tail" tasks that are *implicitly* connected with feedback edges. The role of these tasks is to establish an active feedback channel to the iteration step and provide coordination for processing data records in transit within this feedback channel. Coordination is needed for implementing any type of structured parallel iteration model, such as the Bulk Synchronous Parallel (BSP) model and is implemented using control event. We explain how iterations are implemented in the DataStream and DataSet APIs in Section 4.4 and Section 5.3, respectively.

4 Stream Analytics on Top of Dataflows

Flink's DataStream API implements a full stream-analytics framework on top of Flink's runtime, including the mechanisms to manage time such as out-of-order event processing, defining windows, and maintaining and updating user-defined state. The streaming API is based on the notion of a DataStream, a (possibly unbounded) immutable collection of elements of a given type. Since Flink's runtime already supports pipelined data transfers, continuous stateful operators, and a fault-tolerance mechanism for consistent state updates, overlaying a stream processor on top of it essentially boils down to implementing a windowing system and a state interface. As noted, these are invisible to the runtime, which sees windows as just an implementation of stateful operators.

4.1 The Notion of Time

Flink distinguishes between two notions of time: i) event-time, which denotes the time when an event originates (e.g., the timestamp associated with a signal arising from a sensor, such as a mobile device) and ii) processing-time, which is the wall-clock time of the machine that is processing the data.

In distributed systems there is an arbitrary skew between event-time and processing-time [3]. This skew may mean arbitrary delays for getting an answer based on event-time semantics. To avoid arbitrary delays, these systems regularly insert special events called *low watermarks* that mark a global progress measure. In the case of time progress for example, a watermark includes a time attribute t indicating that all events lower than t have

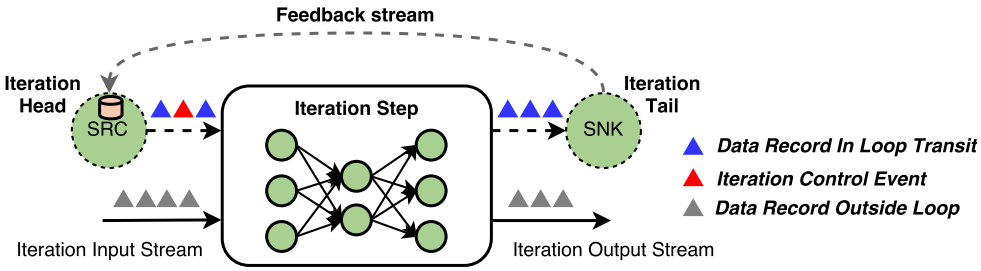


Figure 6: The iteration model of Apache Flink.

already entered an operator. The watermarks aid the execution engine in processing events in the correct event order and serialize operations, such as window computations via a unified measure of progress.

Watermarks originate at the sources of a topology, where we can determine the time inherent in future elements. The watermarks propagate from the sources throughout the other operators of the data flow. Operators decide how they react to watermarks. Simple operations, such as map or filter just forward the watermarks they receive, while more complex operators that do calculations based on watermarks (e.g., event-time windows) first compute results triggered by a watermark and then forward it. If an operation has more than one input, the system only forwards the minimum of the incoming watermarks to the operator thereby ensuring correct results.

Flink programs that are based on processing-time rely on local machine clocks, and hence possess a less reliable notion of time, which can lead to inconsistent replays upon recovery. However, they exhibit lower latency. Programs that are based on event-time provide the most reliable semantics, but may exhibit latency due to event-time-processing-time lag. Flink includes a third notion of time as a special case of event-time called *ingestion-time*, which is the time that events enter Flink. That achieves a lower processing latency than event-time and leads to more accurate results in comparison to processing-time.

4.2 Stateful Stream Processing

While most operators in Flink's DataStream API look like functional, side-effect-free operators, they provide support for efficient stateful computations. State is critical to many applications, such as machine-learning model building, graph analysis, user session handling, and window aggregations. There is a plethora of different types of states depending on the use case. For example, the state can be something as simple as a counter or a sum or more complex, such as a classification tree or a large sparse matrix often used in machine-learning applications. Stream windows are stateful operators that assign records to continuously updated buckets kept in memory as part of the operator state.

In Flink state is made explicit and is incorporated in the API by providing: i) operator interfaces or annotations to statically register explicit local variables within the scope of an operator and ii) an operator-state abstraction for declaring partitioned key-value states and their associated operations. Users can also configure how the state is stored and checkpointed using the StateBackend abstractions provided by the system, thereby allowing highly flexible custom state management in streaming applications. Flink's checkpointing mechanism (discussed in Section 3.3) guarantees that any registered state is durable with exactly-once update semantics.

4.3 Stream Windows

Incremental computations over unbounded streams are often evaluated over continuously evolving logical views, called windows. Apache Flink incorporates windowing within a stateful operator that is configured via a flexible declaration composed out of three core functions: a window *assigner* and optionally a *trigger* and an *evictor*. All three functions can be selected among a pool of common predefined implementations (e.g., sliding time windows) or can be explicitly defined by the user (i.e., user-defined functions).

More specifically, the assigner is responsible for assigning each record to logical windows. For example, this decision can be based on the timestamp of a record when it comes to event-time windows. Note that in the case of sliding windows, an element can belong to multiple logical windows. An optional trigger defines

when the operation associated with the window definition is performed. Finally, an optional evictor determines which records to retain within each window. Flink's window assignment process is uniquely capable of covering all known window types such as periodic time- and count-windows, punctuation, landmark, session and delta windows. Note that Flink's windowing capabilities incorporate out-of-order processing seamlessly, similarly to Google Cloud Dataflow [3] and, in principle, subsume these windowing models. For example, below is a window definition with a range of 6 seconds that slides every 2 seconds (the assigner). The window results are computed once the watermark passes the end of the window (the trigger).

```
stream
  .window(SlidingTimeWindows.of(Time.of(6, SECONDS), Time.of(2, SECONDS))
  .trigger(EventTimeTrigger.create())
```

A global window creates a single logical group. The following example defines a global window (i.e., the assigner) that invokes the operation on every 1000 events (i.e., the trigger) while keeping the last 100 elements (i.e., the evictor).

```
stream
  .window(GlobalWindow.create())
  .trigger(Count.of(1000))
  .evict(Count.of(100))
```

Note that if the stream above is partitioned on a key before windowing, the window operation above is *local* and thus does not require coordination between workers. This mechanism can be used to implement a wide variety of windowing functionality [3].

4.4 Asynchronous Stream Iterations

Loops in streams are essential for several applications, such as incrementally building and training machine learning models, reinforcement learning and graph approximations [9, 15]. In most such cases, feedback loops need no coordination. Asynchronous iterations cover the communication needs for streaming applications and differ from parallel optimisation problems that are based on structured iterations on finite data. As presented in Section 3.4 and Figure 6, the execution model of Apache Flink already covers asynchronous iterations, when no iteration control mechanism is enabled. In addition, to comply with fault-tolerance guarantees, feedback streams are treated as operator state within the implicit-iteration head operator and are part of a global snapshot [7]. The DataStream API allows for an explicit definition of feedback streams and can trivially subsume support for structured loops over streams [23] as well as progress tracking [9].

5 Batch Analytics on Top of Dataflows

A bounded data set is a special case of an unbounded data stream. Thus, a streaming program that inserts all of its input data in a window can form a batch program and batch processing should be fully covered by Flink's features that were presented above. However, i) the syntax (i.e., the API for batch computation) can be simplified (e.g., there is no need for artificial global window definitions) and ii) programs that process bounded data sets are amenable to additional optimizations, more efficient book-keeping for fault-tolerance, and staged scheduling.

Flink approaches batch processing as follows:

- Batch computations are executed by the same runtime as streaming computations. The runtime executable may be parameterized with blocked data streams to break up large computations into isolated stages that are scheduled successively.
- Periodic snapshotting is turned off when its overhead is high. Instead, fault recovery can be achieved by replaying the lost stream partitions from the latest materialized intermediate stream (possibly the source).
- Blocking operators (e.g., sorts) are simply operator implementations that happen to block until they have consumed their entire input. The runtime is not aware of whether an operator is blocking or not. These

operators use managed memory provided by Flink (either on or off the JVM heap) and can spill to disk if their inputs exceed their memory bounds.
- A dedicated DataSet API provides familiar abstractions for batch computations, namely a bounded fault-tolerant DataSet data structure and transformations on DataSets (e.g., joins, aggregations, iterations).
- A query optimization layer transforms a DataSet program into an efficient executable.

Below we describe these aspects in greater detail.

5.1 Query Optimization

Flink's optimizer builds on techniques from parallel database systems such as plan equivalence, cost modeling and interesting-property propagation. However, the arbitrary UDF-heavy DAGs that make up Flink's dataflow programs do not allow a traditional optimizer to employ database techniques out of the box [17], since the operators hide their semantics from the optimizer. For the same reason, cardinality and cost-estimation methods are equally difficult to employ. Flink's runtime supports various execution strategies including repartition and broadcast data transfer, as well as sort-based grouping and sort- and hash-based join implementations. Flink's optimizer enumerates different physical plans based on the concept of interesting properties propagation [26], using a cost-based approach to choose among multiple physical plans. The cost includes network and disk I/O as well as CPU cost. To overcome the cardinality estimation issues in the presence of UDFs, Flink's optimizer can use hints that are provided by the programmer.

5.2 Memory Management

Building on database technology, Flink serializes data into memory segments, instead of allocating objects in the JVMs heap to represent buffered in-flight data records. Operations such as sorting and joining operate as much as possible on the binary data directly, keeping the serialization and deserialization overhead at a minimum and partially spilling data to disk when needed. To handle arbitrary objects, Flink uses type inference and custom serialization mechanisms. By keeping the data processing on binary representation and off-heap, Flink manages to reduce the garbage collection overhead, and use cache-efficient and robust algorithms that scale gracefully under memory pressure.

5.3 Batch Iterations

Iterative graph analytics, parallel gradient descent and optimisation techniques have been implemented in the past on top of Bulk Synchronous Parallel (BSP) and Stale Synchronous Parallel (SSP) models, among others. Flink's execution model allows for any type of structured iteration logic to be implemented on top, by using iteration-control events. For instance, in the case of a BSP execution, iteration-control events mark the beginning and the end of supersteps in an iterative computation. Finally, Flink introduces further novel optimisation techniques such as the concept of *delta* iterations [14], which can exploit sparse computational dependencies Delta iterations are already exploited by Gelly, Flink's Graph API.

6 Related work

Today, there is a wealth of engines for distributed batch and stream analytical processing. We categorise the main systems below.

Batch Processing. Apache Hadoop is one of the most popular open-source systems for large-scale data analysis that is based on the MapReduce paradigm [12]. Dryad [18] introduced embedded user-defined functions in general DAG-based dataflows and was enriched by SCOPE [26], which a language and an SQL optimizer on top of it. Apache Tez [24] can be seen as an open source implementation of the ideas proposed in Dryad. MPP databases [13], and recent open-source implementations like Apache Drill and Impala [19], restrict their API to SQL variants. Similar to Flink, Apache Spark [25] is a data-processing framework that implements a DAG-based execution engine, provides an SQL optimizer, performs driver-based iterations, and treats unbounded

computation as micro-batches. In contrast, Flink is the only system that incorporates i) a distributed dataflow runtime that exploits pipelined streaming execution for batch and stream workloads, ii) exactly-once state consistency through lightweight checkpointing, iii) native iterative processing, iv) sophisticated window semantics, supporting out-of-order processing.

Stream Processing. There is a wealth of prior work on academic and commercial stream processing systems, such as SEEP, Naiad, Microsoft StreamInsight, and IBM Streams. Many of these systems are based on research in the database community [1, 5, 8, 10, 16, 22, 23]. Most of the above systems are either i) academic prototypes, ii) closed-source commercial products, or iii) do not scale the computation horizontally on clusters of commodity servers. More recent approaches in data streaming enable horizontal scalability and compositional dataflow operators with weaker state consistency guarantees (e.g., at-least-once processing in Apache Storm and Samza). Notably, concepts such as "out of order processing" (OOP) [20] gained significant attraction and were adopted by MillWheel [2], Google's internal version of the later offered commercial executor of Apache Beam/Google Dataflow [3]. Millwheel served as a proof of concept for exactly-once low latency stream processing and OOP, thus, being very influential to the evolution of Flink. To the best of our knowledge, Flink is the only open-source project that: i) supports event time and out-of-order event processing ii) provides consistent managed state with exactly-once guarantees iii) achieves high throughput and low latency, serving both batch and streaming

7 Acknowledgements

The development of the Apache Flink project is overseen by a self-selected team of active contributors to the project. A Project Management Committee (PMC) guides the project's ongoing operations, including community development and product releases. At the current time of writing this, the list of Flink committers are : Márton Balassi, Paris Carbone, Ufuk Celebi, Stephan Ewen, Gyula Fóra, Alan Gates, Greg Hogan, Fabian Hueske, Vasia Kalavri, Aljoscha Krettek, ChengXiang Li, Andra Lungu, Robert Metzger, Maximilian Michels, Chiwan Park, Till Rohrmann, Henry Saputra, Matthias J. Sax, Sebastian Schelter, Kostas Tzoumas, Timo Walther and Daniel Warneke. In addition to these individuals, we want to acknowledge the broader Flink community of more than 180 contributors.

8 Conclusion

In this paper, we presented Apache Flink, a platform that implements a universal dataflow engine designed to perform both stream and batch analytics. Flink's dataflow engine treats operator state and logical intermediate results as first-class citizens and is used by both the batch and a data stream APIs with different parameters. The streaming API that is built on top of Flink's streaming dataflow engine provides the means to keep recoverable state and to partition, transform, and aggregate data stream windows. While batch computations are, in theory, a special case of a streaming computations, Flink treats them specially, by optimizing their execution using a query optimizer and by implementing blocking operators that gracefully spill to disk in the absence of memory.

References

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, et al. The design of the Borealis stream processing engine. *CIDR*, 2005.

[2] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. Millwheel: fault-tolerant stream processing at Internet scale. *PVLDB*, 2013.

[3] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *PVLDB*, 2015.

[4] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinlaender, M. J. Sax, S. Schelter, M. Hoeger, K. Tzoumas, and D. Warneke. The stratosphere platform for big data analytics. *VLDB Journal*, 2014.

[5] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. Stream: The stanford data stream management system. *Technical Report*, 2004.

[6] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: Efficient Iterative Data Processing on Large Clusters. *PVLDB*, 2010.

[7] P. Carbone, G. Fóra, S. Ewen, S. Haridi, and K. Tzoumas. Lightweight asynchronous snapshots for distributed dataflows. *arXiv:1506.08603*, 2015.

[8] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, D. Fisher, J. C. Platt, J. F. Terwilliger, and J. Wernsing. Trill: a high-performance incremental query processor for diverse analytics. *PVLDB*, 2014.

[9] B. Chandramouli, J. Goldstein, and D. Maier. On-the-fly progress detection in iterative stream queries. *PVLDB*, 2009.

[10] S. Chandrasekaran and M. J. Franklin. Psoup: a system for streaming queries over streaming data. *VLDB Journal*, 2003.

[11] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM TOCS*, 1985.

[12] J. Dean et al. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 2008.

[13] D. J. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H.-I. Hsiao, R. Rasmussen, et al. The gamma database machine project. *IEEE TKDE*, 1990.

[14] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl. Spinning Fast Iterative Data Flows. *PVLDB*, 2012.

[15] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. On graph problems in a semi-streaming model. *Theoretical Computer Science*, 2005.

[16] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. Spade: the system s declarative stream processing engine. *ACM SIGMOD*, 2008.

[17] F. Hueske, M. Peters, M. J. Sax, A. Rheinländer, R. Bergmann, A. Krettek, and K. Tzoumas. Opening the Black Boxes in Data Flow Optimization. *PVLDB*, 2012.

[18] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *ACM SIGOPS*, 2007.

[19] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, et al. Impala: A modern, open-source sql engine for hadoop. *CIDR*, 2015.

[20] 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*, 2008.

[21] N. Marz and J. Warren. *Big Data: Principles and best practices of scalable realtime data systems*. Manning Publications Co., 2015.

[22] M. Migliavacca, D. Eyers, J. Bacon, Y. Papagiannis, B. Shand, and P. Pietzuch. Seep: scalable and elastic event processing. *ACM Middleware'10 Posters and Demos Track*, 2010.

[23] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. *ACM SOSP*, 2013.

[24] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino. Apache tez: A unifying framework for modeling and building data processing applications. *ACM SIGMOD*, 2015.

[25] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. *USENIX HotCloud*, 2010.

[26] J. Zhou, P.-A. Larson, and R. Chaiken. Incorporating partitioning and parallel plans into the scope optimizer. *IEEE ICDE*, 2010.

CSA: Streaming Engine for Internet of Things

Zhitao Shen*, Vikram Kumaran*, Michael J. Franklin†, Sailesh Krishnamurthy‡, Amit Bhat*, Madhu Kumar*, Robert Lerche* and Kim Macpherson*

*Cisco Systems, Inc
{zhitshen,vkumaran,amibhat,madhuku,rlerche,kimacphe}@cisco.com
†University of California, Berkeley
franklin@cs.berkeley.edu
‡Amazon Web Services, Inc.
sailesh@gmail.com

Abstract

The next generation Internet will contain a multitude of geographically distributed, connected devices continuously generating data streams, and will require new data processing architectures that can handle the challenges of heterogeneity, distribution, latency and bandwidth. Stream query processing is natural technology for use in IOT applications, and embedding such processing in the network enables processing to be placed closer to the sources of data in widely distributed environments. We propose such a distributed architecture for Internet of Things (IoT) applications based on Cisco's Connected Streaming Analytics platform (CSA). In this paper describe this architecture and explain in detail how the capabilities built in the platform address real world IoT analytics challenges.

1 Introduction

By some estimates the number of connected devices will approach 50 Billion by 2020 [1]. The *Internet of Things* (IoT), driven by the explosion in number of end points that will join the Internet, has become a popular movement in the industry today. Many recent papers have outlined the challenges introduced by the IoT (e.g,, [2, 3, 4]). In this paper, we focus on the challenges related to data handling and processing in such an environment. The amount of data generated scales with the number of devices, leading to potentially huge data volumes. Current elastic cloud capabilities give us the ability to store and process large volumes of data, but given the rate, scale and distribution of data generated by IoT devices, processing all the data in the cloud might might not be feasible. Fortunately, however, not every sensor reading is equally important and by processing data near the point of generation, it is possible to make intelligent trade-offs among data fidelity, latency, bandwidth and resources.

For example, in offshore oil fields the volume of data generated typically exceeds the bandwidth available [5]. Intelligent data reduction near the source can solve this problem with minimal loss of information. In other industries such as manufacturing and transportation, where the devices connected to the network are in rapid motion through space, any data generated needs to be analyzed in context with minimum latency to be useful [6].

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

In such situations there is very little leeway in where data is processed. Shipping data back and forth to a central cloud infrastructure over the wide area network is unacceptable to due the challenges of latency and unreliable communication links.

Another challenge posed by the IoT is data privacy, with the need for policy-based restrictions on what gets sent out from devices [2]. Finally, devices connected to the Internet display tremendous heterogeneity in communication protocols, formats and content. To deal with this variety in data sources, one needs intelligence near the data source to translate into a common representation for the system as a whole to be able to work together [2]. The challenges described above are present in most real-life IoT deployments and need to be considered for any successful solution.

Given that IoT devices typically generate streams of data, the ability to process those streams and to be able to correlate and join heterogeneous data streams as they are generated are critical capabilities. The devices generating data out in the field connect to a network gateway. A stream-processing engine, present at that edge gateway and embedded in the network in a high fan-in system [7], is an effective architecture for supporting these applications. Cisco's Connected Streaming Analytics (CSA) provides an embeddable platform capable of processing individual streams as well as stream-stream correlations and joins. It also supports tracking of numerous independent, concurrent, data sessions, making it an ideal platform for an IoT analytics architecture.

There are many use cases across multiple verticals that highlight how stream processing can address real-world IoT challenges.

- The oil and gas industry is increasingly being digitized, with sensors measuring the state of the entire operation around the clock. However operations are typically in remote areas that have poor connectivity, having limited bandwidth and relatively unreliable networks [8, 9]. The volume of data generated by an oil and gas operation runs into gigabytes per second, and it is a losing proposition to move the raw data into a traditional data store. Stream processing helps with intelligent data reduction at the network edge by picking salient features and sending only necessary data for central processing.

- Communications network operators are increasingly reliant on the analysis of real-time network telemetry for providing a disruption-free network. Traditional big data approaches focused scalability and are driven by data volume. However, in a network, the limiting factor is data movement, as using the network to move telemetry puts tremendous strain on its core function, namely, transmitting user data. A distributed analytics solution with in-stream analysis performed at the data sources, embedded on network devices, alleviates this problem [10].

- Another industry being transformed by IoT is manufacturing. Robots and machines, building products we use everyday, are increasingly being instrumented with sensors that continuously measure operation parameters. Distributed control is not a new concept in manufacturing, however current distributed control systems are proprietary boxes with many I/O control points [11]. In the new world of IoT, sensors and actuators are constantly being added and a truly distributed control system needs to be a platform that can incrementally grow in capability and capacity. A stream-processing platform that can run at the edge on gateway routers and switches connecting robots and machines can provide a low-latency open platform on which to build incremental analytics as new data sources and algorithms are developed.

In the industry today many of the architectures proposed to handle the challenges created by an Internet of Things are based on an assumption that all the data will reach and reside in the cloud [4, 12]. The elastic scalability of cloud infrastructure is an attractive solution to the challenge of unprecedented data volume from billions of sensors. We believe, however, that the world of IoT will evolve a very different architecture, primarily due to the challenges described in the previous paragraphs. IoT will not have the luxury of unlimited bandwidth, latency and connectivity in many real-life situations. Current proposed solutions treat software at the edges of the network as simple data accumulators with the primary purpose of shipping data to a central data center for

off-line analysis and human consumption. While there are parts of use cases where that assumption might apply, we strongly believe that across various industry verticals data needs to be processed at the right level of context. In other words the intelligence needed to analyze and process data needs to exist throughout the network in a high fan-in system.

CSA is an advanced stream-processing engine based on the Truviso technology [13, 14] that has been extended with the capability to run embedded in network elements as well as in other parts of the network and the cloud. This approach enables an architecture where intelligence can be located across the network and placed as close to the data source as desired. This architecture is supported by several key features built into CSA:

- **Stream correlation using joins**. Streaming joins are very useful to correlate streams from both homogeneous and heterogeneous data sources. In many IoT cases, streaming joins can be performed at the network edge, as the data sources are mostly geographically correlated. One challenge for stream correlation is out-of-order data arrival due to the complexity of network environment in IoT and because devices may have different latencies for generating streams. The original Truviso system had limited join facilities but we further extend these to support two types of streaming joins (*best-effort* joins and *correlated joins*) to handle time-alignment differences between streams.

- **Session windows**. In CSA, we implement a new type of window operator to support session-based analysis. Sessionization is critical for IoT applications and can be used to correlate streams from homogeneous sources as well as to monitor complex events and on-going status over a single logical stream that is fed by multiple threads of data events coming from multiple sources.

- **Edge processing via containers**. In CSA, we create a low-footprint version of the stream-processing engine that has been ported to run in Cisco's routers and switches at the edge. CSA is built into available secondary compute resources [15] in a container, which enables analytics applications to run on routers and switches. Consequently, no additional hardware beyond the routers and switches is required to retrieve and process the streams from network-connected devices for edge analytics. One key benefit of edge analytics on network devices is that stream processing can scale with network size. The computational complexity for each edge node can be considered as bounded, as the number of devices connected in the sub-network is limited by capacity of the network gateway device.

- **Built-in time-series algorithms**. CSA provides additional machine learning algorithms that can operate over time-series streams for handling common IoT use cases. For example, we implemented an algorithm to discover periodic patterns over time-series data. Also, we can use an ARIMA (Autoregressive Integrated Moving Average) model for forecasting sensor values based on time-series streams.

In the remainder of this paper, we describe the overall architecture we have developed for edge analytics using CSA and focus on the new features listed above. Due to space constraints, however, we do not address time-series algorithms, which we plan to address in a later publication.

2 System Overview

2.1 Architecture

Figure 1 depicts A high-level overview of the architecture for distributed streaming intelligence in the network. The components of the architecture are deployed on a distributed infrastructure. At the edges of the network, the gateway routers and switches connect to sensors and devices that are the sources of data. The edge gateways have spare compute resources that can be used to run data processing applications in the containers. A little higher up in the stack are the fog nodes [16], having somewhat more compute and storage than the edge gateways. They

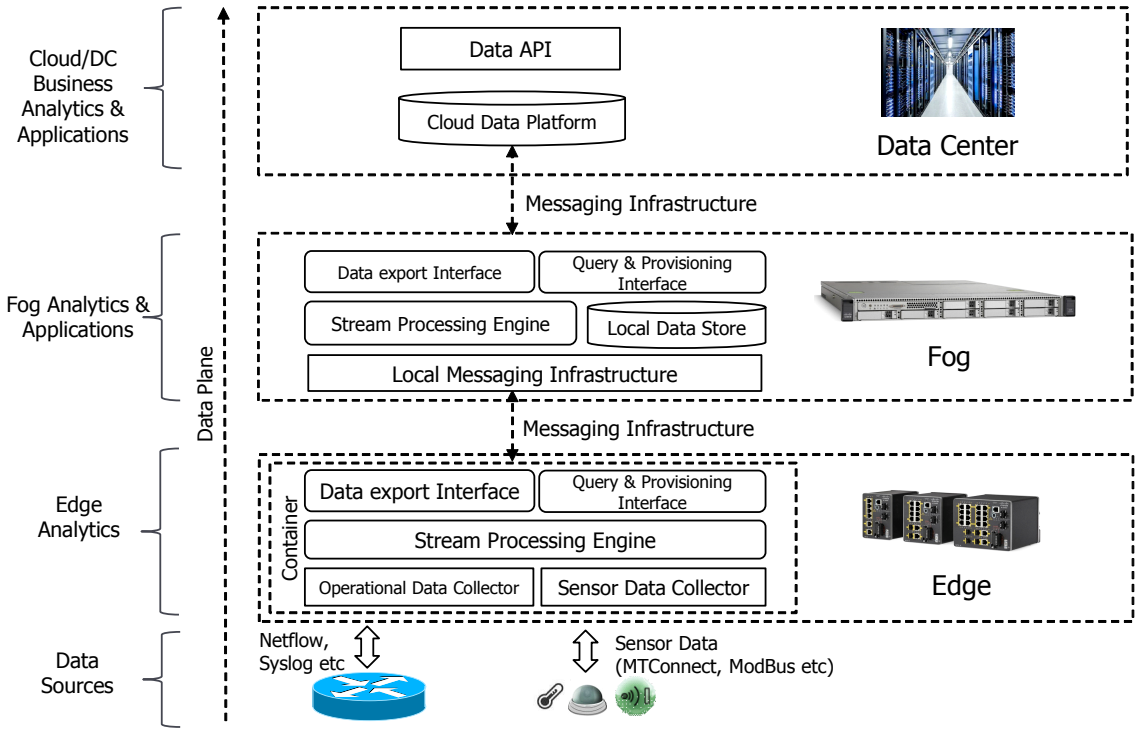


Figure 1: Edge Analytics Architecture

extend the cloud computing paradigm closer to the edge of the network. Further up the stack we get to the cloud or data center, where we have virtually unlimited scalability in terms of compute, storage and network. Cisco's CSA platform is software that can run on all the different levels in the hierarchy with appropriate resource-constrained capability. The main components of this architecture across the hierarchy are as follows:

Data collectors. The sensors and operational data are generated in a variety of protocols and content formats. There are no common standards for sensors and devices participating in the Internet of Things. This lacuna creates the need to have custom adapters to hide device heterogeneity and convert custom data streams into standard format. The *data collector* is a modular library of such adapters that will grow to handle the variations across the industry. The data collectors typically run on spare compute resources available in the edge gateways.

Stream processing engine. The data inputs transformed by the data collectors are processed by the *stream-processing engine*. When running at the edge, stream processing consists typically of simple aggregations, filtering, grouping, joins, local model scoring and prediction. As we move up the deployment hierarchy stream processing engines take on more complex computational tasks. Some of the key fundamental capabilities of the stream engine are discussed in detail in the later sections of this paper.

Query and provisioning interface. The interface remotely manages and monitors raw and derived streams in the engine. The platform is truly distributed and this interface provides a programming interface for remote administration.

Messaging infrastructure. Processing components of the architecture are sometimes distributed within a local area network and in many instances over a wide area network. In this architecture the stream processing engine described above exists in the data path. It consumes raw streams and emits processed data streams. The *messaging infrastructure* connects the various stream processing engines and provides an infrastructure to orchestrate the data flow.

Cloud data platform. In some applications the results of stream processing interact with devices and machines directly in a closed loop; for example, to change policy and influence actions. However, in most real world use cases, human intervention at a central location based on the visible state of the environment is still necessary. The *cloud data platform* provides the ability to combine real-time and historic data, operational data and business data for longer term visibility.

2.2 Streams and Queries

Connected Streaming Analytics (based on the Truviso engine) is designed to manage *streams* (i.e., unbounded, growing, data sets) in addition to relations (i.e., finite data sets) of the kind managed by a traditional RDBMS. CSA allows for these objects (i.e., streams and relations) to be created and queried in standard, full-featured SQL[1] of the sort supported by a typical RDBMS in an integrated fashion [13]. A SQL query that operates over one or more streams produces a continuous stream of results, and is therefore called a *continuous query*. The notion of continuous query is in contrast to a standard SQL query that operates exclusively over relations from a static view of a database and produces a finite data set (another relation) as its output. We refer to such a traditional query as a *static query*.

Streams. A continuous SQL query takes relations and streams as input, and produces streams as output. Unlike relations in a traditional database, a stream can be thought of as an unbounded bag of tuples, traveling though a network, where each tuple has a delineated timestamp attribute. A stream, like a table, is a database object that has an associated schema that defines the format of the data.

Raw and Derived Streams. In CSA, streams can be categorized as two different types depending how they are populated. *Raw streams* are populated by external data sources. A tuple in a raw stream can represent an event or state of the real world at a particular timestamp. *Derived streams* are defined using a continuous query on a raw stream or other derived streams, and populated by CSA.

Aggregation in CSA is computed in a shared fashion [17] and is therefore memory efficient. Additionally, CSA provides the capability of order-independent processing [14] and is useful for handling the out-of-order data appearing in real life IoT applications. In the reminder of this section, we briefly introduce the CSA query language.

Query Language. In CSA, queries can be posed exclusively on relations, exclusively on streams, or on a combination of streams and relations. Since a stream is unbounded, a streaming query that produces a stream never ends and, as stated above, is therefore called a *continuous query* (CQ). The only extension to the standard SQL syntax is a set of window (stream-to-relation) operators.

In order to process an unbounded stream of data, stream-processing engines apply windows that segment the stream into discrete finite data sets. CSA provides rich windowing semantics to support a variety of window definitions. For raw streams, windows may be either time-based (a specified interval of time, e.g. '1 minute') or row-based (a specified number of rows) depending on the need of the query. Derived streams, in addition to row and time windows, can define window-based windows, where the window size is specified as a number of windows in the underlying stream. Window based windows provide a level of abstraction, allowing the properties of a higher-level query to be specified in terms of the windows used by a lower-level query.

CSA offers a wide range of window (stream to relation) operators.

- **Chunking windows**: A *chunking window* is also known as a tumbling window. With chunking windows, the underlying stream is broken into successive, contiguous, and non-overlapping "chunks" of tuples.
- **Sliding windows**: A *sliding window* is expressed using an *advance* interval, and a *visible* interval. The former defines the periodic intervals (and thus the actual window edges) at which a new visible set is

[1]Note that only some of the non-monotonic SQL queries are supported in streaming fashion. For example, EXCEPT is not supported as a continuous query, while in most cases we can rewrite queries using a join operation if we only provide distinct tuples as results.

```
SELECT device_id, count(*) AS err_count
FROM message <SLICES '1 minute'>
WHERE type = 'ERROR'
GROUP BY device_id
ORDER BY err_count DESC
LIMIT 10
```

Figure 2: A Simple Continuous Query

constructed from the stream, while the latter defines the interval of tuples, relative to the periodic edges, that belong in each visible set. Note that both intervals can be either time-based or row-based intervals.

- **Landmark windows**: A *landmark window* is expressed using an advance interval, and a *reset* interval. The former defines the periodic interval ("advance" edges) at which a new visible set is constructed from the stream, while the latter defines a periodic interval that is used to compute a sequence of "reset" edges. Each visible set consists of all tuples that have arrived in the stream after the latest reset edge.
- **Session windows**: A *session window* correlates all tuples belonging to a given group whose time interval between consecutive tuples does not exceed a given timeout value. This window is useful to identify tuple sequences whose total duration is unknown in advance. The details of session windows and its application are discussed in Section 3.2.

Figure 2 shows a simple continuous query to find the top-10 devices with the most error messages in the past minute. <SLICES '1 minute'> defines a 1-minute chunking window and we conceptually transform all the messages in the past 1 minute into a relation via the window operation. Upon this resulting relation, the top-10 answers can be calculated by the standard SQL query using grouping and ordering.

2.3 Out-of-Order and Delayed Streams

Most stream-processing systems from both academia and industry assume that input streams arrive in order. This assumption is usually not true in real environments even for a single data source. For instance, data transportation with UDP packets may cause out-of-order delivery. In IoT environments, out of order data is the norm. One typical approach used to deal with this issue is to have the system rely on the physical order of streams. Tuples are timestamped using the clock time when they arrive. However, this approach is likely to produce incorrect results when trying to detect event sequences. Exact ordering is required for sequence matching.

Delayed streams are slightly different from out-of-order arrival. They can occur even if each independent data source is in order. The possible reasons are for delayed streams are: 1) the clocks of the local sources are not synchronized. 2) Network latencies are different from different sources to the engine. 3) The source device may encounter a delay while producing streaming data.

In CSA, we have two ways to handle correlating queries over out-of-order or delayed streams: 1) buffer-and-reorder mechanisms can reorder streams before feeding order-sensitive operators such as sessionization windows, for example, slack and drift [14] can be used to handle the streams with small degrees of out-of-orderness. 2) Coordinated joins can be used to correlate streams in a time-aligned fashion. We discuss these techniques in the following section.

3 Correlation, Sessionization and Joins

One of challenges for IoT analytics is the ability to correlate records from a single data source or multiple data sources. The typical streaming queries for correlating records are the join operators inherited from the

relational database world, and pattern matching usually used for complex-event processing [18]. In this section, we introduce how correlating queries are performed in CSA: streaming joins and the sessionization window operator supporting pattern matching as well as complex-event processing.

3.1 Streaming Joins

Streaming join is a fundamental operation for relating information from different streams. Over the last decade, a much previous work has focused on *sliding-window joins* [19, 20]. As streams are potentially unbounded, an obvious issue of un-windowed streaming joins is that the join state grows continuously and will eventually outgrow memory. Therefore, windows are usually applied to the input streams to restrict the scope of the join. Continuous Query Language (CQL for short) [21] specifies the semantics of a sliding-window streaming join by treating it as a view of a relational join over the sliding windows.

Consider the challenge for time alignment in IoT applications. In CSA, joins can be performed in two ways: *best-effort* and *coordinated*.

Best-effort Joins. In best-effort fashion, the join is processed immediately once a window is emitted when the end of the window (specified for example, in time or as a number of records) is reached. The window is joined against the most recent windows of the other join inputs. The idea behind the best-effort joins is, where possible, to generate the join results with minimum latency. Basically, best-effort can accept slightly out-of-order data and is useful when the skewness of the multi-source streams is low.

In CSA, if any of the inputs to a join is a row-based or window-based window, best-effort joins are performed. As each window of the join operand is received, the window is joined against the most recent windows of other input streams. Hence, the results of best-effort joins can be non-deterministic and will depend on the order in which the input streams' windows arrive.

```
SELECT
  s.device_id, s.torque
FROM
  sensor s <VISIBLE 1000 rows
            ADVANCE 1 row>,
  message m <SLICES 1 row>
WHERE
  s.device_id = m.device_id AND
  s.torque > 100 AND
  m.type='ERROR'
```

Figure 3: Example for Best-effort Joins

```
SELECT
 s.device_id, s.torque
DEOM
  sensor s <VISIBLE '1 minute'
            ADVANCE '1 second'>,
  message m <SLICES '1 second'>
WHERE
  s.device_id = e.device_id AND
  s.torque > 100 AND
  m.type = 'ERROR'
```

Figure 4: Example for Coordinated Joins

An example of best-effort join is shown in Figure 3. This join specifies that a tuple from the message stream joins with the last 1000 tuples from the sensor stream. As it is a best-effort join, the join results will output whenever a new record arrives form either stream. In this example, the join outputs a result whenever an error message occurs after an abnormal sensor reading (e.g., torque too high) is observed for the same device.

Coordinated Joins. In many Internet of Things applications, timestamps from the stream are generated by the edge device collecting or generating the data. However the stream-processing engine might not receive the tuples in the order of timestamps due to (for among other reasons) latency in the network. In such a case, correlating streams in a time-aligned fashion is important. To this end, the join usually is processed in a synchronized order from multiple stream sources. Unlike best-effort joins, time plays a very important role for coordinated joins. To perform coordinated joins, the join operator will ensure that when a window of one of the input streams arrives, it is joined against the latest possible windows of the other streams according to their respective timestamps. In CSA, if all the inputs to the streaming join are time-based and their timestamps are in

the same domain, coordinated joins are enabled.

An example of a coordinated join is shown in Figure 4. Similar to the best-effort join, this query shows how we join records from two streams. As both windows are time-based, the join is performed in a time-aligned fashion, that is, for a certain time t, the join will match the the records exactly between $t - 1$ *min* and t from the both streams based on the timestamps included in the data. Unlike best-effort joins, coordinated joins require buffering the tuples from the faster streams.

Coordinated joins are commonly used when one of the input streams is a derived stream with order-sensitive operators (e.g., aggregations), since the exact statistic or status for a certain time point is required.

3.2 Sessionization

The fixed-interval window operations (e.g., chunking, sliding, landmark) allow aggregates to be computed over data stream segments that are demarcated by a predetermined (user-specified) time interval or record count. While such windows enable many useful types of analytics, the rigidity of a given window size (time or row-based) can be too restrictive in situations where the segments of a data stream over which to run analytics are not known in advance. We developed techniques for *sessionization* to overcome such rigidity. Sessionization provides a way to operate on independent data event threads or sessions, each having its own independent window segments. We first describe the syntax of sessionization and then provide a simple example to introduce the main building blocks of CSA sessionization.

```
< SESSION session_key[, ...]
  TIMEOUT interval | NONE
 [EXPIRE WHEN conditions
             [RETAIN EDGE]]
 [ADVANCE interval
  OR
  ADVANCE WHEN conditions]
>
```

Figure 5: Syntax of Sessionization

```
SELECT
  device_id,
  FIRST(date_time) AS start_time,
  cq_close(*) AS check_time
FROM robot
  <SESSION device_id TIMEOUT NONE
   EXPIRE WHEN (FIRST(type) != 'START' OR
   LAST(type) != 'START')
   ADVANCE '1 second'>
GROUP BY device_id
HAVING
  cq_close(*)-first(date_time)>'10 minutes'
```

Figure 6: Example for Session Query

The details of the syntax of sessionization are shown in Figure 5. The session_key after the SESSION keyword specifies the keys for identifying the sessions. These keys should be same as those in the expressions for any GROUP BY clause in the stream queries and GROUP BY is mandatory when using session windows. Session windows are defined based on the semantic expressions rather than on fixed time intervals. Since sessions often do not have an explicit end record, the TIMEOUT clause specifies a timeout (expiring) interval for sessions that have no further associated tuples. If an EXPIRE WHEN condition is specified and is satisfied by the arrival of a tuple, then the session that the tuple belongs to is expired. If the optional RETAIN EDGE clause is specified, then after expiry, a new session is started with the current tuple as its first record. If an ADVANCE clause with a time interval is specified in a session definition, the session aggregation emits a result triggered by a time interval. For example ADVANCE '5 minutes' will cause the aggregation to emit a result every 5 minutes. If an ADVANCE WHEN condition is specified and is satisfied by the arrival of a tuple, then a result (projected in the SELECT clause of a stream definition) is emitted.

Consider the example of manufacturing in IoT. Suppose that we want to monitor a robot's status. When a robot is started, its START message will be sent out once. But sometimes the robot runs into a failure state and nothing is sent out after that. We need to detect such situation and reboot the device.

The session window definition in Figure 6 continuously computes sessions based on the individual device_id of a robot. The session starts only when we receive a START message and will be expired when we receive a non-START message[2]. According to the ADVANCE clause, the calculation occurs every second and only the sessions with duration longer than 10 minutes will be emitted, as specified in the HAVING clause.

There are several key features of CSA's sessionization:

- It enables precise metric computation over the sessions. It supports per-session expiry as well as result generation based on semantics that a user can specify as an aggregate (or even a complex combination of aggregates) as opposed to simply specifying rules based on the attribute(s) of a single tuple. Such semantics include not only CSA's built-in aggregates, but also any user-defined aggregates, including those that can do pattern-matching.
- Unlike the pattern-matching approach used in other stream-processing systems, the ADVANCE clause provides the ability to peek into ongoing activity for the sessions. For example, we can list all the ongoing sessions for each hour and compute an aggregation on top of it.
- Sessionization provides a TIMEOUT clause to expire the sessions which are not active for a certain period of time.
- Sessionization processing is memory-efficient, since we manage sessions within the shared aggregation infrastructure [17]. Also, we can avoid storing many of the tuples of a window in memory. Aggregate states are maintained for each session. The memory usage of sessionization depends on the number of concurrent sessions, not on their individual length.
- As each session maintains it own state, sessionization can easily scale out to multiple instances of the stream-processing engine for parallel computation. Key-based partitioning can be utilized to distribute data into multiple instances.

3.3 Applicability for the Internet of Things

In real world IoT analytics applications, we have to cope with challenges such as heterogeneous sources, differences in data formatting and temporal alignment of the streams. Joins and sessionization are useful for addressing these challenges posed by sensor data in an IoT deployment.

Integrating Homogeneous Data Sources Homogeneous data sources can be found in IoT deployments when similar devices and sensors are geographically collocated. Similar devices generate events with similar data schemas. In CSA, we suggest having single raw stream for homogeneous data schema from multiple sources. This greatly simplifies application of correlated queries, such as sessionization and joins.

Vertically Partitioned Data In many IoT protocol standards, the data is vertically partitioned. For example, a typical schema for the sensor stream includes {Timestamp, Type, Sub Type, Name, Id, Sequence, Value}. Self-joins on the stream are typically used to flatten attributes (correlate partitioned values) for the same device within a small time window.

Integrating Heterogeneous Data Sources Multiple data streams generated from a variety of devices and sensors are in many cases heterogeneous in their data schemas. A typical example is a machine that generates both sensor streams of physical measurements and event streams of state changes. For such situations, we can employ correlating queries on these streams, as we have shown in the preceding example. For heterogeneous data sources, separate raw streams are suggested. Streaming joins can be used to correlate the records from multiple streams.

[2]It is possible that only one START message is received. Usually heartbeats (punctuation) can be utilized to make the stream advance.

4 Edge Processing via Containers

Unlike traditional data-warehouse solutions where all data is collected and stored in a centralized place, the architecture we propose enables computation to be placed throughout the network, including at the edge. The CSA stream-processing engine is deployed on network-edge gateway routers and switches. Many of these edge gateways have spare compute and memory that can be exploited for non-network operations. The streaming engine is optimized for running in such constrained environments, and as majority of the processing is done in memory, there is very limited dependency on disk storage. In typical deployments, the CSA stream-processing engine runs inside a Linux container that is provided as a part of a Cisco edge gateway [15]. The container is hosted on a Cisco network device. Consequently, no additional hardware is required to retrieve and process the streams from network-connected devices for edge analytics.

There are several advantages to deploying the stream processing engine into a Linux container on a edge gateway: 1) The resources used by processes at the network edge are in a controlled space and the container reserves the essential resources for the network operation. 2) The application is isolated from the network OS which helps give security guarantees, 3) Linux containers are lightweight and fast for deployment. Running applications in the container is more efficient than running in a VM. Additionally, packaging CSA within a container image helps us to deploy applications into devices located in different layers of network (edge, fog and cloud) without much effort.

Another key benefit of edge processing on network devices is that streaming analytics can scale with network size. The number of devices connected in a sub-network can be considered bounded, since network devices normally have limited capacity and can only afford finite device connections. Therefore, we can consider the computational demand on each edge node to be bounded.

Besides efficiency in both network bandwidth and latency, edge processing is also very important for privacy. In many IoT applications such as Smart Cities, we are only allowed to bring processing to the streams and can expose only summaries or conclusions rather than raw data. Also, we can scrub and validate the data to be stored in data centers. For example, we can use CSA to anonymize sensitive personal information (information that can be used to identify a person, e.g., client MAC addresses) on the fly at the network edge, and expose only the data that is allowed to be stored in a data center according to local privacy laws.

5 Related Work

The Internet of Things and related applications have been widely studied [2, 3, 4]. However, streaming analytics in the context of Internet of Things is only starting to receive much attention. Aggarwal et al. [22] discuss how RDF streams can be handled with RDF queries and big-data facilities. Sheykh Esmaili [23] investigated event detection and FPGA implementation for embedded environments but the system described is not a fully functional stream-processing engine and is limited in its capabilities.

Earlier work on sliding-window joins[19, 20] does not consider time alignment and out-of-order events, which are widely observed in the real world. Li et al.[24] discuss out-of-order processing for stream joins. Recently, researchers from industry have been studying streaming joins in their respective contexts. Photon [25] from Google applies streaming joins to continuously combine a click event from a click log with its corresponding query event from a separate query log. Photon leverages distributed computing infrastructure from Google and joins are processed through different data centers. However, Photon is specifically designed for joining click and query streams and is not optimized for general streaming-join purposes. Also, sliding windows are not explicitly defined for joining. In CSA, we propose both coordinated joins and best-effort joins to cope with the challenges in correlating multiple data sources.

While sessionization was originally introduced for Web analysis [26], few implementations perform sessionization over streaming data. Akidau et al.[27] define session windows in a dataflow model. However, only

timeouts are provided for grouping tuples into sessions. Also related to sessionization are event pattern-matching and Complex Event Processing (CEP) for event streams. SASE [18] and Cayuga [28] are examples of systems supporting CEP over event streams. These systems usually provide a NFA-based pattern matching implementation. A key difference between these systems and CSA is that they treat event processing as distinct from traditional Relational query processing. In comparison, CSA is an extension of a traditional database system so it can leverage existing feature sets (e.g. user defined functions and database extension) in the Relational world and can easily combine streaming and static data. Sessionization in CSA is also efficient as we reuse an existing aggregation framework [17]. NiagaraST [29] proposed T+D frames which are similar to session windows, but did not specify a full-featured query language.

6 Conclusions

Modern Internet of Things applications are pushing traditional database and data warehousing technologies beyond their limits due to the explosive increase in data volumes, distributed data creation and requirements for low latency. To address these issues, we advocate an architecture deploying the Connected Streaming Analytics (CSA) engine, inside throughout the network, on edge gateways, fog nodes and on data center machines. This architecture enables a variety of new IoT applications.

CSA provides a query language for continuous queries over streams that supports various window operators, efficient shared aggregations, the functionality of an integrated relational database, out-of-order stream processing and correlation queries such as streaming joins and sessionization. In this paper, we showed how streaming joins and sessionization support correlating heterogeneous data sources from the Internet of Things. The features provided by CSA can solve important challenges in real world applications such as temporal alignment for heterogeneous sources. For network edge processing, we deploy CSA in a Linux container on network devices. The marriage of networking capabilites with stream query processing is unique and, we believe, can change how we analyze data created by connected things in the emerging world of IoT.

References

[1] "Connections counter: The Internet of Everything in motion." http://newsroom.cisco.com/feature-content?type=webcontent&articleId=1208342, 2013.

[2] L. Atzori, A. Iera, and G. Morabito, "The Internet of Things: A survey," *Computer Networks*, vol. 54, no. 15, pp. 2787–2805, 2010.

[3] C. Perera, A. Zaslavsky, P. Christen, and D. Georgakopoulos, "Context aware computing for the Internet of Things: A survey," *Communications Surveys & Tutorials, IEEE*, vol. 16, no. 1, pp. 414–454, 2014.

[4] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of Things (IoT): A vision, architectural elements, and future directions," *Future Generation Computer Systems*, vol. 29, no. 7, pp. 1645–1660, 2013.

[5] M. P. Mills, "Shale 2.0: Technology and the coming big-data revolution in america's shale oil fields." http://www.manhattan-institute.org/html/eper_16.htm#.VgFwvnvShHK, May 2015.

[6] A. Pye, "Mining's drive for efficiency," *Engineering & Technology*, vol. 10, no. 5, pp. 80–83, 2015.

[7] M. J. Franklin, S. R. Jeffery, S. Krishnamurthy, F. Reiss, S. Rizvi, E. Wu, O. Cooper, A. Edakkunni, and W. Hong, "Design considerations for high fan-in systems: The HiFi approach," in *CIDR*, pp. 290–304, 2005.

[8] D. Bigos, "5 ways IoT technologies are enabling the oil and gas industry." http://www.ibmbigdatahub.com/blog/5-ways-iot-technologies-are-enabling-oil-and-gas-industry, June 2015.

[9] "Creating digital oil fields and connected refineries." http://www.cisco.com/web/strategy/energy/external_oil.html", 2015.

[10] A. Clemm, M. Chandramouli, and S. Krishnamurthy, "DNA: An SDN framework for distributed network analytics," in *Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on*, pp. 9–17, IEEE, 2015.

[11] D. Brandl, "Distributed controls in the Internet of Things create control engineering resources."

http://www.controleng.com/single-article/distributed-controls-in-the-internet-of-things-create-control-engineering-resources/fb0eea6e0b9d0d8cd97aad4025e5c080.html, June 2014.

[12] A. Alamri, W. S. Ansari, M. M. Hassan, M. S. Hossain, A. Alelaiwi, and M. A. Hossain, "A survey on sensor-cloud: architecture, applications, and approaches," *International Journal of Distributed Sensor Networks*, vol. 2013, 2013.

[13] M. J. Franklin, S. Krishnamurthy, N. Conway, A. Li, A. Russakovsky, and N. Thombre, "Continuous analytics: Rethinking query processing in a network-effect world," in *CIDR*, www.cidrdb.org, 2009.

[14] S. Krishnamurthy, M. J. Franklin, J. Davis, D. Farina, P. Golovko, A. Li, and N. Thombre, "Continuous analytics over discontinuous streams," in *SIGMOD Conference*, pp. 1081–1092, ACM, 2010.

[15] P. Jensen, "Cisco fog computing solutions: Unleash the power of the Internet of Things." http://www.cisco.com/web/solutions/trends/iot/docs/computing-solutions.pdf, May 2015.

[16] F. Bonomi, R. Milito, P. Natarajan, and J. Zhu, "Fog computing: A platform for Internet of Things and analytics," in *Big Data and Internet of Things: A Roadmap for Smart Environments*, pp. 169–186, Springer, 2014.

[17] S. Krishnamurthy, C. Wu, and M. J. Franklin, "On-the-fly sharing for streamed aggregation," in *SIGMOD Conference* (S. Chaudhuri, V. Hristidis, and N. Polyzotis, eds.), pp. 623–634, ACM, 2006.

[18] D. Gyllstrom, E. W. 0002, H.-J. Chae, Y. Diao, P. Stahlberg, and G. Anderson, "SASE: Complex event processing over streams (demo)," in *CIDR*, pp. 407–411, www.cidrdb.org, 2007.

[19] L. Golab and M. T. Özsu, "Processing sliding window multi-joins in continuous queries over data streams," in *VLDB*, pp. 500–511, 2003.

[20] U. Srivastava and J. Widom, "Memory-limited execution of windowed stream joins," in *VLDB*, pp. 324–335, Morgan Kaufmann, 2004.

[21] A. Arasu, S. Babu, and J. Widom, "The CQL continuous query language: semantic foundations and query execution," *VLDB J*, vol. 15, no. 2, pp. 121–142, 2006.

[22] C. C. Aggarwal, N. Ashish, and A. P. Sheth, "The Internet of Things: A survey from the data-centric perspective," in *Managing and Mining Sensor Data*, pp. 383–428, Springer, 2013.

[23] K. Sheykh Esmaili, *Data stream processing in complex applications*. PhD thesis, ETH Zürich, 2011.

[24] 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.

[25] R. Ananthanarayanan, V. Basker, S. Das, A. Gupta, H. Jiang, T. Qiu, A. Reznichenko, D. Ryabkov, M. Singh, and S. Venkataraman, "Photon: fault-tolerant and scalable joining of continuous data streams," in *SIGMOD Conference*, pp. 577–588, 2013.

[26] D. Gayo-Avello, "A survey on session detection methods in query logs and a proposal for future evaluation," *Information Sciences*, vol. 179, pp. 1822–1843, May 2009.

[27] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle, "The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," *PVLDB*, vol. 8, no. 12, pp. 1792–1803, 2015.

[28] A. J. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. M. White, "Cayuga: A general purpose event monitoring system," in *CIDR*, pp. 412–422, www.cidrdb.org, 2007.

[29] D. Maier, M. Grossniklaus, S. Moorthy, and K. Tufte, "Capturing episodes: may the frame be with you," in *DEBS*, pp. 1–11, ACM, 2012.

Trill: Engineering a Library for Diverse Analytics

Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, James F. Terwilliger
Microsoft
{badrishc, jongold, mbarnett, jamest}@microsoft.com

Abstract

Trill is a streaming query processor that fulfills three requirements to serve the diverse big data analytics space: (1) Query Model: Trill is based on the tempo-relational model that enables it to handle streaming and relational queries with early results, across the latency spectrum from real-time to offline; (2) Fabric and Language Integration: Trill is architected as a high-level language library that supports rich data-types and user libraries, and integrates well with existing distribution fabrics and applications; and (3) Performance: Trill's throughput is high across the latency spectrum. For streaming data, Trill's throughput is 2-4 orders of magnitude higher than comparable traditional streaming engines. For offline relational queries, Trill's throughput is comparable to modern columnar database systems. Trill uses a streaming batched-columnar data representation with a new dynamic compilation-based system architecture that addresses all these requirements. Trill's ability to support diverse analytics has resulted in its adoption across many usage scenarios at Microsoft. In this article, we provide an overview of Trill: how we engineered it as a library that achieves seamless language integration with a rich query language at high performance, while executing in the context of a high-level programming language.

1 Introduction

Cloud applications accumulate data from a variety of data sources, such as machine telemetry and user-activity logs. This accumulation has resulted in an increasing need to derive value in an efficient and timely manner from such data. At Microsoft, we have seen a variety of cloud applications with a diverse range of analytics scenarios:

- An application may monitor telemetry (e.g., user clicks on advertisements or memory usage of a service) and raise alerts when problems are detected.
- An application may wish to correlate live data streams with historical activity (e.g., from one week back).
- Users may wish to develop the initial monitoring query using logs, before deploying it in a real-time system. Conversely, they may want to back-test their live monitoring queries over historical logs, perhaps with different parameters (in a *what-if* style of analysis).
- Analysts may want to run relational analyses (in the form of *business intelligence* queries) over historical logs. Further, they may prefer quick approximate results by streaming the data, as that better fits an exploratory environment.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

This diverse and interconnected nature of cloud analytics has resulted in an ecosystem of disparate tools, data formats, and techniques [13]. Combining these tools with application-specific glue logic is a tedious and error-prone process, with poor performance and the need for translation at each step. The lack of a unified data model across these scenarios precludes the ability to reuse logic, e.g., by developing queries on historical data and deploying them directly to live streams.

In order to alleviate the complexities outlined above, we built Trill [14], a single analytics engine that can serve a diverse analytics space. Trill simultaneously addresses three requirements present in the scenarios above:

- ***Query Model***: Trill is based on a unifying temporal data model based on application time, which enables the diverse spectrum of analytics described earlier: real-time, offline, temporal [7], relational, and progressive (approximate) [8] queries.
- ***Fabric & Language Integration***: Trill is written as a library in the *high-level-language* (HLL) C#, and thus benefits from arbitrary HLL data-types, a rich library ecosystem, integration with arbitrary program logic, ingesting data without "handing off" to a server or copying to native memory, and easy embedding within scale-out fabrics and as part of cloud application workflows.
- ***Performance***: Trill handles the entire space of analytics described earlier, at best-of-breed or better levels of performance. In Chandramouli et al. [14], we showed that Trill processes streaming temporal queries at rates that are 2 to 4 orders-of-magnitude higher than traditional streaming engines. Further, for the case of offline relational (non-temporal) queries over logs, Trill's query performance is comparable to modern columnar databases, while supporting a richer query model and language integration. Trill is very fast for simple payload types (common for early parts of a pipeline), and degrades gracefully as payloads become complex, such as machine learning models (common on reduced data).

Trill achieves these requirements using a hybrid system architecture that exposes a latency-throughput trade-off to users. Users specify a latency requirement, and Trill repacks streams into a sequence of batches with a goal of meeting the requirement. Unlike other batched streaming systems, such as Spark Streaming [21], our query model allows batching to be purely physical (not commingled with application time) and therefore easily variable: query results are always identical to the case of per-event processing, regardless of batch sizes or data-arrival rates. The user's query is converted into a directed acyclic graph of streaming operators that each receive and produce streams of data batches. Further, within each batch, Trill uses a columnar data organization when possible, along with new and highly efficient columnar streaming operators that work directly on the columnar batches. Engineering such a query processor as a high-level language library introduced several challenges; this article describes how we addressed these problems as we built a generally usable engine.

- Trill operators expect data to be batched in timestamp order for high performance. On the other hand, real-time data may arrive one event-at-a-time, and may have inherent disorder. Section 4 describes our data model that makes batching a purely physical construct, and our ingress-egress design that provides users with control for handling disorder and other requirements.
- Queries in Trill are language-integrated. Users expect a powerful query language capable of both relational-style operations and temporal manipulations such as data-dependent windowing, while staying in the context of a HLL and type system. Section 3 uses a running example to describe several key Trill language elements that enable expressive query specification seamlessly in a HLL.
- Section 4 covers our design of dynamic code generation to enable user-transparent columnar batched execution in a HLL. Further, it discusses Trill's threading choices and features such as checkpointing, which are necessary to use the engine in the context of a distributed fabric for resilient real-time processing.

We conclude the article in Section 5 with a brief overview of the ways Trill is used in practice, and some lessons learned from these scenarios.

Running Example

As our running example in this article, we consider an advertising platform that tracks advertisement (ad) impressions shown to users, and clicks on the ads. We can use a C# type to capture the event contents as below.

```
struct AdInfo {
   long Timestamp;          long UserId;
   long AdId;               bool IsClick;
}
```

Here, `IsClick` is a boolean value that denotes whether the event is a click (true) or an impression (false). We wish to ingest such a data stream arriving at Trill from diverse sources, execute a variety of temporal queries over the stream, and output results, for example, to a dashboard or console.

2 Trill Data Model, Ingress, and Egress

Logically, we view a stream as a temporal database (TDB) [12] that is presented incrementally [3, 11, 14]. Each event is associated with a data window (or interval of application time) that denotes its period of validity. This association creates a sequence of *snapshots* across time, where a snapshot at time t is a collection of events that are valid at time t. The user query is logically executed against these snapshots in an incremental manner.

2.1 Event Representation

Consider an event with a data window of $[s, e)$. This event may arrive directly as an interval, at application time s. We call s the *sync-time* of the interval event. Alternatively, the event may arrive broken up into a separate insert into the TDB (called a *start-edge*) at time s, optionally followed by a delete from the TDB (called an *end-edge*) at a later time, e. The start-edge and the end-edge have sync-times of s and e respectively. Sync-time is an important concept in Trill; it denotes the logical instant when a fact about the stream content becomes known. Events are always processed by Trill in strictly non-decreasing sync-time order (we discuss the handling of late-arriving events in Section 2.2.1). Because time in Trill is just a `long` (64-bit integer) type, we can, for example, re-interpret time to mean query progress when executing progressive relational queries [8].

`StreamEvent<T>` is a Trill struct that represents an event with payload type `T`, and includes static methods to create interval, start-edge, and end-edge events. We may also create a *point event*, an interval event with an data window of one chronon (the smallest unit of time). In our example, users may ingest clicks and impressions as point events: `StreamEvent<AdInfo>.CreatePoint(timestamp, new AdInfo { ... })`.

Further, users can ingest a special kind of event called a *punctuation*. A punctuation is associated with a timestamp t, and serves two purposes: (1) It denotes the passage of application time until t, in the absence of data, and allows operators to clean up system state; and (2) Each operator internally batches events (up to the maximum batch size) before sending the batch to the next operator. A punctuation enforces the immediate flushing of batches through Trill, to force processing and output generation until t.

2.2 Event Ingress

Data is available for querying in Trill by representing the source as an instance of a special generic interface that we call `IStreamable<T>`. This interface is Trill's variant of `IObservable<T>` [17], the standard .NET interface for pushing data. Briefly, `IObservable<T>` provides the ability for a data source to push objects of type `T` to a downstream observer `o` that "subscribes" to the observable via a `Subscribe(o)` call.

In our running example, we could create an `IObservable<StreamEvent<AdInfo>>` instance to push a sequence of individual events of type `StreamEvent<AdInfo>` to Trill as follows.

```
IObservable<StreamEvent<AdInfo>> o = Network.CreateObservable<AdInfo>(...);
```

Other ingress mechanisms supported in Trill include efficient bulk-ingress using a stream of arrays of events (`IObservable<ArraySegment<StreamEvent<T>>>`), pull-based sequences (e.g., `IEnumerable<T>`), and generic data-reader formats such as `IDataReader` [1].

We transform an instance of an input such as `IObservable<StreamEvent<T>>` into an `IStreamable<T>` using a special ingress method, defined on the `IObservable` instance, called `ToStreamable(...)`, whose parameters specify policies for ingesting data into Trill. These policies are described next.

2.2.1 Ingress Policies

When ingesting data into Trill from the outside world, we need to: (1) specify how to handle disorder in the stream; (2) automate the flushing of data into the system as (columnar) batches; and (3) specify system behavior when the input stream comes to an end. These transformations are driven by three user-defined policies that are provided as part of the `ToStreamable(...)` call:

- **Disorder**: Trill processes data in timestamp order for efficiency. We provide multiple ways of handling disorder, using a disorder policy. We support the policies of **adjust** (modify the late-arriving event to have the current sync-time as its timestamp), **drop** (drop the late event), and **throw** (throw an exception on encountering a late event). Further, each of these policies takes a *reorder latency* argument that is used to buffer and reorder late-arriving events within the provided reorder latency budget. Later-arriving events are handled using the specified policy of drop, adjust, or throw.
- **Flush**: The flush policy allows Trill to automate the injection of punctuation into the stream, in order to flush partially filled batches in the system. Supported policies include (1) **count**, which takes a parameter c and flushes the stream every c events; and (2) **time**, which takes a time duration argument d, and flushes the stream every d units of application time.
- **Completed**: When a stream completes, we can (1) halt the query without flushing partial batches in the system; (2) flush partial batches, but not force the current sync-time to move forward; or (3) move the current sync-time to ∞ (possibly producing new output) and flush the system.

In our running example, we could reorder late-arriving events within a timespan of `r` units (dropping later events), and issue flushes every 1000 events, while ingesting into Trill, as follows.

```
var s0 = o.ToStreamable(OnCompletedPolicy.EndOfStream(), DisorderPolicy.Drop(r),
                        PeriodicPunctuationPolicy.Count(1000));
```

2.3 Query Specification and Egress

An `IStreamable` instance such as `s0` is returned by the `ToStreamable(...)` call. Trill's query specification hangs off this instance in the form of functional method invocations. Each method returns a new `IStreamable` instance, allowing users to chain an entire query plan. We describe query specification in detail in Section 3. Note that query specification itself does not start query execution; this is done by subscribing to a Trill query using a variety of techniques. A common use case is to egress results as an observable sequence of `StreamEvent<T>` instances using a `ToStreamEventObservable(...)` method. We support an optional egress policy called `CoalesceEdges`: when set, this policy indicates that Trill will coalesce start-edge and end-edge pairs into intervals before outputting them. Since Trill emits events in sync-time order, this egress policy can incur latency because output has to be held back when we encounter a start-edge, until a matching end-edge is seen (in order to construct and emit the corresponding interval event). In our running example, we could output all the events to the console (as a pass-through) as follows.

```
s0.ToStreamEventObservable().Subscribe(e => Console.WriteLine(e.ToString()));
```

3 The Trill Query Language, by Example

Any values of type `IStreamable`, such as `s0`, are stream endpoints over which a Trill query can be written. Trill's query language, called Trill-LINQ, is modeled after LINQ [19], with temporal interpretation of the standard relational operations, along with new operations for temporal manipulation. In this section, we cover several language constructs in Trill using our running example.

3.1 Filtering and Projection

Assume that we want to consider only a 5% sample of users in the stream. We use the `Where` operator in Trill to filter the stream as follows:

```
var s1 = s0.Where(e => e.UserId % 100 < 5);
```

The expression in parentheses is called a lambda expression [10]; it is an *anonymous function*, in this case from the type `AdInfo` to a boolean value specifying for each row (event) `e` in the stream that it is to be kept in the output stream, `s1`, if its `UserId` modulo 100 is less than 5. Each Trill operator is a function from stream to stream which allows for easy functional composition of queries.

We can also transform the data to a different type, using the `Select` operator to perform a projection:

```
var s2 = s1.Select(e => e.AdId);
```

In this case, the lambda expression is a function from `AdInfo` to `long` indicating how the input payload type is transformed into a new output payload type by taking the result of the previous query, `s1`, and dropping the fields other than `AdId` to form a stream with exactly one field. Thus, stream `s2` has the type `IStreamable<long>`.

3.2 Windowing

Trill supports the notion of altering event lifetimes to support windowed operations and correlating data across time. In its most basic form, this is accomplished using the `AlterEventLifetime` operation. This operation accepts two expressions as input: a start-time selector which maps an interval's start-time to a new start-time, and a duration selector, which maps a start-time and end-time to a new duration. We limit timestamp modifications to those that preserve output sync-time order. Trill also provides macros that allow users to easily create hopping, tumbling, and sliding windows using `AlterEventLifetime` and its variants such as `AlterEventDuration`, which serves to alter an event's duration, leaving the start-time unmodified. For example, we can create a 5-minute tumbling window over the (sampled) stream `s1` as follows.

```
var s3 = s1.TumblingWindow(fiveMinutes);
```

3.3 Aggregation

Aggregation in Trill is done using an operator framework called *user-defined snapshot*, which enables the integration of custom incremental HLL logic into stream processing without sacrificing performance. It handles the class of operations that incrementally compute a result per time snapshot. In fact, all our built-in aggregates (including complex multi-valued aggregates such as top-k) are implemented using this general framework, described in Chandramouli et al. [14]. For example, we can compute a 5-minute tumbling window count of events using `s3`, as follows.

```
var s4 = s3.Aggregate(w => w.Count());
```

We also support the simultaneous application of multiple aggregates in a single snapshot operator, with the ability to combine results on a per-snapshot basis (see Chandramouli et al. [14] for details).

3.4 Grouped Computation

Trill supports a `GroupApply` operation, where the user specifies a grouping key selector and a sub-query. Logically, `GroupApply` executes the given sub-query on each sub-stream corresponding to each distinct key, as determined by the grouping key selector. For example, we could compute the five-minute tumbling window count on a per-ad basis as follows:

```
var s5 = s1.GroupApply(e => e.AdId,
                       s => s.TumblingWindow(fiveMinutes)
                             .Aggregate(w => w.Count()),
                       (g, p) => new { AdId = g, Count = p });
```

Here, the first lambda expression specifies the grouping key, and the second lambda expression specifies the query to be executed per key. The final lambda allows the user to combine the grouping key and the per-group payload into a single result payload.

3.5 Correlation and Set Difference

The temporal join operator in Trill allows one to correlate (or join) two streams based on time overlap, with an (optional) equality predicate on payloads. Suppose we wish to augment the filtered AdInfo stream `s1` with additional information from another *reference* stream `ref1` that contains per-user demographics data such as age. We would express such a query in Trill as follows:

```
var s6 = s1.Join(ref1, l => l.UserId, r => r.UserId,
                 (l, r) => new Result { l.AdId, l.UserId, r.Age });
```

The second and third parameters to `Join` represent the equi-join predicate on the left and right inputs (`UserId` in this case), while the final parameter is a lambda expression that specifies how matching input tuples (from the left and right) are combined to construct the result events of payload type `Result`, yields a stream of type `IStreamable<Result>`. As a more complex example, suppose we wish to join ad impressions to clicks on the same ad, and by the same user, within 10 minutes. This query is written as:

```
var s7 = s1.GroupApply(e => new { e.UserId, e.AdId },
                       s => s.Where(e => !e.IsClick)
                             .AlterEventDuration(tenMinutes)
                             .Join(str.Where(e => e.IsClick), (l, r) => r),
                       (g, p) => p);
```

Trill also support a temporal set difference operator called `WhereNotExists`. For instance, we can output all clicks that were not preceded by an impression within 10 minutes, as follows:

```
var s8 = s1.GroupApply(e => new { e.UserId, e.AdId },
                       s => s.Where(e => e.IsClick)
                             .WhereNotExists(str.Where(e => !e.IsClick)
                                                .AlterEventDuration(tenMinutes),
                                             (l, r) => r),
                       (g, p) => p);
```

3.6 Data-Dependent Windowing

Trill supports the creation of windows based on data. Such windows can, for instance, be used to create *session windows* that limit an event's influence to the end of the session. For example, suppose we wanted to take impressions and restrict their lifetime to be either 10 minutes or the first click after the impression, whichever comes earlier. We express this query using the `ClipEventDuration` operator, which *clips* the duration of an

event E to end at the start-time of the first matching event on the right-side input that falls within E's time interval.

```
var s9 = s1.GroupApply(e => new { e.UserId, e.AdId },
                       s => s.Where(e => !e.IsClick)
                             .AlterEventDuration(tenMinutes)
                             .ClipEventDuration(str.Where(e => e.IsClick),
                                                (l, r) => r),
                       (g, p) => p);
```

4 Internal Architecture

4.1 Batching with Columnar Organization

As mentioned earlier, we physically batch events before feeding them to Trill, based on the user-specified latency requirement. Batches allows system overhead to be amortized over many events. While batching is advantageous in its own right, it enables us to re-organize data within batches. We store batch content in columnar format. A *columnar batch* (referred to hereafter just as *batch*) is a structure of that holds one array for each column in the event. For example, one array holds the sync-time values for all events in the batch, while another array holds a second timestamp associated with events (called the *other-time*). Internally, every event is associated with a *grouping key* in order to enable efficient grouped operations. We precompute and store the grouping key (and its hash) as two additional arrays in the batch. We also include an *absentee bitvector* to identify which rows in the batch are currently active. The bitvector allows filter operations to logically remove rows without having to physically reorganize the batch. For instance, the `Where` query in Section 3 just sets the bit corresponding to each row for which the function returns false.

Being in a high-level language, we use the generic type system to get strong type safety for batches expressed over the two types `K` and `P` for the key type and payload type, respectively.

```
class Batch<K,P> {
   long[] SyncTime;        long[] OtherTime;
   K[] Key;                int[] Hash;
   P[] Payload;            long[] BitVector;
}
```

As in database systems, columnar representation results in better data locality, bringing much less data to the CPU. Further, we are able to use a custom memory allocation scheme for the arrays: for instance, the output batch of a selection operator does not modify the sync-time of each event and so can share a reference to that array with the input batch. We aggressively pool arrays using a global memory manager to alleviate the cost of memory allocation and garbage collection. In a streaming setting, the system quickly achieves a steady state with the memory allocated for output batches being reused for succeeding input batches.

Note that the payload of each event above remains a row structure. For instance, the example of Section 3 results in the `Payload` array being of type `AdInfo`[1]. This means that operators accessing very few fields of the payload may not enjoy the data locality that is provided by the columnar layout of the other fields. Each operator in Trill has an implementation that executes against this representation. We call them the *row-based* operators since the payloads exist as an individual instance per row.

[1] In .NET, since that type was defined as a `struct` — a value type — the array is physically laid out in memory as a contiguous sequence of bytes. However, if it were defined as a `class` — a reference type — then the array would be a contiguous sequence of pointers, with the storage for each instance individually allocated somewhere in the heap.

4.2 Code Generation

We can adopt a columnar data layout for payload fields as well, by allocating a separate array for each field in the payload. For the type `AdInfo`, we have three arrays of `long` and one array of `bool`:

```
class ColumnarBatchForAdInfo<K> : Batch<K, AdInfo> {
   // Other arrays inherited from Batch<>, Payload array ignored in base class
   long[] Timestamp;       long[] UserId;
   long[] AdId;            bool[] IsClick;
}
```

With this representation, an operator that accesses a single field will result in contiguous memory loads for that field alone. If a payload type cannot be made columnar (e.g., it is a class with private fields), we revert to the data format described in Section 4.1.

Note that there is an impedance mismatch between the user's view of the data — the type `AdInfo` available at compile-time — and the system's view — the type `ColumnarBatchForAdInfo` — which is not available at compile-time. Since queries and data are dynamic, i.e., a new query expressed over a new schema (payload type) is not predefined, the system must be able to create the generated types and operators that use those types during runtime. We solve this problem using *dynamic code generation* to create new type definitions, e.g., `ColumnarBatchForAdInfo` for batches, and optimized *columnar operators* that are aware of the columnar representation, and inline operations on the columnar format. Columnar organization also enables optimized serialization and string handling; see Chandramouli et al. [14] for details. These transformations are transparent to users, who continue to operate with their row-based data model. For example, the `Select` operator generated for the example in Section 3 computes the single payload column in each output batch in the stream `s2` in constant time; we simply copy the single pointer to the `AdId` column from the input batch of stream `s1`.

We use T4 [15], a text-templating system in Visual Studio, to create the C# source file for batch types and operators. The source file is compiled, and the dynamic loading facilities of the .NET runtime are used to load and instantiate the types. This technique also allows us to put breakpoints and debug generated code easily. We cache and re-use generated types to reduce the overhead of code generation and compilation. Because we use C# source to define the generated code, we need a way to translate user expressions such as `Where` predicates into inlined C#. An expression is passed to Trill as an *expression tree*, a .NET object model for representing code [10]. Expression trees do not provide a conversion to C# source, since there exist expression trees for which such a translation cannot be done. However, since we are willing to accept a best-effort solution, we wrote our own translator, which is now in use as a stand-alone component for other projects as well.

Columnar execution is best-effort; if we encounter a situation where an operator or type cannot execute in columnar mode, we process the data in row-mode (see Section 4.1). If an expression cannot execute in columnar mode (e.g., it invokes a black-box method), we reconstitute rows on-the-fly to invoke the method. If necessary, we insert a *col-to-row* operator into the query plan. This generated operator converts columnar payloads back into a single column of payload instances for downstream row-based operators. Users are notified when such fallback occurs, so they can try to modify their query or data to remain in the more efficient columnar mode.

4.3 Other Details

Threading By default, Trill does not create any threads: it accepts data on the thread that pushes the data into Trill's ingress methods and executes all operators on that thread until the output (if any) is produced and the call stack is unwound. The one exception is that, depending on a user-configurable option, Trill will use separate threads for scaling operators across multiple cores on a single processor.

Checkpointing We support a client's need for resiliency by offering a synchronous checkpointing service. Under user control, the internal state of a running query can be persisted. The query can later be resumed by loading this state back, possibly on a different machine, and replaying data received since the checkpoint. In conjunction

with Trill's threadless library mode, checkpointing allows Trill to fit in with the existing resiliency solutions of distributed fabrics. The fabric can decide whether it replays events exactly from the checkpoint position for correctness, or resumes from a later (e.g., current) stream position, tolerating the resulting inaccuracy.

5 Usage Scenarios and Lessons Learned

5.1 Usage Scenarios

Trill is being used today in diverse scenarios that serve to illustrate how performance, fabric and language integration, and query model enabled Trill to support a diverse range of use cases.

- Orleans-hosted real-time: Orleans [4] is a programming model and fabric that enables low-latency distributed computations with units of work called grains. Orleans owns threads and manages distribution, while Trill is used as a library to express streaming queries as part of users' grain code.
- Analytics Back-End: Trill is used as a building block for several analytics services. Tempe [9] is a Web-based interactive analytics environment that allows users to author and visualize queries over real-time and offline streams. It uses Trill to run temporal and progressive relational queries. We recently described how the Halo team used Tempe and Trill to quickly analyze large amounts of real-time customer data for hunting down bugs [18]. Azure Stream Analytics [2] is a Cloud service that uses Trill as a query processor [20]. SCOPE [5] is a map-reduce platform that allows arbitrary .NET code as custom reducers. As with Orleans, SCOPE owns threads and schedules reducer code; thus, analysts can embed Trill as a library within their reducers in order to perform temporal analytics [7]. Recently, we also reported on the use of Trill with a streaming version of SCOPE to reduce the latency of Bing Ads reporting [16].
- Monitoring Server: Trill is used to monitor system logs generated by machines in a data center, and visualize real-time performance. Here, Trill is used as a server that processes data from multiple sources in close to real-time (several seconds of latency).
- Trace-Log-Analysis Tools: A large number of time-oriented traces are generated by applications and operating systems. Trill is used as part of stand-alone tools and Cloud services, to allow users to analyze such offline traces, for example, to detect anomalies or complex patterns.

5.2 Lessons Learned

We have learned several things from building Trill and interacting with its users. Our prior work [7, 8] showed that a single model could, in theory, handle a diverse range of analytics scenarios. However, users chose to use specialized systems for performance reasons, which led us to re-examine streaming engine architectures with a goal of achieving best-of-breed or better performance across the latency spectrum.

In addition, a key design decision was to create a *library* instead of a *server*. Implementing Trill as a HLL library meant that it could be immediately integrated into diverse environments, each of which had its own policies on thread management, distribution, scheduling, resiliency, and resource utilization. By default, Trill is passive and performs work only on the thread that feeds data to it. This choice also simplified Trill's implementation considerably, since we could focus on efficient query processing. Subsequently, we created a lightweight scheduler that takes a user-specified set of threads to efficiently use multiple cores on a machine. With this scheduler, we made it easy to build servers using the Trill library as well.

Another crucial aspect was to directly support the HLL data model that users wish to analyze in. For instance, users often wish to stream complex data-types such as dictionaries and machine learning models through Trill. We extended LINQ to make query specification and execution a seamless part of user programming, and our

powerful query language is able to express a wide variety of data processing tasks. Further, columnar batching and code generation needed to be automated and done under the hood, to avoid complicating the user experience.

Finally, none of these decisions would have induced users to adopt Trill as enthusiastically as they have, if it did not work at extremely high speeds. Getting high performance meant starting with a simple `for` loop with an inlined predicate, and working our way out, ensuring that performance was not lost at any step along the way. Once the overall system architecture was decided, it was crucial to observe the resulting design patterns throughout all system components. For example, using custom memory management for the strategic data allocations of batches and columns, restricting the operations performed in the tight loops within each operator, and creating custom data structures (such as hash tables) for optimizing the memory-usage of stateful operators, were all critical to achieving and retaining high performance.

References

[1] ADO.NET DataReader. `http://aka.ms/datareader`. Retrieved 10/14/2015.

[2] Azure Stream Analytics. `https://azure.microsoft.com/en-us/services/stream-analytics/`. Retrieved 10/14/2015.

[3] R. Barga, J. Goldstein, M. Ali, and M. Hong. Consistent streaming through time: A vision for event stream processing. In *CIDR*, 2007.

[4] P. Bernstein, S. Bykov, A. Geller, G. Kliot, and J. Thelin. Orleans: Distributed virtual actors for programmability and scalability. Technical report, Microsoft Research, 2014.

[5] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: easy and efficient parallel processing of massive data sets. *PVLDB*, 1(2), 2008.

[6] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, D. Fisher, J. C. Platt, J. F. Terwilliger, and J. Wernsing. Trill: A High-Performance Incremental Query Processor for Diverse Analytics. *PVLDB*, 8(4), 2014.

[7] B. Chandramouli, J. Goldstein, and S. Duan. Temporal Analytics on Big Data for Web Advertising. In *ICDE*, 2012.

[8] B. Chandramouli, J. Goldstein, and A. Quamar. Scalable Progressive Analytics on Big Data in the Cloud. *PVLDB*, 6(14), 2013.

[9] R. DeLine, D. Fisher, B. Chandramouli, J. Goldstein, M. Barnett, J. F. Terwilliger, and J. Wernsing. Tempe: Live scripting for live data. In *IEEE Symp. on Visual Languages and Human-Centric Computing*, 2015.

[10] Expression Trees. `https://msdn.microsoft.com/en-us/library/bb397951.aspx`. Retrieved 10/14/2015.

[11] M. A. Hammad et al. Nile: A query processing engine for data streams. In *ICDE*, 2004.

[12] C. Jensen and R. Snodgrass. Temporal specialization. In *ICDE*, 1992.

[13] H. Lim et al. How to fit when no one size fits. In *CIDR*, 2013.

[14] D. Maier, J. Li, P. Tucker, K. Tufte, and V. Papadimos. Semantics of data streams and operators. In *ICDT*, 2005.

[15] Microsoft Visual Studio T4 Template System. `http://aka.ms/eeg4w5`. Retrieved 10/14/2015.

[16] Now Available in Bing Ads: Campaign Performance Data in Under an Hour. `http://aka.ms/bing-trill`. Retrieved 10/14/2015.

[17] Reactive Extensions for .NET. `http://aka.ms/rx`. Retrieved 10/14/2015.

[18] The high-tech research behind making Halo 5: Guardians multiplayer better for gamers. `http://aka.ms/fenfxy`. Retrieved 10/14/2015.

[19] The LINQ Project. `http://tinyurl.com/42egdn`. Retrieved 10/14/2015.

[20] Trill Moves Big Data Faster, by Orders of Magnitude. `http://aka.ms/w6y2kt`. Retrieved 10/14/2015.

[21] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *SOSP*, 2013.

Language Runtime and Optimizations in IBM Streams

Scott Schneider
IBM Research
scott.a.s@us.ibm.com

Buğra Gedik
Bilkent University
bgedik@cs.bilkent.edu.tr

Martin Hirzel
IBM Research
hirzel@us.ibm.com

Abstract

Stream processing is important for continuously transforming and analyzing the deluge of data that has revolutionized our world. Given the diversity of application domains, streaming applications must be both easy to write and performant. Both goals can be accomplished by high-level programming languages. Dedicated language syntax helps express stream programs clearly and concisely, whereas the compiler and runtime system of the language help optimize runtime performance. This paper describes the language runtime for the IBM Streams Processing Language (SPL) used to program the distributed IBM Streams platform. It gives a system overview and explains several language-based optimizations implemented in the SPL runtime: fusion, thread placement, fission, and transport optimizations.

1 Introduction

The increase in available data, commonly referred to as *big data*, has caused renewed exploration in systems for data management and processing. Processing this larger volume of data in a timely manner has necessitated moving away from the data-at-rest model, where data is archived in a database, and external applications query and process that data. In order to handle large volumes of data in real time, systems must exploit multiple levels of parallelism at scale.

The MapReduce [9] programming model was widely adopted as a solution in industry to the big data problem. While it brought parallel and distributed programming out of the niche of high performance computing, the model and its implementations have several deficiencies that make it ill-suited for handling online big data. First, the programming model is limited, as all computations must be expressed as *map* and *reduce* operations. In theory, one can express any arbitrary computation with sequences of such operations, but in practice the result may be difficult to understand, and will not necessarily perform well. Second, the design for MapReduce systems were inherently batch-based, which is incongruous with continuous, online data processing. Finally, MapReduce was still a data-at-rest solution: the data was stored in a shared file system prior to running any jobs.

Distributed stream processing is a more appropriate solution for online big data processing. Stream processing systems are designed to contend with continuously arriving data that must be processed quickly. Distributing such computations across a cluster enables the scalability required to deal with large volumes of data. Just as important as the underlying system is the programming model exposed to programmers. The stream processing programming model naturally exposes parallelism that is easily exploitable by the underlying runtime system.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

```
1  stream<CDR> Calls = TCPSource() {
2    param role: client; address: "1.2.3.0";
3  }
4  stream<CDR> UniqueCalls = Deduplicate(Calls) {
5    window Calls: sliding, time(3600.0);
6  }
7  stream<Customer> Customers = TCPSource() {
8    param role: client; address: "1.2.3.1";
9  }
10 stream<CDR, tuple<rstring fromName>> Enriched =
11  Enricher(UniqueCalls; Customers) {
12 }
13 stream<rstring fromName, float64 avgLen> Stats =
14  Aggregate(Enriched) {
15   window Enriched: sliding, time(300.0);
16   output Stats: avgLen = Average(len);
17 }
18 () as Visualize = Dashboard(Stats) {
19 }
20 () as Persist = DBSink(Enriched) {
21   param address: "1.2.3.2"; table: "calls";
22 }
```

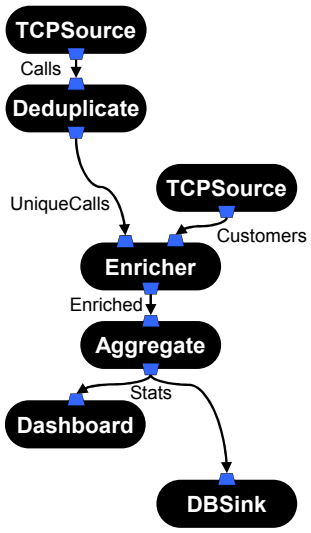


Figure 1: SPL code (left) and corresponding stream graph (right) for telecommunications example.

By allowing programmers to define their applications as independent operators that communicate over streams, distributed stream processing is the full realization of bringing parallel programming to application developers.

This paper presents the runtime for SPL, a stream processing language that targets the IBM Streams [15] platform for distributed stream processing. The SPL runtime was designed with performance as a goal: it supports low-latency, high-throughput streaming applications that execute continuously. SPL is a programming language designed to naturally expose task, pipeline, and data parallelism. The runtime system for SPL exploits such parallelism on hardware ranging from a single machine with many cores to many single-core machines.

Prior publications related to SPL focused on the language design [13] or specific optimizations applied in a streaming context [11, 20]. This paper is the first to focus on the SPL runtime system itself. It identifies the requirements for the SPL runtime, which are focused on the core semantics of the language and delivering high performance that is scalable and configurable. It presents the runtime system that meets those requirements and details its optimizations. These optimizations are possible because of the semantics of the stream programming model realized in SPL.

2 Background on Streaming

This section reviews core streaming concepts as embodied by SPL. It starts with an example application, then introduces development-time concepts, followed by runtime concepts, and wraps up with a discussion of alternative design choices.

Figure 1 shows a simplified version of the real-world telecommunications application presented by Bouillet et al. [6]. The first `TCPSource` in lines 1–3 ingests call detail records (CDRs) from an external system such as a telco switch. The `Deduplicate` in lines 4–6 drops duplicate CDRs in a 1-hour sliding window. The second `TCPSource` in lines 7–9 watches changes to customer information from an external system such as a subscriber database. The `Enricher` in lines 10–12 augments each CDR with a customer name, by buffering `Customers` information in memory and using it to look up names by phone numbers. The `Aggregate` in lines 13–17 computes statistics over a 5-minute sliding window; for simplicity, in this example, the aggregate statistics are just the average call length. The `Dashboard` in lines 18–19 visualizes aggregate statistics for online monitoring, whereas the `DBSink` in lines 20–22 persists them for offline analysis.

The code in Figure 1 exemplifies the development-time view of an application by describing the static structure of an SPL stream graph. Each vertex of the graph is an *operator invocation*, such as the first `TCPSource`. An operator invocation instantiates an operator, configures it (e.g. with a **`param`** clause), and connects it to streams (e.g. the `Calls` stream). *Streams* are the edges of the directed graph of operator instances. *Ports* are the points where streams connect to operator instances. Each operator instance can have zero or more input ports and zero or more output ports, and each port can be connected to zero or more streams. An operator, such as `TCPSource`, is a template that can be instantiated multiple times. Different invocations of the same operator can be configured differently; the specifics for how SPL supports this configuration are not relevant to this paper and can be found elsewhere [13]. What is important is that SPL poses no restrictions on permitted topologies, which can have multiple roots (sources) and leaves (sinks) and may even be cyclic. This flexibility makes it possible to support a broad set of applications. However, once deployed, the stream graph is fixed, to support static optimization.

The runtime view of SPL adheres to the semantics for the dynamic behavior of an SPL application. At runtime, streams send tuples between operator instances. Most non-source operators only activate when a tuple arrives at an input port. (Source operators activate based on external triggers; from the perspective of the application, they appear to activate spontaneously.) Note that each tuple causes a separate activation that consumes the tuple that triggered it; as a corollary, ports fire independently. The per-tuple activation semantics minimize the need for synchronization and are formalized in the Brooklet calculus [22]. An operator activation typically modifies operator-local state (if any) and submits zero or more tuples on output ports (if any), and then the operator becomes passive again waiting for the next activation. State is in-memory and operator-local, and thus state access requires no inter-operator coordination, avoiding a performance bottleneck for distributed deployments. When multiple streams converge on a single port, their tuples are interleaved in an unspecified order. When multiple streams originate from a single port, they all carry the same tuples.

Some streaming languages, such as CQL [5], focus primarily on relational operators. In contrast, SPL has an extensive operator library of which relational operators make up only a small fraction. This library, and the support for user-defined operators, provide an ecosystem for SPL applications in diverse domains. Some other streaming languages, such as StreamIt [12], focus primarily on operators with statically known selectivity. The *selectivity* of an operator is the number of tuples consumed and produced in an activation. In SPL, activations of non-source operators consume exactly one tuple, but the number of tuples produced varies per activation and is not known statically. Again, this design choice was important for SPL to work in many domains.

Overall, SPL provides the generality needed to address many applications and run on a distributed system, while retaining enough static information for language-based optimizations as described later in this paper.

3 Requirements

The runtime system for a streaming language has two primary responsibilities: to enforce the semantics of the programming language and to deliver high performance.

3.1 Enforcing Semantics

The responsibility of enforcing the semantics of the programming model belongs primarily to the compiler. But it is the language runtime that provides the streaming primitives that the compiler targets. A runtime for SPL has the following requirements:

- *Operator-state protection:* Operator state is exclusively local to that operator. The runtime is responsible for enforcing operator-state protection, even if multiple operators happen to execute in the same address space.

- *Asynchronous tuple-at-a-time:* Operators must be able to asynchronously yet safely process individual tuples. The runtime is responsible for delivering tuples to operators while preventing data races and deadlock.

- *Ordered streams:* Operators must be able to send tuples over streams to other operators for asynchronous processing. The runtime must deliver tuples to the operators that consume the stream even if the consuming operators are on a separate machine. The runtime must also maintain tuple order on streams: if an operator submits tuple *a* before tuple *b*, all operators that receive tuples from that stream must receive *a* before *b*.

- *Communication across applications*: Stream programs must be able to choose to publish or subscribe to streams from other stream programs. The runtime is responsible for matching publication and subscription specifications as new applications enter the system, and for delivering the appropriate tuples.

3.2 Delivering High Performance

From an implementation perspective, delivering high performance is at odds with enforcing language semantics: the simplest means to enforce the semantics tend to result in unacceptable performance. The following requirements are needed for SPL to deliver high performance, and will determine the runtime optimizations:

- *High throughput:* The primary performance metric for most SPL applications is throughput: tuples processed per second.

- *Low latency:* Tuple processing must not incur undue latency for any individual tuple. This requirement means that aggressively optimizing for throughput via large batches is not acceptable.

- *Continual processing:* Applications must be able to execute indefinitely, without the loss of performance. The runtime must be designed such that a single application can process data continuously for months.

- *System independence:* The abstractions provided in SPL allow any given application to map to any arbitrary distributed system. The SPL runtime must deliver on this promise, in both directions. The runtime must provide the ability for the same application to execute on many different kinds of distributed systems, and, given a particular distributed system, the runtime must be able to handle any arbitrary SPL application.

- *Parallel execution:* Operators in an application must be able to execute across a distributed cluster, in parallel. Parallelism is one of the means through which the runtime delivers high throughput, so any decision that limits parallelism must improve performance in some other way.

- *Explicit user control:* Experts with a deep understanding of the underlying distributed system—and how the abstractions in SPL map to that system—need to be able to control how their applications are deployed. That control is required both for influencing the optimizations in the runtime (such as parallelism or cheap communication) and for dealing with the constraints of a particular system (such as which machines in a cluster are allowed to access remote data sources).

4 System Overview

Creating and executing distributed streaming applications is more involved than the typical compile-and-execute model for general-purpose languages. This section gives a brief overview of the system as a whole, including the artifacts that are introduced in each stage of the application life cycle.

As a platform for distributed and parallel applications, IBM Streams must provide services such as name resolution, application life-cycle management, and scheduling. However, platform services are outside of the scope of this paper, which focuses on the SPL runtime.

Compilation. The primary entity in the SPL runtime is the *processing element*, or PE. Multiple operators can execute inside a PE, and determining which operators will execute together in the same PE is called *fusion*.

The compiler is responsible for operator fusion. The two main artifacts produced by compiling an SPL application are the PEs and the ADL (application description language) file. The optimization aspects of fusion are covered in Section 6.1. From the system's perspective, the PEs are dynamic libraries that contain the code for all of the operators fused into that PE. The ADL contains a meta-description of the entire application, including all of the PEs and the operators they contain. The connections between all operators within each PE, and between all PEs, are fully represented in the ADL.

Developers can annotate operator invocations to parallelize arbitrary sub-graphs. The compiler recognizes these annotations, but it does not perform the parallel expansion. Instead, it records in the ADL which regions of the stream graph should be parallelized at job submission time.

Job submission. SPL applications start executing when the ADL for the application is submitted to the Streams platform. Parallel expansion occurs at job submission, using the information from the ADL to indicate which portions of the application should be parallelized. The transformation process produces the PADL (physical ADL), which is the final representation of the stream graph that will execute.

The transformation process replicates all relevant operators and streams, and is responsible for connecting the replicated streams back into the unparallelized portions of the application. Because fusion happened at compile time, the parallel expansion cannot change which operators are in which PEs. There are two means by which it can achieve parallelism: replicate an entire PE, or replicate operators within a PE and inject threaded ports to ensure parallelism. In both cases, the PE binaries remain unchanged; the replication happens entirely in the stream graph representation in the PADL. This late-stage transformation is enabled by the separation between the high-level description of the application in the ADL and the actual code that executes in the PE binaries.

From the PADL, the Streams platform creates an AADL (augmented ADL) for each PE, which details what part of the stream graph that PE is responsible for. Finally, the platform is responsible for scheduling the PEs on the available hosts.

Execution. The Streams platform launches all of the PEs in the SPL application. Upon start-up, the PEs refer to their AADL to know which operators to start, how those operators are connected to each other, how those operators are connected to the input and output ports of the PE itself, and which connections to establish with the other PEs in the application. PEs created through the parallel expansion will execute the same PE binary, and operators replicated inside of PEs will simply instantiate the same operator multiple times.

Cancellation. Unlike applications in general-purpose languages, streaming applications are designed to execute indefinitely. For that reason, users must explicitly tell the Streams platform to cancel a particular job. When a PE receives a cancellation notification from the platform, it informs the operators it is responsible for, so they can safely clean up their resources.

5 The SPL Runtime

The SPL runtime manages the life-cycle and execution of the operators that are contained within the same PE. It also interacts with the larger Streams runtime to participate in application life-cycle management, dynamic connection management, metrics collection, and remote debugging support.

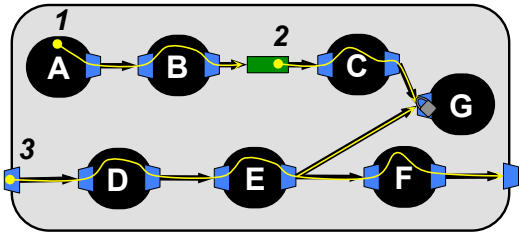


Figure 2: PE with three threads: a thread in a source operator, a threaded port, and a thread from the PE input port.

5.1 PE Execution

Operators within the same PE are executed as a single operating system process. The system component responsible for inter-PE communication is called the *transport*. The SPL runtime can use multiple threads within a PE to execute the PE's operators, as shown in Figure 2. In particular, source operators and input ports that are fed by the transport (PE input ports) are driven by dedicated threads. These threads execute the operator graph that is downstream of their associated source operators or PE input ports. The stream connections within a PE are implemented via function calls, using simple reference passing to avoid costly serialization. Tuples that go through inter-PE connections are buffered within the transport, whereas those that go through intra-PE connections implemented by function calls are not buffered. Further parallelism is achieved within a PE via the use of *threaded ports*. A threaded port is an input port within a PE that maintains a tuple buffer and uses a dedicated thread to execute its downstream operator graph. These threaded ports can be inserted manually by the application developer, as well as automatically by the SPL runtime [23]. In addition to these, individual operators can also request one or more SPL runtime managed threads for executing asynchronous tasks.

5.2 Operator Execution

The SPL runtime and the user-defined operators interact via an event-driven model. Operators handle tuples by implementing a tuple-handler function. They can submit tuples to their output ports, either as part of the tuple-handler function in reaction to a tuple arrival, or as part of the asynchronous tasks they execute. SPL also supports *punctuations*, which are out-of-band signals embedded within the tuple flow. Punctuations are handled via handler functions just like tuples. They can also be submitted to output ports. Two kinds of punctuations are supported: *window* punctuations and *final* punctuations.

Window punctuations are used to mark window boundaries within a stream. They enable custom windowing semantics, where the boundaries of the windows are not determined by a predefined windowing policy, but instead they are determined based on the presence of window punctuations in the stream.

Final punctuations are used to handle application termination. Receiving a final punctuation on an input port indicates that no tuples are to be received from that input port in the future. The SPL runtime manages the creation and forwarding of final punctuations automatically. Operators can opt to handle final punctuations in order to perform finalization tasks.

5.3 Window Management

SPL offers windowing syntax for any operator, not limited to relational ones. The SPL runtime facilitates the implementation of such windowed operators by providing a windowing API. In particular, the SPL runtime maintains windows in-memory, provides access to window contents, and lets user-defined operators register callback functions to handle various windowing events. SPL supports *tumbling* and *sliding* windows, including

partitioned varieties. Tumbling windows are non-overlapping, whereas sliding windows are potentially overlapping. Partitioned windows maintain independent windows for different sub-streams based on a partitioning attribute. Windows are configured via *window eviction* and *window trigger* policies. SPL supports time-based, count-based, and attribute-delta based eviction and trigger policies [10]. For a tumbling window, the eviction policy specifies when to flush the window, such as after every 10 tuples or after the timestamp attribute increases by 10 units. For a sliding window, the eviction policy specifies when to evict old tuples from the window, such as when the window size grows beyond 10 (as a count or based on a timestamp attribute). For a sliding window, the trigger policy specifies when to process the window contents, such as after every 2 tuples, or after the timestamp attribute increases by 2 time units. Tumbling windows do not have trigger policies, as they trigger when the window is flushed.

5.4 Back-Pressure Management

The SPL runtime implements back-pressure to handle potential differences in the processing rates of operators. When an operator is faster than those downstream of it, submit calls will eventually block, as the downstream operators' input port buffers will be full. This will in turn slow down the operator at hand. As time progresses, the back-pressure will propagate further upstream. It will eventually reach source operators, and through them, external sources. Via the use of back-pressure, streaming operators naturally throttle themselves to avoid continuously growing buffers, without the need for shedding any tuples. Since the SPL runtime implements tuple submissions via function calls within a PE, back-pressure manifests at the boundaries where tuple submissions go through a buffer. These include tuple submissions to PE output ports (that go into the transport buffers) and tuple submissions to output ports that are connected to threaded ports (that go into the threaded port buffers).

SPL allows feedback loops in its flow graphs, where a downstream operator can produce an output that is fed back into the input port of an upstream operator. Such feedback loops create cycles in the flow graph, yet arbitrary cycles can cause deadlocks in the presence of back-pressure. To avoid deadlocks, SPL only allows feedback connections into *control ports*. A control port is a special kind of input port with the restriction that it cannot trigger the production of output tuples. Typically, control ports consume the incoming tuples to update the operator's internal state.

5.5 Consistent Regions

SPL applications can achieve fault tolerance through user-applied *consistent regions* [8]. Tuples in consistent regions are guaranteed to be processed at least once, even in the presence of operator and PE failure. The SPL runtime achieves this guarantee with a combination of operator state checkpointing and tuple replay. Source operators in consistent regions periodically send out special punctuations that inform operators that it is time to checkpoint their local state. Because streams are ordered, when an operator checkpoints its local state, it is guaranteed that the state contains the result of all tuples prior to the punctuation. The accumulated application state across all operators after they have all finished checkpointing is a consistent view of the application's state.

In the event of a failure, the platform notifies the source operators in consistent regions. The source operators then send out another special punctuation that tells all operators in the region to discard their current state, and reload their state from their last checkpoint. Following that punctuation is a replay of tuples that came after the last checkpoint. Through failure tracking, checkpointing, and a specialized protocol, the SPL runtime is able to guarantee at-least-once tuple processing. If the operators in the consistent region do not have externally visible behavior that cannot be rolled back, then from an operator developers perspective, this guarantee becomes exactly-once.

5.6 Dynamic Connections

A typical stream connection is established between an operator output port and an operator input port, based on the connection specification defined within an SPL program. Such connections are considered *static*. A complementary form of connections are *dynamic connections*, where the exact endpoints are established at runtime, subject to constraints specified in an SPL program at compile-time. Dynamic connections enable a few use cases that cannot be satisfied by static connections. One such use case is incremental deployment of applications, where an application is deployed in piecemeal fashion, adding new components as the application evolves. Another example is dynamic discovery of sources and sinks, where an application is designed to consume/produce data from/to a variable set of producers and consumers. These producers and consumers can be other applications sharing the same runtime instance. As a concrete example, in an operational monitoring application, new log sources (producers) as well as new analytic applications (consumers) could be added/removed at runtime via the use of dynamic connections.

SPL supports dynamic connections via *export properties* and *import specifications*. An output port that produces a stream can export it by associating a list of export properties with the stream. Dually, an input port that consumes streams can import them by providing an import specification. Import specifications are Boolean expressions that make use of export properties and basic arithmetic and logical operations on them. Both export properties and import specifications can either be defined within SPL programs or dynamically changed via runtime APIs. Based on export properties and import specifications, the Streams runtime performs continuous matching to determine changes on the dynamic connections. When such changes are detected, it coordinates with the SPL runtime to establish new connections and/or tear down existing ones to keep the dynamic connections up to date. Changes in the dynamic connections can happen due to changes in the list of SPL applications running within a Streams instance, or due to changes in the export properties or import specifications of existing SPL applications.

5.7 Dynamic Filters

Dynamic connections enable operators to subscribe to streams on demand. However, once a stream is subscribed via an import specification, its entire contents are received, since the matching is on stream-level export properties and not on tuple-level attributes. To support subscribing to a selective subset of imported streams, SPL supports *dynamic filters*. Dynamic filters, which can be specified together with import specifications, are Boolean expressions defined on tuple attributes. These filters are shipped by the Streams runtime to the PEs that are producing the exported streams and are evaluated by the SPL runtime to perform the filtering.

6 Runtime Optimizations

The SPL runtime implements several optimizations, with a particular focus on maximizing the throughput of applications by taking advantage of parallelization and distribution opportunities.

6.1 Fusion

The fusion optimization aims at grouping operators into PEs, so that the stream-processing application can be distributed over multiple hosts. Since process migration is costly, SPL performs fusion at compile-time. However, profiling data is collected during runtime and earlier runs guide the fusion decisions based on this profiling data. The profile-optimize cycle can be iterated to improve accuracy.

Fusion is a graph-partitioning problem, where the goal is to minimize the volume of data flow between PEs, while keeping the total cost of operators within a PE under a limit. Minimizing the volume of data flow between PEs minimizes the costly transmission of tuples across PEs, since stream connections are implemented

as function calls within a PE. Limiting the total cost of operators within a PE avoids overloading a single host and makes it possible to utilize multiple hosts. The partitioning of the application flow graph for fusion can be implemented bottom-up, starting with one operator per PE and iteratively merging PEs; or top-down, starting from a single PE and iteratively dividing PEs. SPL's *auto-fuser* takes the latter approach, which is shown to have better performance [17] and can be easily adapted to work in the presence of the fission optimization in Section 6.3 [20].

SPL also enables application developers to explicitly request fusion via PE-level *co-location*, *ex-location*, and *isolation* directives. Co-location places a group of operators into the same PE. Ex-location enforces that a group of operators pair-wise do not share their PEs. Isolation runs an operator inside a PE by itself, with no other operators present. SPL's auto-fuser respects these fusion constraints.

6.2 Intra-PE Thread Placement

The intra-PE thread placement optimization aims to take advantage of multiple cores on a single host for executing operators within a PE. It can exploit both pipeline and task parallelism inherently present in streaming applications. In SPL, threaded ports perform thread placement. However, it is difficult to find a close-to-optimal configuration by hand, because it depends on the per-tuple costs and selectivities of operators. These properties are difficult to guess at development time. Furthermore, the number of possible placements increases combinatorially with the number of input ports and hardware threads available in the system. SPL solves this problem via an *auto thread placer*[1] that can automatically insert threaded ports as the application is executing [23].

The auto thread placer is a runtime component that incorporates a profiler and an optimizer. The profiler uses an application-level operator stack to track thread execution and periodically samples this stack to measure operator costs and thread utilizations. The optimizer uses these values to find bottleneck threads and decides where to insert threaded ports to maximize the application throughput. Additional runtime machinery is used to put these decisions into effect with minimal disruption to the active data flow. The process is iterative, where at each iteration additional threaded ports are added until no further improvements are possible.

The key insight used by SPL's auto thread placer is that, at each step, additional threaded ports decrease the workload of all of the highly utilized threads, as otherwise the optimization process will get stuck at a local minimum. This is particularly due to the dependence of the throughput on the slowest component of a pipeline. Another important consideration is that, sometimes, adding new threaded ports may not improve performance due to external effects, such as globally shared resources like files, locks, and databases. The auto thread placer monitors the achieved performance after changes in the threaded port configuration, in order to rescind ineffective changes. It also uses a blacklist to avoid them in the future.

6.3 Fission

Fission is an optimization that exploits data parallelism. To apply fission, a region of the application graph is replicated, the data is distributed over these replicas via a *split* operator, and the results from the replicas are re-ordered via a *merge* operator. In Streams, fission can be user-defined or automatic[2]. In user-defined fission, the application developer annotates the region that will take advantage of data parallelism, called the *parallel region*, and specifies the number of replicas. The runtime system handles the actual instantiation of the replicas, the distribution of tuples over the replicas, and the re-ordering at the end to maintain the sequential semantics.

Auto-fission both detects parallel regions and determines the number of parallel channels automatically, without involving the application developer. Auto-fission requires static code analysis to determine when the optimization is safe and runtime support to maintain that safety. The SPL compiler locates data-parallel regions by analyzing operator models as well as the configurations of the individual operator instances in the SPL

[1] Auto thread placer is available in a research version of the system [23].

[2] Auto-fission is available in a research version of the system [11, 20].

program [20]. It uses a left-to-right heuristic to consider operators in the graph and merges as many consecutive operators as possible into a parallel region to minimize parallelization overhead. The left-to-right heuristic is motivated by the observation that most streaming applications apply progressive filtering. Operators can be combined into parallel regions if they are suitable for data parallelism and their partitioning keys are compatible. Only operators that are either stateless or partitioned stateful can be used for data parallelism.

Auto-fission automatically discovers the degree of parallelism that achieves the best throughput, and adapts to changes in workload and resource availability. For this purpose a control algorithm is implemented within the splitters [11]. It uses throughput and congestion metrics to adjust the number of channels for the parallel region. The basic principle behind the control algorithm is to increase the number of channels until the congestion goes away. However, if the congestion is due to a downstream bottleneck that cannot be resolved by the parallel region at hand, then this situation is detected by the lack of improvement in the throughput in response to an increase made in the number of channels. Various additional mechanisms are employed to satisfy *SASO* properties: stability (no oscillations), accuracy (close to optimal throughput), settling time (number of channels is set quickly), and overshoot (no excessive resource consumption). In the presence of partitioned stateful operators, auto-fission requires support for state migration. Migration is needed whenever the number of channels changes, as some partitions are assigned to new operators. SPL addresses this issue by automatically managing operator state via a key-value store [11], using consistent hashing [16] to minimize the amount of data migrated.

6.4 Transport Optimizations

The Streams runtime provides various transport options, including InfiniBand for high-performance network hardware, TCP for general-purpose inter-host PE communication, and Unix domain sockets for intra-host PE communication. Various configuration options are provided related to buffering of tuples by the transport as well as thread usage for receiving tuples, in order to adjust the trade-off between latency and throughout.

The SPL runtime uses serialization and deserialization to transform between in-memory and on-the-wire representation of tuples. For highly performance-sensitive applications, this conversion may introduce significant overhead. Given SPL's dynamically-sized types (strings, lists, maps, and sets), these transformations are necessary in the general case. The SPL runtime implements an optimization called *façade tuples* to eliminate this overhead when the tuples involved contain only fixed-size types. The SPL language's support for fixed-size types includes bounded strings and bounded versions of lists, maps, and sets, in addition to the regular primitive types. Fixed-size types always occupy space corresponding to their maximum size, irrespective of their current effective size. The façade tuple optimization uses the same on-the-wire and in-memory representation for tuples that contain only fixed-size attributes. On the down-side, accessing façade tuple attributes might result in unaligned memory access, which may be unavailable in some systems and slightly slower in others.

7 Related Work

The first main topic of this paper is the distributed runtime system for SPL. Here, we compare SPL's runtime to other streaming runtimes.

Like SPL, TelegraphCQ [7] and CQL [5] enable continuous dataflow processing. Furthermore, like SPL, CQL has a language-centric design. However, both TelegraphCQ and CQL focus on relational stream queries, whereas a primary objective of SPL is support for operators beyond the relational domain. Furthermore, unlike SPL, TelegraphCQ and CQL lack distributed runtimes. Borealis pioneered distributed stream-relational systems [1]. However, it did not have a language-centric design. Therefore, unlike SPL, Borealis does not offer language-based optimizations. Another streaming platform with a language-centric design is StreamIt [12]. It does not emphasize a relational approach and supports distribution. However, unlike SPL, StreamIt only allows a restrictive set of topology combinators, ruling out commonly-needed cases such as multiple sources

or sinks. Furthermore, unlike SPL, StreamIt focuses on operators with statically known selectivity. Microsoft StreamInsight is a streaming platform that derives from earlier stream-relational systems [4]. However, by using LINQ (language-integrated queries), it augments its relational foundation with user-defined code. Unlike SPL, StreamInsight was not designed with a distributed runtime in mind.

Recently, there has been a flurry of new streaming platforms that primarily focus on distribution: Google Millwheel [2], Spark Streaming with its micro-batch approach [25], Microsoft Naiad with its timely dataflow approach [18], and Twitter Storm [24]. Like SPL, they advance the state of the art for scalable and resilient distribution. However, none of them use a language-centric design, which means that unlike SPL, they do not offer much in the way of language-based optimization.

The second main topic of this paper is language-based optimizations for SPL. Here, we review streaming optimizations work that is closely related to SPL. For a comprehensive overview, see our survey paper [14]. Optimization algorithms must tackle two challenges, safety and profitability. Safety ensures that the optimized application produces the same results as the original code, and profitability ensures that it runs faster or uses less resources or scales to bigger work-loads.

Fusion combines operators to avoid the overhead of serialization and transport. There are variants of fusion depending on whether the operators are only combined in a single process or also in a single thread [23]. Fusion safety tends to be easy to establish. COLA offers a sophisticated solution to fusion profitability in the context of SPL [17]. Languages that focus on streaming with statically known selectivity solve fusion profitability even more comprehensively [21]. Fission introduces data parallelism by replicating an operator or even an entire subgraph of the stream graph. Fission is the killer optimization for StreamIt [12]. In the context of SPL, we have researched both fission safety [20] and fission profitability [11]. Fission is so important for performance that recent streaming platforms design partitioning deeply into their semantics to make fission the default [2, 18, 24, 25]. Transport optimizations reduce the overheads for sending tuples between distributed streaming operators across process or machine boundaries. The SPL runtime includes a highly optimized transport fabric with good defaults, but can be further tuned for extreme situations [19]. Many other distributed streaming systems start out with higher transport overheads, which can be optimized by reducing threads, serialization, etc. [3].

8 Conclusion

This paper describes the SPL language runtime and its optimizations. The SPL runtime provides the system support for hosting a graph of operators on multiple cores and multiple machines while enforcing the semantics of the programming language. Furthermore, the SPL runtime supports several language-based optimizations: fusing operators in the same operating-system process to reduce communication cost; placing multiple threads into such a process to increase intra-machine parallelism; using fission to replicate subgraphs of operators to increase inter-machine parallelism; and optimizing the transport to eliminate serialization overheads. The SPL runtime enables both user-directed and fully-automated variants of these optimizations.

References

[1] D. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the Borealis stream processing engine. In *Innovative Data Systems Research Conference (CIDR)*, 2005.

[2] T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. MillWheel: Fault-tolerant stream processing at internet scale. In *Very Large Data Bases (VLDB) Industrial Track*, pages 734–746, 2013.

[3] S. Akram, M. Marazakis, and A. Bilas. Understanding and improving the cost of scaling distributed event processing. In *International Conference on Distributed Event-Based Systems (DEBS)*, pages 290–301, 2012.

[4] M. Ali, B. Chandramouli, J. Goldstein, and R. Schindlauer. The extensibility framework in Microsoft StreamInsight. In *International Conference on Data Engineering (ICDE)*, pages 1242–1253, 2011.

[5] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: semantic foundations and query execution. *Journal on Very Large Data Bases (VLDB J.)*, 15(2):121–142, 2006.

[6] E. Bouillet, R. Kothari, V. Kumar, L. Mignet, S. Nathan, A. Ranganathan, D. S. Turaga, O. Udrea, and O. Verscheure. Experience report: Processing 6 billion CDRs/day: From research to production. In *Conference on Distributed Event-Based Systems (DEBS)*, pages 264–267, 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. A. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Conference on Innovative Data Systems Research (CIDR)*, 2003.

[8] G. J. da Silva. Guaranteed tuple processing in InfoSphere Streams v4 with consistent regions. `https://developer.ibm.com/streamsdev/2015/02/20/processing-tuples-least-infosphere-streams-consistent-regions/`. Retrieved December, 2015.

[9] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Operating Systems Design and Implementation (OSDI)*, pages 137–150, 2004.

[10] B. Gedik. Generic windowing support for extensible stream processing systems. *Software: Practice & Experience (SP&E)*, 44(9):1105–1128, 2014.

[11] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu. Elastic scaling for data stream processing. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 25(6):1447–1463, 2014.

[12] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 151–162, 2006.

[13] M. Hirzel, H. Andrade, B. Gedik, G. Jacques-Silva, R. Khandekar, V. Kumar, M. Mendell, H. Nasgaard, S. Schneider, R. Soulé, and K.-L. Wu. IBM Streams Processing Language: Analyzing big data in motion. *IBM Journal of Research and Development*, 57(3/4):7:1–7:11, 2013.

[14] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm. A catalog of stream processing optimizations. *ACM Computing Surveys (CSUR)*, 46(4), Apr. 2014.

[15] IBM Streams. `http://ibmstreams.github.io/`. Retrieved September, 2015.

[16] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Symposium on the Theory of Computing (STOC)*, pages 654–663, 1997.

[17] R. Khandekar, K. Hildrum, S. Parekh, D. Rajan, J. Wolf, K.-L. Wu, H. Andrade, and B. Gedik. COLA: Optimizing stream processing applications via graph partitioning. In *International Middleware Conference*, pages 308–327, 2009.

[18] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *Symposium on Operating Systems Principles (SOSP)*, pages 439–455, 2013.

[19] Y. Park, R. King, S. Nathan, W. Most, and H. Andrade. Evaluation of a high-volume, low-latency market data processing system implemented with IBM middleware. *Software: Practice & Experience (SP&E)*, 42(1):37–56, 2012.

[20] S. Schneider, M. Hirzel, B. Gedik, and K.-L. Wu. Auto-parallelizing stateful distributed streaming applications. In *Parallel Architectures and Compilation Techniques (PACT)*, pages 53–64, 2012.

[21] J. Sermulins, W. Thies, R. Rabbah, and S. Amarasinghe. Cache aware optimization of stream programs. In *Languages, Compiler, and Tool Support for Embedded Systems (LCTES)*, pages 115–126, 2005.

[22] R. Soulé, M. Hirzel, R. Grimm, B. Gedik, H. Andrade, V. Kumar, and K.-L. Wu. A universal calculus for stream processing languages. In *European Symposium on Programming (ESOP)*, pages 507–528, 2010.

[23] Y. Tang and B. Gedik. Autopipelining for data stream processing. *Transactions on Parallel and Distributed Systems (TPDS)*, 24(12):2344–2354, Dec. 2013.

[24] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy. Storm @twitter. In *International Conference on Management of Data (SIGMOD)*, pages 147–156, 2014.

[25] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Symposium on Operating Systems Principles (SOSP)*, pages 423–438, 2013.

FUGU: Elastic Data Stream Processing with Latency Constraints

Thomas Heinze[1], Yuanzhen Ji[1], Lars Roediger[1], Valerio Pappalardo[1], Andreas Meister[2], Zbigniew Jerzak[1], Christof Fetzer[3]

[1]SAP SE {firstname.lastname}@sap.com

[2]University of Magdeburg andreas.meister@iti.cs.uni-magdeburg.de

[3]TU Dresden christof.fetzer@tu-dresden.de

Abstract

Elasticity describes the ability of any distributed system to scale to a varying number of hosts in response to workload changes. It has become a mandatory architectural property for state of the art cloud-based data stream processing systems, as it allows treatment of unexpected load peaks and cost-efficient execution at the same time. Although such systems scale automatically, the user still needs to set configuration parameters of a scaling policy. This configuration is cumbersome and error-prone.

In this paper we propose an approach that tries to remove this burden from the user. We present our data stream processing system FUGU, which optimizes the selected scaling policy automatically using an online parameter optimization approach. In addition, we demonstrate how our system considers user-defined end to end latency constraints during the scaling process.

1 Introduction

Data stream processing systems [1] continuously produce output for a set of standing queries and potentially unbounded input streams. Many real-world workloads for data stream processing systems have a high variability, which means that the data rates of the input streams and the selectivities of query operators are frequently changing in unpredictable ways. Several authors [5, 7, 8] have proposed data stream processing prototypes that automatically scale in or out based on workload characteristics to handle such dynamic workloads. Such systems are called *elastic* [11] and support increasing system utilization by using only the minimum required number of hosts. However, in all these prototypes, the user needs to manually specify a scaling strategy, which controls when and how the system scales.

The challenge of correctly configuring the scaling strategy has been studied for many cloud-based systems [4, 12, 13, 16]. A large number of solutions exist, including auto-scaling techniques [13, 16] and task-classification approaches [4, 12]. These systems can be classified into three major algorithmic categories: prediction-based, sampling-based, and adaptive (learning-based) solutions. Both sampling and prediction-based approaches are hard to apply in a data stream processing system, because its workload is hard to predict or sample due to its high variability. An adaptive auto-scaling technique is able to improve the utilization of such a system, but degrades the quality of service [9]. Each reconfiguration decision in a data stream processing system interferes

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

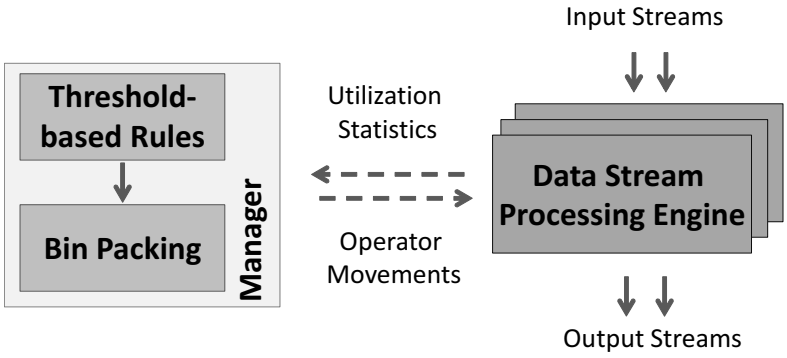


Figure 1: Architecture of FUGU

with the data processing and as a result has a high impact on major quality of service metrics such as end to end latency [8]. Therefore, this characteristic needs to be reflected in the scaling strategy to achieve a good trade-off between the spent monetary cost and the achieved quality of service.

In the context of our elastic data stream processing prototype FUGU, we study how we can relieve the user from configuring these parameters and how to support different quality of service levels. In this paper, we outline the two major concepts we use to realize this vision in context of FUGU: (1) the latency-aware scaling strategy and (2) online parameter optimization. The latency-aware scaling strategy introduces a model to estimate the latency peak created by a scaling decision. This information is used to derive scaling decisions with a minimal latency peak and avoid scaling decisions with a too high latency peak. Online parameter optimization presents a white-box model to study the influence of different parameters on scaling behaviour. This white-box model can be used to search for good parameter settings for the current workload.

In the following, we describe both techniques in the context of an existing data stream processing system. In addition, we present a real-world evaluation to demonstrate the strength of the presented techniques.

2 Background

The concepts presented here are implemented as an extension of the elastic data stream processing prototype FUGU [8, 9] (see Figure 1). The existing system consists of a centralized management component, which dynamically allocates a varying number of hosts. The manager executes on top of a distributed data stream processing engine, which is based on the Borealis semantic [1].

The data stream processing system processes continuous queries, which can be modeled as directed acyclic graphs of operators. Our system supports primitive relational algebra operators (selection, projection, join, and aggregation) as well as additional data stream processing specific operators (sequence, source, and sink). Each operator can be executed on an arbitrary host and a query can be partitioned over multiple hosts. The number of hosts is variable and dynamically adapted by the management component to changing resource requirements.

The centralized management component serves two major purposes: (1) it derives scaling decisions, including decisions on allocating new hosts or releasing existing hosts, and assigns operators to hosts; and (2) it coordinates the construction of the operator network in the distributed data stream processing engine.

The management component constantly receives statistics from all running operators in the system. Based on these measurements and a set of thresholds and parameters, it decides when to scale and where to move operators. Typically, these thresholds and parameters are manually specified by the user. Our system supports the movement of both stateful (join and aggregation) and stateless operators (selection, sink, and source). A state of the art movement protocol [8, 15] ensures an operator moves to the new host without information loss.

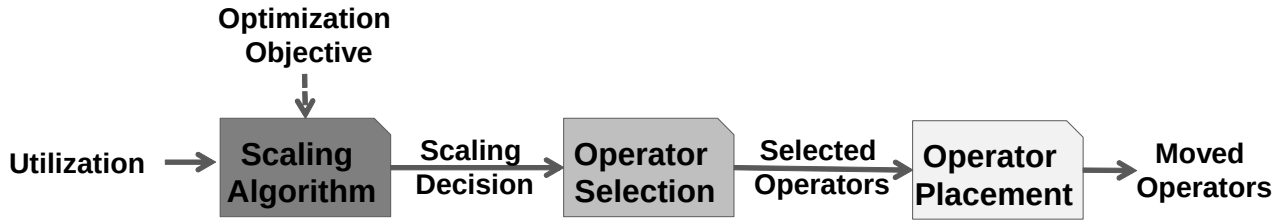


Figure 2: Scaling Strategy of FUGU

3 Threshold-based Elastic Scaling

The scaling approach used by the FUGU server is illustrated in Figure 2. A vector of node utilization measurements (CPU, memory, and network consumption) and a vector of operator utilizations are used as input to the *Scaling Algorithm*. The *Scaling Algorithm* derives decisions that mark a host as overloaded or the system as underloaded. The *Operator Selection* algorithm decides which operators to move and the *Operator Placement* algorithm determines where to move these operators.

The default scaling strategy of FUGU is threshold-based, namely, a set of threshold rules are used to define when the system needs to scale. These thresholds mark either the entire system or an individual host as over/underloaded. A threshold rule describes an exceptional condition for the consumption of one major system resource (CPU, network, or memory), which triggers a scaling decision in FUGU. Some examples for these rules include:

1. A host is marked as overloaded if the CPU utilization of the host is above 80% for three seconds.
2. A host is marked as underloaded if the CPU utilization of the host is below 30% for five seconds.

The threshold-based rules need to be used carefully [6]. In particular, the frequent alternating allocation and deallocation of virtual machines, called thrashing, should be prevented. Several steps are taken in FUGU to avoid thrashing. First of all, each threshold needs to be exceeded for a certain number of consecutive measurements before a violation is reported. This number is called the *threshold duration*. In addition, after a threshold violation is reported, no additional scaling actions are done for the corresponding host for a certain time interval called a *grace period* (or cool-down time). The system checks for overloaded or underloaded host each time a new batch of utilization measurements for all operators has been received. Our scaling strategy checks all hosts using the overload criteria first, afterwards it tests if the system is underloaded. This order avoids to first release a host due to an underload and afterwards allocate a new host to solve an overload.

The load in a data stream processing system is partitioned among all operator instances running in the system. Therefore, each scaling decision needs to be translated into a set of moved operators. The first problem is to identify which operators to move. This identification is done by the *Operator Selection* algorithm. If the system is marked as underloaded, it selects all operators running on the least loaded hosts. For an overloaded host, the *Operator Selection* algorithm chooses a subset of operators to move in a way, that the summed load remaining on the host is smaller than the given threshold. FUGU models this decision as a *subset sum problem* [14], where the operators on the host are the possible items and the threshold represents the maximum sum. We use a heuristic, which identifies the subset of all operator instances whose accumulated load is smaller than the threshold and no other subset with a larger accumulated load fulfilling this condition exists. All operators selected by this algorithm are kept on the host; the remaining operators are selected for movement.

The selected operators are the input of the *Operator Placement* algorithm, which decides *where* the operators should be moved. We solve this problem using different bin packing algorithms [3]. The goal of a bin-packing algorithm is to assign each item to exactly one bin in a way that (1) the number of bins is minimized and (2)

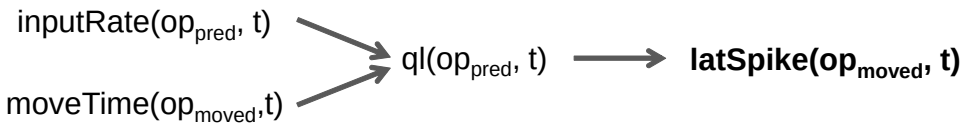


Figure 3: Latency Peak Estimation

the sum of the weights of all assigned items is smaller than the capacity of the bin. In the context of FUGU, an operator represents an item and its CPU usage is its weight. A host is modeled as a bin with its CPU resource as the capacity. In addition, we use network and memory consumption as sub-constraints. The bin-packing problem is known to be NP-complete [14], however, many efficient heuristics have been proposed to solve it. For FUGU we implemented two well-known bin-packing heuristics, FirstFit and BestFit.

4 Latency-aware Elastic Scaling

As illustrated in the previous section, a set of operators needs to be moved between hosts in the system in response to a scaling decision. This movement has to ensure that no information is lost. This condition requires the usage of an operator movement protocol [15], which guarantees that an operator and its state are moved together. For each operator to be moved, the protocol used first pauses the processing of the predecessor operators, which causes all newly arriving events to be enqueued. Then, a new instance of the operator is created and the operator state is moved. When the state movement is completed, the predecessor operator is restarted. As the processing of the enqueued events at the predecessor operator is delayed, a latency peak can be observed. Existing scaling strategies [5, 7] optimize the scaling decision based only on the CPU load moved or the state size moved and ignore the resulting latency peak.

In FUGU we deal with this problem by introducing a model to estimate the latency peak created by an operator movement. The model (see Figure 3) estimates the queue length $ql(op_{pred}, t)$ of the predecessor operator created during the movement, which determines the observed latency peak. As input for this estimation two major factors are considered: workload characteristics such as the current input rate *inputRate* of the predecessor operator op_{pred} and the movement time *moveTime* of the moved operator op_{moved}. The major challenge is that the movement time of an operator depends on multiple factors such as the state size, the operator type, and the current host load [8]. Therefore, we collect a set of samples of these characteristics together with the corresponding latency peak online. The samples are clustered based on these factors, and for a new operator movement, the cluster of samples with the highest similarity is identified. That subset of samples is used to estimate the movement time for new movements.

This estimation model is used to extend the *Operator Selection* algorithm presented in Section 3. Our system allows the user to define a latency threshold, which is considered when the scaling decisions are computed. We classify scaling decisions into two categories (1) mandatory and (2) optional movements. All scaling decisions necessary to avoid an overload of the system are mandatory scaling decisions. The release of a host due to underload is an optional scaling decision. Any optional scaling decision can be postponed or canceled in case the estimated latency peak would be too high. Thereby, unnecessary violations of the latency constraints can be avoided. The operator selection for an overloaded host is modified to identify a set of candidate solutions whose summed operator loads are above a certain CPU threshold. Among all candidates, the solution with the minimum estimated latency peak is chosen. In addition, the way in which the system handles CPU underload is changed. Normally, if the system detects a system underload, the host with the minimal CPU load is released and all operators running on this host are moved to other hosts. In our latency-aware elastic scaling the system releases the host, that minimizes the estimated latency peak for moving all operators on the host. If no host with

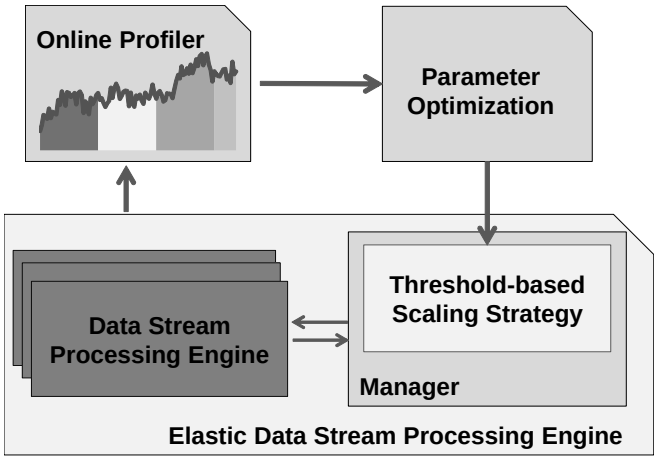


Figure 4: Architecture of Online Parameter Optimization

an estimated latency peak below the user-defined threshold exists, only a subset of the operators on the host with the smallest estimated latency peak is moved.

5 Online Parameter Optimization

The configuration of a threshold-based scaling strategy is very difficult for an inexperienced user, as he typically has a limited understanding of the system and the influence of the possible parameter settings on system performance. Therefore, we introduce an online parameter optimization approach, which chooses these parameter settings automatically based on current workload characteristics.

Online parameter optimization adds two new components to the existing elastic scaling data stream processing engine (see Figure 4): a parameter optimization component and an online profiler. We identified a set of six major parameters for our system, such as utilization thresholds and the bin packing method used, that primarily influence the scaling behaviour of the system and describe the parameter configuration of the scaling strategy. For each parameter, we determine a reasonable domain. In total, 720,000 parameter configurations exist [10].

Our optimization component automatically discovers a good parameter configuration based on a short-term utilization history of the running system. In this approach we use a cost function [10], that models the influence of these parameters on the scaling behaviour. Threshold-based scaling deterministically derives a scaling decision for a given operator assignment of operator instances, current utilization values and a setting of the mentioned parameters. For the cost function, we input a time series of utilization values and assignments and get as a result a set of scaling decisions for the given parameter settings. From these scaling decisions, we can determine both the amount of resources used and the latency peaks created by the scaling decisions.

We determine possible parameter configurations using an improved random search algorithm [17] and identify a configuration with a good trade-off between resources used and latency based on the short term utilization history. Finally, we compare these results with the results of the current parameter configuration of the system and adapt the parameters, if a configuration with less host use and a less or equal number of moved operators was found.

The previously mentioned online profiler determines the frequency of triggering the parameter optimization. It monitors changes of the workload pattern based on the overall CPU load using an adaptive window [2]. The system periodically adds a new value to the window. If this new value is similar to the existing values, it is simply appended at the head of the window. If a significant change is detected, values from the tail are deleted until all values in the window are similar again. Parameter optimization is triggered each time a change is detected. The

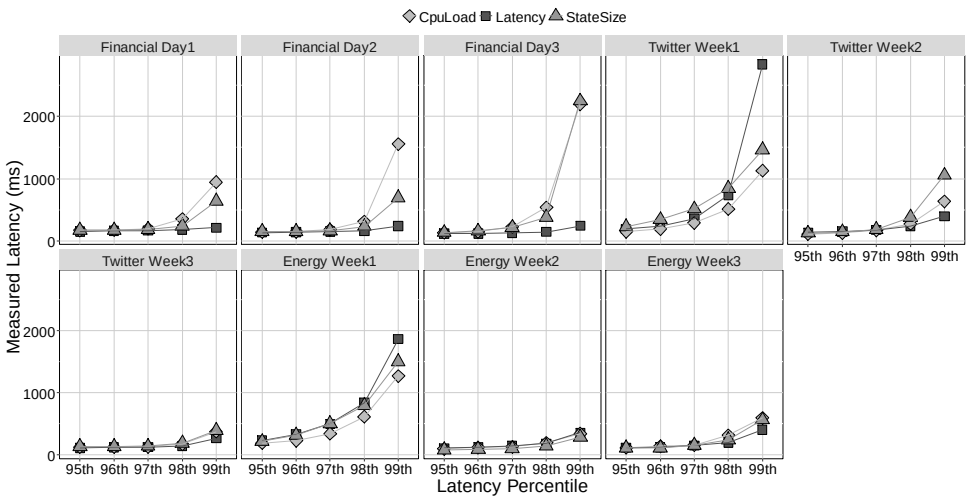


Figure 5: Latency Results for Different Operator Selection Strategies

length of the window also specifies the length of the short-term history of current load characteristics to use for the online parameter optimization. This approach allows adaptively identifying a good parameter setting for the system.

6 Evaluation

We implemented both latency-aware elastic scaling and online parameter optimization as extensions of FUGU. During an evaluation with three real-world scenarios, we tried to answer two major questions:

1. Does latency-aware elastic scaling improve latency compared to other operator selection strategies?
2. Does online parameter optimization provide a good trade-off between system utilization and query processing latency, thus relieving the user of the burden of manually configuring the parameters?

In the evaluation we use a private, shared cloud environment with one master node and up to twelve workers. We run three different real-world scenarios [10]: a scenario with financial data, one with Twitter messages, and a third with smart meter measurements. For each case we use three different traces, which make up in total nine workloads. Each experiment lasts for 90 minutes, where end to end latency and host utilization are measured roughly every five seconds. For a single measurement point, we use the average utilization of all hosts and average latency of all queries to quantify the utilization of the system and the quality of service, respectively.

6.1 Latency-aware Elastic Scaling

We compare our latency-aware operator selection strategy with two alternative operator selection strategies [8]: CPULoad and StateSize. The CPULoad strategy selects operators to move in a way that minimize the total CPU load moved. In contrast, the StateSize strategy minimizes the total state size moved, when moving operators between hosts. For each strategy we evaluated six different thresholds and average the results to avoid any influence of the chosen threshold configurations on the results. We present the resulting latency in Figure 5 and the measured utilization values in Figure 6.

For the latency results we show the 95th, 96th, 97th, 98th, and 99th percentiles of all measurements. The measured results for the 95th, 96th and 97th percentile for the three strategies differ only very marginally, which

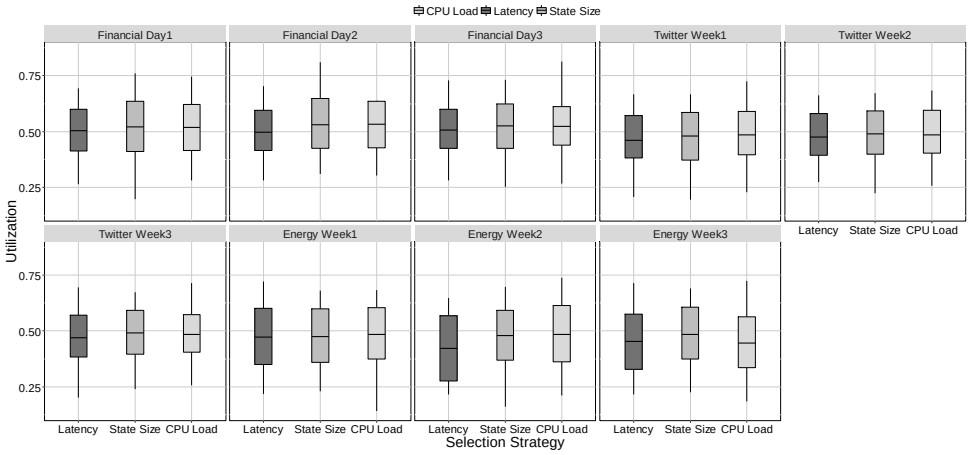


Figure 6: Utilization Results for Different Operator Selection Strategies

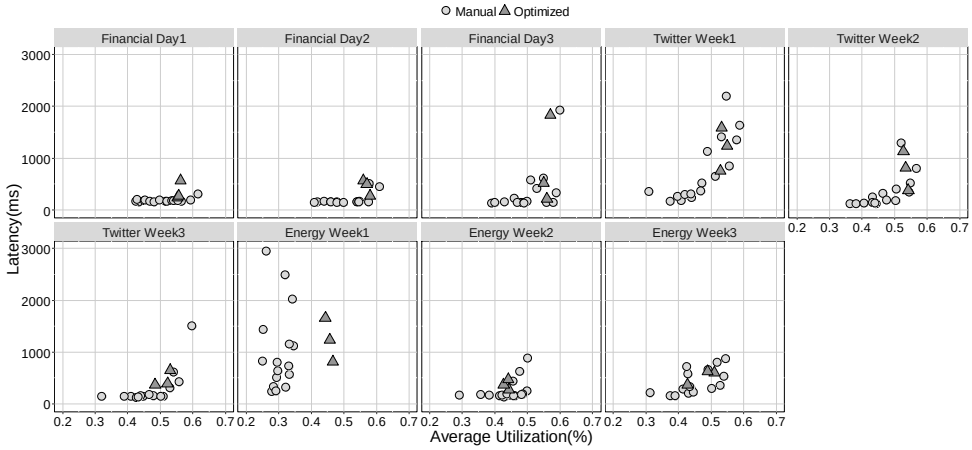


Figure 7: Comparison of Online Parameter Optimization and Manually Tuned Thresholds

demonstrates that the operator selection strategy used influences only the measured latency peaks. The latency-aware operator selection we presented outperforms the two other strategies in seven out of nine scenarios. On average, over all nine scenarios, the latency-aware selection strategy has a 18% and 19% lower 98th percentile latency than the CPULoad and StateSize strategies, respectively. For the 99th percentile our strategy's latency is 16% and 22% lower than for the CPULoad or StateSize strategies.

Figure 6 shows a comparison of the utilization results, where we present a comparison of the average utilization for the three different strategies using a boxplot. The operator selection strategy used has only a small influence on the utilization achieved. The latency-aware strategy has only a two percent point smaller utilization than the CPULoad or the StateSize strategy.

6.2 Online Parameter Optimization

As a baseline for online parameter optimization, we manually tuned the thresholds. We evaluated 16 different threshold configurations and compared the results achieved for our parameter optimization over three different runs. We show the average node utilization and the 98th percentile of the averaged latency in Figure 7.

The results show a significant variance in both the average utilization and the latency for different configurations: the minimal and maximal utilization differ by 20 percentage points. From the 16 measurements, we

extract the average to estimate the results that an inexperienced user might achieve. Online parameter optimization shows a five percentage point better utilization with only a slight increase of the 98th percentile latency (231 ms) averaged over all scenarios.

Subsequently, we selected the three best configurations per workload and compared them to the configuration derived by online parameter optimization. Online parameter optimization shows comparable utilization results (0.02% worse) and again only a small increase of the 98th percentile latency (330 ms).

From these results we conclude that our online parameter optimization provides a good trade-off between system utilization and query processing latency. It also removes the burden of manually choosing the thresholds from the user.

7 Summary

Elastic scaling allows a data stream processing system to react to unexpected load spikes and reduce the amount of idling resources in the system. Although several authors proposed different approaches for elastic scaling of a data stream processing system, these systems require a manual tuning of the thresholds used, which is an error-prone task and requires detailed knowledge about the workload.

In this paper we introduce a model to estimate the latency peak created by a scaling decision and present an approach to minimize that peak accordingly. In addition, we propose an online parameter optimization approach, which automatically adjusts the scaling strategy of an elastic scaling data stream processing system. Our system minimizes the number of hosts used and at the same time keeps the number of latency peaks low. Both approaches have been evaluated in the context of several real-world use cases and have demonstrated their applicability for such use cases.

References

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina *et al.*, "The Design of the Borealis Stream Processing Engine," in *CIDR '05: Proceedings of the Second Biennial Conference on Innovative Data Systems Research*, 2005, pp. 277–289.

[2] A. Bifet and R. Gavaldà, "Learning from Time-Changing Data with Adaptive Windowing," in *SDM 2007: Proceedings of the Seventh SIAM International Conference on Data Mining*, 2007, pp. 443–448.

[3] E. G. Coffman Jr, M. R. Garey, and D. S. Johnson, "Approximation Algorithms for Bin Packing: A Survey," in *Approximation algorithms for NP-hard problems*. PWS Publishing Co., 1996, pp. 46–93.

[4] M. Ead, H. Herodotou, A. Aboulnaga, and S. Babu, "PStorM: Profile Storage and Matching for Feedback-Based Tuning of MapReduce Jobs," in *EDBT '14: Proceedings of the* 17^{th} *International Conference on Extending Database Technology*, 2014, pp. 1–12.

[5] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Integrating Scale Out and Fault Tolerance in Stream Processing Using Operator State Management," in *SIGMOD '13: Proceedings of the SIGMOD International Conference on Management of Data*. ACM, 2013, pp. 725–736.

[6] H. Ghanbari, B. Simmons, M. Litoiu, and G. Iszlai, "Exploring Alternative Approaches to Implement an Elasticity Policy," in *CLOUD '11: Proceedings of the IEEE International Conference on Cloud Computing*. IEEE, 2011, pp. 716–723.

[7] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez, "StreamCloud: An Elastic and Scalable Data Streaming System," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 23, no. 12, pp. 2351–2365, 2012.

[8] T. Heinze, Z. Jerzak, G. Hackenbroich, and C. Fetzer, "Latency-aware Elastic Scaling for Distributed Data Stream Processing Systems," in *DEBS '14: Proceedings of the* 8^{th} *ACM International Conference on Distributed Event-Based Systems*. ACM, 2014, pp. 13–22.

[9] T. Heinze, V. Pappalardo, Z. Jerzak, and C. Fetzer, "Auto-scaling Techniques for Elastic Data Stream Processing," in

ICDEW '14: Workshops Proceedings of the 30^{th} *International Conference on Data Engineering Workshops*. IEEE, 2014, pp. 296–302.

[10] T. Heinze, L. Roediger, A. Meister, Y. Ji, Z. Jerzak, and C. Fetzer, "Online Parameter Optimization for Elastic Data Stream Processing," in *SoCC '15: Proceedings of the ACM Symposium on Cloud Computing 2015*. ACM, 2015, pp. 276–287.

[11] N. R. Herbst, S. Kounev, and R. Reussner, "Elasticity in Cloud Computing: What It Is, and What It Is Not," in *ICAC '13: Proceedings of the* 10^{th} *International Conference on Autonomic Computing*, 2013, pp. 23–27.

[12] H. Herodotou and S. Babu, "Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs," *Proceedings of the VLDB Endowment*, vol. 4, no. 11, pp. 1111–1122, 2011.

[13] T. Lorido-Botrán, J. Miguel-Alonso, and J. A. Lozano, "Auto-scaling Techniques for Elastic Applications in Cloud Environments," *Department of Computer Architecture and Technology, University of Basque Country, Tech. Rep. EHU-KAT-IK-09*, vol. 12, 2012.

[14] S. Martello and P. Toth, "Algorithms for Knapsack Problems," *Surveys in Combinatorial Optimization*, vol. 31, pp. 213–258, 1987.

[15] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin, "Flux: An Adaptive Partitioning Operator for Continuous Query Systems," in *ICDE '03: Proceedings of the* 19^{th} *IEEE International Conference on Data Engineering*. IEEE, 2003, pp. 25–36.

[16] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, "Cloudscale: Elastic Resource Scaling for Multi-tenant Cloud Systems," in *SoCC '11: Proceedings of the second ACM Annual Symposium on Cloud Computing*. ACM, 2011, pp. 1–14.

[17] T. Ye and S. Kalyanaraman, "A Recursive Random Search Algorithm for Large-scale Network Parameter Configuration," in *SIGMETRICS '03: Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. ACM, 2003, pp. 196–205.

Exploiting Sharing Opportunities for Real-time Complex Event Analytics

Elke A. Rundensteiner[1], Olga Poppe[1], Chuan Lei[2], Medhabi Ray[3], Lei Cao[4], Yingmei Qi[5], Mo Liu[6], and Di Wang[7]

[1]Worcester Polytechnic Institute, 100 Institute Road, Worcester, MA 01609

[2]NEC Labs America, 10080 N Wolfe Rd, Cupertino, CA 95014

[3]Microsoft Corporation, 205 108th Ave. NE, Bellevue, WA 98004

[4]IBM T.J.Watson Research Center, 1101 Route 134 Kitchawan Rd, Yorktown Heights, NY 10598

[5]Google, 601 N 34th St, Seattle, WA 98103

[6]Sybase Corporation, 1 Sybase Drive Dublin, CA 94568

[7]Facebook, 1730 Minor Ave, Seattle, WA 98101

{rundenst,opoppe}@cs.wpi.edu, chuan@nec-labs.com, meray@microsoft.com, caolei@us.ibm.com, ymqi@google.com, mo.liu@sybase.com, wangdi@fb.com

Abstract

Complex event analytics systems continuously evaluate massive workloads of pattern queries on high volume event streams to detect and extract complex events of interest to the application. Such time-critical stream-based applications range from real-time fraud detection to personalized health monitoring. Achieving near real-time system responsiveness when processing these workloads composed of complex event pattern queries is their main challenge. In this article, we first review several unique optimization opportunities that we have identified for complex event analytics. We then introduce a family of optimization strategies that consider event correlation over time to maximally leverage sharing opportunities in event pattern detection and aggregation. Lastly, we describe the event-stream transaction model we designed to ensure high performance shared pattern processing on modern multi-core architectures.

1 Introduction

Many streaming systems from sensor networks to financial transaction processing generate high-volume, high-velocity event streams. These events have many dimensions (such as time, location, dollar amount). Each dimension may be hierarchical in nature (such as time measured in years, months, days and so on). In many monitoring applications, it is imperative that a huge workload of expressive event-pattern queries analyze these event streams to detect complex event patterns, aggregate trends and derive actionable insights in near real time.

Motivating Example. Consider an evacuation system where RFID technology is used to track the mass movement of people and goods during natural disasters. Terabytes of RFID data could be generated by such a system. Facing this huge volume of data, an emergency management system must detect and aggregate complex

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

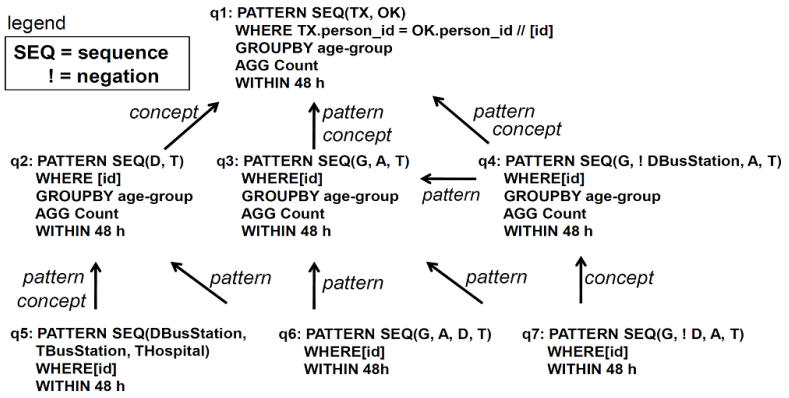


Figure 1: Event pattern queries supporting emergency management at different levels of abstraction [36]

event patterns across multiple dimensions at different granularities in real time. For example, the emergency personnel may monitor people movement as well as traffic patterns of needed resources (such as medicine, food, and blankets) at different levels of abstraction (e.g., bus station, Austin, Texas). Consider Figure 1, where during a hurricane the federal government may monitor people fleeing from Texas to Oklahoma for global resource distribution planning (query q_1); while the local authorities in Dallas may focus on people movement starting from the Dallas bus station, traveling through the Tulsa bus station, and ending in the Tulsa hospital, to determine the need for additional means of transportation (query q_5).

These event queries tend to contain similar or sometimes even identical sub-patterns. Hence, techniques that exploit their similarities for optimization can save computational resources and improve the system responsiveness. Many event-stream-based applications from online advertising, click-stream analytics, social network services to financial fraud detection all feature these huge workloads composed of similar event queries. Thus, performance gains due to leveraging such customized event-optimization technology for shared computations among such event queries could have a tremendous benefit across this wide range of applications.

Challenges. To design an effective event-analytics infrastructure, we must tackle the following challenges.

Rich Application Semantics. Streaming applications have rich semantics. This semantics involves event-sequence construction of arbitrary length; event conjunction, disjunction and negation; expressive predicates; time-, count- or predicate-based windows; event-pattern grouping and aggregation. Therefore, we must develop efficient processing techniques for a large workload of such expressive event-pattern queries.

Real-time System Responsiveness. We target time-critical applications in which milliseconds can make a difference in decision making. Thus, event query computations should be shared or even completely eliminated if we can do so without compromising result quality. These computational savings speed up the decision-making process, improve resource allocation, reduce environmental pollution and even save human lives. However, sharing is not always beneficial. Even if two event queries syntactically share a sub-pattern, the actual sets of matches of these queries may not overlap at runtime [43]. Sharing computations across such event queries may result in negligible performance gain at the cost of adding significant synchronization overhead. Fortunately, while the number of identical sub-patterns in a query workload at times may be limited, other hierarchical relationships among event queries can be exploited for optimization [36].

Correct Event-Stream Execution. Sharing common or similar sub-patterns between several event queries makes these queries interdependent. Indeed, the shared sub-pattern must be computed before the queries that share it. An efficient runtime execution infrastructure should process a workload of such interdependent event queries while leveraging the concurrent execution capabilities of modern multi-core machines. Thus, a concurrency control mechanism is needed that ensures correct concurrent stream processing. Furthermore, if we strive to delay or even skip event-sequence construction while computing event-sequence aggregations, we must assure that no potential event sequence matches are missed under the premise that aggregation is computed on-the-fly

and events are instantly pruned upon their aggregation.

State of the Art. Multi-query optimization is an established technology in relational databases [8, 11, 21]. Unfortunately, these techniques cannot be applied directly to shared event-query processing because streaming data is continuously under flux. Thus, the data-driven approach of event processing may trigger the pattern matching process to be spawned in diverse orders based on the arrival of events. The nature of continuous event-stream-processing systems stands in contrast to the traditional static processing frameworks where all data is given a priori and execution can be fully orchestrated.

Many complex-event-processing systems do not exploit sharing opportunities across the event-query workload [7, 13, 35, 50]. While XML-filtering approaches leverage some sharing opportunities, such as shared prefix-matching, they disregard other sharing opportunities [14, 15]. While the approaches proposed in [2, 47] share sub-patterns in the distributed context, they do not provide any guarantee to produce a globally optimal plan for multiple event queries. Several approaches [12, 38] are devoted to the optimization of multiple event queries. However, these approaches neglect inter-query event correlations and thus may miss optimization opportunities. Existing solutions to processing multiple concurrent event queries over different abstraction levels, online event pattern aggregation, and general stream transaction models are either missing or limited by having assumptions that do not hold in our event context.

Key Innovations. In this article, we present an overview of four orthogonal innovations for the optimization of complex event analytics developed by members at WPI and collaborators. Each of these innovations leverages shared processing opportunities unique to event analytics. These innovations include:

1) *Event-Sequence Pattern Sharing.* We analyze the benefit of sharing event-sequence construction considering both intra- and inter-event pattern correlations over time [43]. We show that the problem of optimizing a workload of event-sequence patterns to minimize its CPU processing time is equivalent to the NP-hard Minimum Substring Cover problem [28]. This result then leads us to apply the polynomial-time approximate Local-Ratio algorithm to our problem with proven acceptable bounds on optimality [28].

2) *Hierarchical-Event Pattern Sharing.* Event queries, even if not identical, can still be related to each other in terms of both *concept abstractions* and *pattern refinements*. These relations open up unique opportunities for shared processing of similar event-sequence patterns. This pattern similarity leads us to establish the E-Cube hierarchy composed of event queries at different levels of abstraction [36]. Our efficient processing strategies evaluate all event patterns in the workload in a specific order to reuse their intermediate results.

3) *Shared Event-Pattern Aggregation.* Since all event sequences are discarded once their aggregation is computed, we aggregate event-sequences without constructing them. We achieve such on-the-fly event-sequence aggregation by dynamically maintaining a prefix counter and instantly discarding events after their aggregation. Thus, we reduce the event-sequence aggregation costs from polynomial to linear [42]. This optimization technique is exploited while sharing the aggregation of common sub-patterns in the query workload.

4) *Stream Transaction Model.* Given concurrent accesses and updates to shared event pattern matches, we avoid race conditions by designing an appropriate concurrency-control mechanism. To this end, we introduce our stream transaction model [49]. Since the classical Strict-Two-Phase-Locking algorithm incurs a large synchronization delay due to its rigorous order preservation, we introduce event-centric scheduling methods for real-time streaming applications to maximize concurrent execution.

Our thorough experimental studies using both synthetic and real data sets reveal that these optimization techniques achieve several orders of magnitude performance gain compared to state-of-the-art solutions [36, 42, 43, 49]. Furthermore, our technology was tested out successfully in a real-world setting. In particular, we installed our complex event analytics software in the intensive care units at UMASS Memorial Hospital under leadership of Dr. Ellison, head of infection control at UMASS. We analyzed the results of a clinical evaluation of this technology for improving health-care hygiene [16, 17, 49].

Outline. This article is organized as follows. We start with our event-analytics model in Section 2. Afterwards, we present our sharing techniques for sequence patterns in Section 3 and abstraction patterns in Section 4. Section 5 is devoted to the shared processing of aggregations over event patterns. We propose our stream trans-

action model in Section 6. Related work is discussed in Section 7, while Section 6 concludes this article.

2 Event-Analytics Model

Event Data Model. *Time* is represented by a linearly ordered set of time points $(\mathbb{T}, \leq)$, where $\mathbb{T} \subseteq \mathbb{Q}^+$ the non-negative rational numbers. An *event* is a message indicating that something of interest happened in the real world. An event e has an *occurrence time* $e.time \in \mathbb{T}$ assigned by the event source. Each event e belongs to a particular *event type* E, denoted $e.type = E$. An event type E is described by a *schema* that specifies the set of *event attributes* and the domains of their values. Events are sent by event producers (e.g., RFID tag readers) to event consumers (e.g., an emergency management system) on *event streams*.

Event Pattern Query. Event queries in our event-analytics model consist of clauses similar to other event query languages, for example, SASE+ [1, 50]. These clauses are the following:

Window (`WITHIN` clause) specifies the portion of the potentially unbounded input event stream to be processed by one event-query invocation. Our language supports both fixed-length time or count-based tumbling or sliding windows [3, 33] and variable-length predicate-based windows [19].

Pattern (`PATTERN` clause) defines the structure of event occurrences in the input event stream that must match in order for a complex event to be detected [36, 50]. Let E be an event type, P and P' be event patterns. Then, an event pattern is defined by a composition of operators including event occurrence of type E, event-pattern non-occurrence $!P$, event-pattern conjunction $AND(P, P')$ and disjunction $OR(P, P')$, event sequence of fixed length $SEQ(P, P')$, and event pattern of arbitrary length $P+$.

Predicates (`WHERE` clause) impose additional constraints on event-pattern matches. These constraints are boolean expressions composed of arithmetical and comparison operators on event attribute values and constants.

Grouping and Aggregation (`GROUPBY` and `AGG` clauses) can be applied to event pattern matches. Event pattern matches are grouped, for instance, by the attribute values of matched events. Our language supports common aggregation functions such as *count, sum, avg, min* and *max*.

For example, query q_1 in Figure 1 counts the number of people (`AGG Count`) who fled from Texas to Oklahoma (`PATTERN SEQ(TX, OK) WHERE TX.person_id = OK.person_id`) within 48 hours (`WITHIN 48 h`) per age group (`GROUPBY age-group`). Other event queries in Figure 1 behave similarly.

3 Event-Sequence Pattern Sharing

Event Correlations. We target the efficient detection of event-sequence patterns in data streams via shared concurrent pattern execution [43]. Our solution takes as input a set of pattern queries. It estimates the benefit of sharing the computation of sub-patterns based on the time-ordering across events and the inter-query event correlation hidden in the event streams. Sharing an event sub-pattern between multiple queries is not always beneficial. It may even cause more harm than good by incurring unnecessary concurrency-control overhead. Based on this observation, we design a lightweight yet effective method for estimating the *time-sensitive co-occurrence properties* of event streams to accurately capture the benefit of sharing event patterns. The proposed method takes the following two types of event correlations into consideration: (1) *Intra-query event correlation* estimates the number of event sub-pattern matches per time interval, e.g., the percentage of events of type A that follow an event of type B. This ratio estimates the number of matches produced by a single event pattern. (2) *Inter-query event correlation* estimates the sharing potential across multiple event patterns as the ratio of the number of shared sub-pattern matches to the total number of matches.

Benefit of Event Pattern Sharing. We analyze the degree of sharing of sub-pattern matches in a *sample time period* by tracking the number of matches for a sub-pattern within this time period. This process is periodically repeated to provide the up-to-date statistics. Figure 2 shows that the number of matches of a sub-pattern $SP = SEQ(A, B)$ produced by the two patterns P_1 and P_2 may vary over time. Consequently, the number of

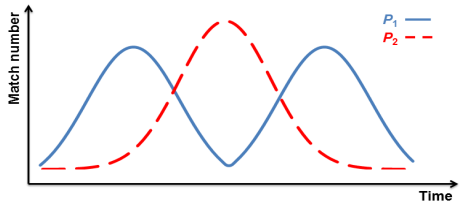


Figure 2: Distribution of event pattern matches over time [43]

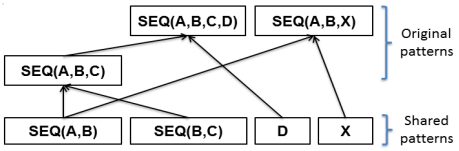


Figure 3: Shared plan of event-sequence patterns [43]

pattern matches for SP that can be shared across P_1 and P_2 also varies over time. This observation leads us to two insights essential for the sub-pattern sharing task: (1) The crests and troughs of P_1 and P_2 never align in this example, even though their average cardinalities over time happen to be similar. Hence, the inter-query correlation between P_1 and P_2 is low. Thus, sharing this sub-pattern between P_1 and P_2 may cause more harm than good due to concurrency control overhead. (2) Even if the cardinalities of the sub-pattern matches happen to be the same for two patterns over time, the match re-use is still not guaranteed since the sub-patterns may not be common for these patterns *at the event-instance level*. Indeed, the benefit of sub-pattern sharing depends on the occurrences of the other sub-patterns in these patterns. In short, cardinality alone is no reliable indicator since individual matches may be non-overlapping. Based on these observations, we design a cost model that accurately estimates the ratio of the cost to compute matches of a *shared* sub-pattern SP for all its parent patterns to the cost of producing all matches of the sub-pattern SP for each parent pattern *separately* as the *redundancy-ratio score*. The lower the score, the higher the benefit of sharing this event sub-pattern.

Shared Event Pattern Plan. Leveraging this redundancy ratio scoring model, we can now tackle the problem of *sub-pattern sharing optimization*. Namely, we aim to find a subset of sub-patterns such that all queries in the given workload share the processing of this subset and the redundancy ratio of this subset is minimal compared to all other possible subsets. We can show that this problem is equivalent to the Minimum Substring Cover problem [43]. Thus, our optimizer can leverage the polynomial-time approximate Local-Ratio algorithm for the Minimum Substring Cover problem to produce the set of sub-patterns to share [28]. Once the set of event sub-patterns is selected, our optimizer iteratively builds up a shared-pattern plan for the workload in a bottom-up fashion. This shared-pattern plan is a graph in which each node is a (sub-)pattern. For example, the original patterns $SEQ(A,B,C)$, $SEQ(A,B,C,D)$, and $SEQ(A,B,X)$ are decomposed into the shared sub-patterns $SEQ(A,B)$, $SEQ(B,C)$, D and X (Figure 3).

4 Hierarchical Event Pattern Sharing

Event-Sequence-Pattern Abstraction Hierarchy. As motivated in Section 1, the number of event-sequence patterns that have syntactically identical sub-patterns (as assumed in Section 3) may be limited. Thus, we now explore effective sharing strategies that also consider hierarchical event queries. This hierarchy is essential for performance optimization in multi-query evaluation since it provides a blueprint for shared online event-query matching. We differentiate between the *concept* and the *pattern hierarchy* [23, 36].

A *concept hierarchy* (Figure 4) is used to summarize information at different levels of abstraction. Many dimensions (e.g., time, location, object type) are hierarchical in nature and thus create a concept hierarchy of

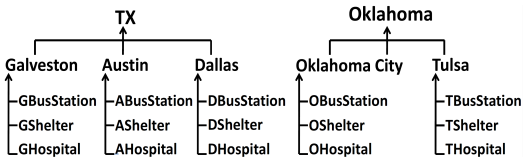


Figure 4: Concept hierarchy of primitive event types [36]

the corresponding event types. Event concept hierarchies for primitive event types are predefined by system administrators based on domain knowledge. An event concept hierarchy is a tree with the most-specific event types as leafs and more-general event types as inner nodes. An event type E_k that is a descendant of an event type E_j is at a finer level of abstraction than E_j, denoted by $E_k <_c E_j$. The non-existence (existence) of a negative (positive) event type at a coarser (finer) concept level enforces more constraints as compared to a negative (positive) event type at a finer (coarser) concept level. In Figure 1, the query q_1 is at a coarser concept level than the query q_2 because TX $>_c$ D and OK $>_c$ T. The query q_4 is at a coarser concept level than the query q_7 since the negative type D in q_4 is coarser than DBusStation in q_7 (D $>_c$ DBusStation).

A *pattern hierarchy* is defined as follows: A query q_k can be drilled-down to a finer-level query q_j by inserting additional event types into the pattern of q_k, denoted by $q_k >_p q_j$. For example, q_6 is at a finer level than q_3 because q_3 enforces the existence of less event types and sequential event relationships than q_6 (Figure 1).

An *E-Cube hierarchy* is a directed acyclic graph where each node is a query q_i and each edge corresponds to a *pairwise refinement relationship* between two queries in terms of either concept or pattern refinement. Each directed edge (q_i, q_j) is labeled with either the label "concept" if $q_i <_c q_j$, "pattern" if $q_i <_p q_j$, or both to indicate the refinement relationship between the queries [25]. Figure 1 shows an example E-Cube hierarchy.

Advanced Event Analytics via Event Pattern Exploration. We now illustrate that a concept or a pattern can be drilled-down into or rolled-up such that we can navigate from one node (with its respective matches) to another node in the E-Cube hierarchy by skipping, adding or replacing sub-patterns. For example in Figure 1, we apply a pattern-drill-down operation on $q_3 = SEQ(G, A, T)$ by adding a $!D$ constraint and get $q_7 = SEQ(G, !D, A, T)$. Similarly, we apply a concept-roll-up operation on $q_2 = SEQ(D, T)$ by one level from Dallas to Texas and from Tulsa to Oklahoma and get $q_1 = SEQ(TX, OK)$.

Optimal E-Cube Evaluation. This E-Cube hierarchy represents the sharing plan for all hierarchical event-pattern queries. For each query q in the E-Cube hierarchy, we have a choice between: (1) Computing q independently from other queries, (2) Conditionally computing q from one of its ancestors or (3) Conditionally computing q from one of its descendants. Our cost model [24, 36] estimates the cost of each option and assigns this cost as a weight on each corresponding directed edge between a pair of queries. Having this directed weighted graph, our goal is to determine an optimal query-evaluation plan ordering, i.e., an ordering of sub-patterns with minimal total execution costs. We show that we can reduce this problem to the Minimal Spanning Tree problem. This reduction allows us to apply the Gabow algorithm [18] to achieve our goal.

5 Shared Event Pattern Aggregation

Online Event Pattern Aggregation. The computation of aggregation over event sequences such as in Figure 1 in our motivating example opens unique opportunities as we illustrate next. We compute an *event sequence count* without ever constructing the actual event sequences. Such online event sequence count can be computed correctly by continuously updating a *prefix counter* in *constant time* upon the arrival of each new event such that a new event, once processed, can be discarded instantly [42].

For example, event sequences matched by the pattern $SEQ(A, B, C)$ are counted in Figure 5. When the events shown on top arrive, the prefix counter for the patterns shown on the left are updated as follows. When the event b_2 arrives, 3 new sub-sequences (a_1, b_2), (a_2, b_2) and (a_3, b_2) are formed using previously arrived events

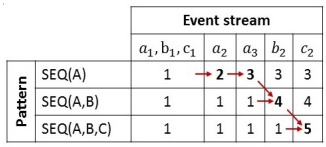


Figure 5: Prefix counters

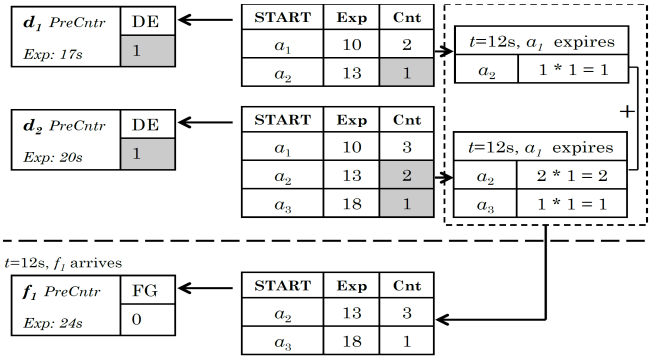


Figure 6: Prefix counters for snapshots [42]

a_1, a_2 and a_3. Thus, the total count of event sequences matched by the pattern $SEQ(A, B)$ is now 4, including the 3 newly formed sequences and a_1, b_1 that we had found before. We observe that when b_2 arrives, we can obtain the count of the event sequences by adding two counts: (1) The count of the sub-pattern $SEQ(A)$ where b_2 would be appended to the matches of this sub-pattern, and (2) The count of the sub-pattern $SEQ(A, B)$. We notice that the actual event sequences do not have to be constructed to update the count. Omitting event sequence construction reduces the aggregation computation costs from polynomial to linear [42].

Other aggregation functions can be supported analogously. For example for *sum*, we maintain an extra *sum* field in each prefix counter on the event type the attribute value of which is to be summed. When an event arrives and causes an update of a count, its respective *sum* field will also be updated.

Negation. Negation requires the non-occurrence of events of the negated event types at certain positions within the event sequences. The arrival of such events can invalidate potential matches. Therefore, when an event arrives whose occurrence is negated in the query, we simply reset the corresponding prefix counter.

Predicates. Local predicates impose constraints on the attribute values of events, for example, *age>20*. Such predicates can filter events before they are aggregated. Equivalence predicates correlate events in a sequence [50]. For example, to monitor people's movement during an emergency, we require the same value of the person identifier attribute in all events contributing to one event sequence matched by the queries in Figure 1. Such predicates partition the event stream into several sub-streams. This partitioning then allows us to compute aggregation separately for each partition using the above described principles.

Sliding Window. When the window slides, multiple events expire and multiple new events become relevant. One expired event might invalidate an arbitrary number of event sequences and thus require an update of the aggregation results. However, the expiration of most events has no affect on the aggregated value. We determine the minimum subset of events whose expiration could indeed affect the aggregation result in [42].

Aggregation Sharing. Shared aggregation of single events is well-studied [30, 34, 51]. However, shared aggregation of event sequences poses new challenges such as pushing the aggregation through the sequence-construction process to save the resources. We could consider the sharing of common sub-sequences between multiple similar queries (Section 3). To minimize the CPU costs, event queries that have common sub-patterns are chopped into sub-patterns to aggregate them separately using the highly scalable techniques introduced above. We then stitch these partial results together to get the final results for the original pattern requests.

S2PL		Requested	
		Read	Write
Held	Read		X
	Write	X	X

LWM		Requested	
		Read	Write
Held	Read		
	Write	may be	X

Figure 7: Lock incompatibility [49]

However, events expire over time. Let $\#s_1$ and $\#s_2$ be the counts of the sub-patterns s_1 and s_2 respectively. When a triggering event of s_2 arrives, $\#s_1$ might become invalid due to the expiration of some of the matches aggregated by s_1. This situation risks causing erroneous aggregation results. To support event expiration, we maintain snapshots for each sub-pattern. The idea is the following: For each first event in a sequence, we store the expiration time point and the number of sequences that start with this event in the snapshot. When a first event expires, we ignore its respective count, since all the sequences it participates in expire too. For example, assume the pattern $SEQ(A, B, C, D, E, F, G)$ is chopped into 3 sub-patterns $s_1 = SEQ(A, B, C)$, $s_2 = SEQ(D, E)$, and $s_3 = SEQ(F, G)$. When the event f_1 arrives at time $t = 12s$, we consider only non-expired counts (they are highlighted in Figure 6). First, we multiply the count of each match of the sub-pattern $SEQ(D, E)$ with the counts in its respective snapshot of the sub-pattern $SEQ(A, B, C)$. Second, we sum up the counts for the same first event across all matches (a_2 in our example). Third, we store the resulting counts in the snapshot of the sub-pattern $SEQ(A, B, C, D, E)$ for future reference.

6 Facilitating Concurrency for Efficient Complex Event Analytics

Stream Transactions. In prior sections we have illustrated various strategies to detect shared sub-patterns and then to reuse their partial results. To achieve high system responsiveness, we leverage modern multi-threaded solutions on multi-core architectures instead of forcing all computation to proceed sequentially. Thus, to avoid race conditions, read and write operations on shared storage (e.g., results of a shared sub-pattern) must be synchronized. Traditional transaction models should be reexamined since: (1) Events are not static, rather they continuously arrive on streams. (2) Event queries are standing, they continuously monitor these event streams [6]. (3) Neither abort nor restart of a transaction at a later time point (used in MVCC [5]) may be acceptable for externally visible output or actions typical for real-time streaming applications [45, 49].

Towards Event-Stream Transactions. Here we briefly introduce an appropriate notion of transactions in the context of event streams, which we henceforth refer to as stream transaction. A *stream transaction* is a sequence of all system changes that are triggered by a single input event. Two operations are called *conflicting* if they are performed on the same data item and at least one of them is a Write. An algorithm for scheduling operations on a shared data item performed by event queries is then considered to be *correct* if every schedule produced by the algorithm processes conflicting operations in order by their application time stamps.

Let us now examine one simple transaction model in this context. Similarly to the classical MVCC [5], the historic records of each shared data item could be maintained. We then define the *low-water-mark* as the oldest time stamp among all the time stamps of Write locks. A Read lock is granted if all Writes earlier than the Read have completed. A Write lock is granted if it is the oldest Write lock among all Write locks on this data item. Given this lock-granting strategy, we can relax the lock incompatibility in two ways (Figure 7): (1) A Read lock does not block acquiring a Write lock since the previous version is read while a new one is created. (2) A Write lock does not necessarily block a Read lock if earlier versions can be read. This modification allows for faster responsiveness compared to the sequential Strickt Two Phase Locking (S2PL) [49].

This stream transaction model is generic since it is applicable to the sharing techniques described above. However, a customized transaction model that considers the semantics of the view-maintenance operations on a shared common view might be more efficient. Ray et al. [43] introduce a customized stream transaction model

for shared views. This model defines the compatibility of read, append, and purge operations on shared views. It then uses S2PL to schedule transactions composed of such operations. Future work could focus on developing concurrent processing models that best support each of the different shared analytics optimization scenarios.

7 Related Work

Complex Event Query Processing. Existing event processing systems focus on the specification and optimization of automaton-based [1, 13, 50] and query-plan-based [40] execution paradigms. Liu et al. [35] consider nested event patterns and introduce a top-down iterative approach for processing such queries. However, these approaches neither address the issue of supporting queries at different concept and pattern hierarchy levels nor do they develop efficient computation strategies for the shared execution of multiple event queries.

Sharing Multiple Event Queries. Shared event query processing is part of the native architecture of TelegraphCQ [9]. Madden et al. [38] proposed an adaptive tuple-level sharing technique. However, routing individual tuples among operators introduces considerable overhead. Instead, our approach produces a stable sharing plan and re-optimizes only if there is significant change in statistics [31].

Hong et al. [29] introduce materialization-based optimization techniques into XML-stream query processing. This approach does not consider windows, event correlations, view maintenance and concurrent query access to these views. YFilter [14] is limited to prefix-matching. In contrast, our technique shares sub-patterns at arbitrary positions. Ray et al. [44] propose continuous sliding-view maintenance over event streams for a single query. Sharing such views among multiple queries is not considered.

Hierarchical Event-Query Sharing. Traditional OLAP technologies focus on static pre-computed and indexed data sets. They aim to quickly provide answers to analytical queries that are multi-dimensional in nature [10, 22, 27]. OLAP techniques allow users to navigate the data at different abstraction levels. However, these solutions either do not support real-time streams [20, 26, 37], or they are set-based instead of sequence-based [22]. Furthermore, these approaches do not support concept hierarchies. They provide neither result reuse strategies nor any cost analysis for patterns expressing event sequence and negation.

Shared Event Query Aggregation. The optimization of CEP aggregation is critical for high performance pattern matching over event streams [1, 13, 40, 50]. However, no specific technique has been proposed to date to optimize the on-the-fly computations of event-sequence aggregation. Instead, existing approaches apply aggregation as a *post-processing step* that takes place after all event sequences have been constructed. Obviously, this is an inefficient solution. Incremental techniques [30, 34] have been proposed to avoid re-computations among overlapping sliding windows. Zhang et al. [51] maintain aggregates using multiple levels of temporal granularity: older data is aggregated using coarser granularity while more recent data is aggregated with fine detail. However, these approaches do not address our sequence aggregation problem, that is, they compute aggregation over individual events rather than over event sequences that are continuously detected in real time.

Aggregation is well-supported in static sequence databases [32, 37]. These approaches assume that the data is statically stored and indexed prior to processing. In contrast, our approach targets dynamic streaming data where results are produced continuously upon event arrival and events are discarded once they are aggregated.

Range-based aggregation approaches [32, 48] aggregate independent data records within a certain time range. Some approaches [41, 46] consider aggregation for patterns with recursion. However, these approaches work with independent individual data records. In contrast to that, our approach aggregates event sequences matched by expressive event patterns, i.e., interdependent multi-record matches.

Stream Transaction Models. Botan et al. [6] adapt the traditional database transaction model to event stream processing. That is, a transaction is a sequence of user-defined operations. Events must be processed in order by their arrival time stamps. Other stream transaction models [4, 39] define a transaction as a sequence of operations triggered by one or more input events. Events are usually batched and their processing is ordered by event time stamps. However, these approaches are too restrictive, since they process events in strict order

and disallow concurrent operations on the same data item, unlike our proposed Low-Water-Mark scheduler [49]. This strictly ordered processing strategy slows down execution and results in poorer system responsiveness.

8 Conclusion and Future Work

In this article, we have presented an overview of four innovative techniques for scaling shared event analytics, namely: (1) To effectively share identical sub-patterns, we consider intra- and inter-query correlation, match distribution over time and match sharing at the event instance level. (2) Since the number of identical sub-patterns in an event query workload may be limited, we also share computations among hierarchical event queries. (3) While computing event sequence aggregation, we do not construct the actual event sequences and thus reduce the computation costs from polynomial to linear. Multiple aggregation event queries share the aggregation computation of their common sub-patterns. (4) Our stream transaction model guarantees correct concurrent execution of multiple inter-dependent event queries sharing their intermediate results.

In the future, we will extend our online shared aggregation approach to a broader class of event queries. For time-critical decision making applications, certain urgent insights are useful only if derived within a strict time constraint. Thus, we will define different consistency levels and propose prioritized scheduling algorithms to ensure prompt responsiveness using limited resources. Furthermore, these techniques have been proposed in the context of a central albeit possibly multi-threaded architecture. The next logical step would be to explore their effectiveness in context of deploying complex event analytics on an distributed computing platform.

Acknowledgements

Section 4 is the result of a successful collaboration with the researchers from HP Labs, in particular, Chetan Gupta, Song Wang and Abhay Mehta. The authors also thank Kara Greenfield and Ismail Ari for productive collaboration. For Section 6, the authors collaborated with UMass Medical School, in particular, Richard T. Ellison III. We thank Dr. Ellison for his leadership of installation of our event analytics system in the intensive care units at UMASS Memorial Hospital. This work was supported by the following grants: NSF IIS-III-1018443, NSF IIS 0917017, NSF CRI (Equipment Grant), HP Lab Innovation Research Grant, UMMS-WPI CCTS Collaborative Grant, and Turkish National Science Foundation TUBITAK under career award 109E194.

References

[1] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *SIGMOD*, pages 147–160. ACM, 2008.

[2] M. Akdere, U. Çetintemel, and N. Tatbul. Plan-based complex event detection across distributed sources. *In VLDB*, 1(1):66–77, 2008.

[3] A. Arasu, and J. Widom. Resource sharing in continuous sliding-window aggregates. In *VLDB*, pages 336–347, 2004.

[4] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.

[5] P. Bernstein and E. Newcomer. *Principles of Transaction Processing: For the Systems Professional*. Morgan Kaufmann Publishers Inc., 1997.

[6] I. Botan, P. M. Fischer, D. Kossmann, and N. Tatbul. Transactional stream processing. In *EDBT*, pages 204–215, 2012.

[7] B. Cadonna, J. Gamper, and M. H. Bohlen. Sequenced event set pattern matching. In *EDBT*, pages 33–44, 2011.

[8] U. Chakravarthy and J. Minker. Multiple query processing in deductive databases using query graphs. In *VLDB*, pages 384–391, 1986.

[9] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, pages 668–680, 2003.

[10] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *In SIGMOD*, 26(1):65–74, 1997.

[11] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *ICDE*, pages 190 – 200, 6-10 Mar 1995.

[12] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *SIGMOD*, pages 379–390, 2000.

[13] A. J. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. M. White. Cayuga: A general purpose event monitoring system. In *CIDR*, pages 412–422, 2007.

[14] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. Fischer. Path sharing and predicate evaluation for high-performance XML filtering. *ACM TODS*, 28(4):467–516, 2003.

[15] Y. Diao, P. Fischer, M. J. Franklin, and R. To. Yfilter: Efficient and scalable filtering of XML documents. In *ICDE*, pages 341–342, 2002.

[16] R. Ellison, D. W. Constance M. Barysauskas, Elke A. Rundensteiner, and B. Barton. A prospective controlled trial of an electronic hand hygiene reminder system. In *IDWeek Conference, Advancing Science Improving Care*, 2013. Abstract 314.

[17] R. Ellison, D. W. Constance M. Barysauskas, Elke A. Rundensteiner, and B. Barton. A prospective controlled trial of an electronic hand hygiene reminder system. *Open Forum Infectious Diseases*, 2(4):1–8, Dec. 2015.

[18] H. N. Gabow, Z. Galil, T. H. Spencer, and R. E. Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6(2):109–122, 1986.

[19] T. M. Ghanem, W. G. Aref, and A. K. Elmagarmid. Exploiting predicate-window semantics over data streams. *In SIGMOD*, 35(1):3–8, Mar. 2006.

[20] H. Gonzalez, J. Han, and X. Li. Flowcube: Constructuing RFID FlowCubes for multi-dimensional analysis of commodity flows. In *VLDB*, pages 834–845, 2006.

[21] J. Grant and J. Minker. On optimizing the evaluation of a set of expressions. *Int. J. of Computer & Information Sciences*, pages 179–191, 1982.

[22] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-query processing in data warehousing environments. In *VLDB*, pages 358–369, 1995.

[23] C. Gupta, S. Wang, A. Mehta, M. Liu, and E. Rundensteiner. Computing a hierarchical pattern query from another hierarchical pattern query, Apr. 25 2013. Patent US20130103638 A1.

[24] C. Gupta, S. Wang, A. Mehta, M. Liu, and E. Rundensteiner. Determining an execution ordering, Apr. 5 2016. Patent US9305058 B2.

[25] C. Gupta, S. Wang, A. Mehta, M. Liu, E. Rundensteiner, and M. Ray. Nested complex sequence pattern queries over event streams, Mar. 29 2016. Patent US9298773 B2.

[26] J. Han, Y. Chen, G. Dong, J. Pei, B. W. Wah, J. Wang, and Y. D. Cai. Stream Cube: An architecture for multi-dimensional analysis of data streams. *Distributed and Parallel Databases*, 18(2):173–197, 2005.

[27] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *SIGMOD*, pages 205–216, 1996.

[28] D. Hermelin, D. Rawitz, R. Rizzi, and S. Vialette. The minimum substring cover problem. In *Int. Conf. on Approximation and Online Algorithms*, pages 170–183, 2008.

[29] M. Hong, A. J. Demers, J. E. Gehrke, C. Koch, M. Riedewald, and W. M. White. Massively multi-query join processing in publish/subscribe systems. In *SIGMOD*, pages 761–772, 2007.

[30] S. Krishnamurthy, C. Wu, and M. J. Franklin. On-the-fly sharing for streamed aggregation. In *SIGMOD*, pages 623–634, 2006.

[31] C. Lei and E. A. Rundensteiner. Robust distributed stream processing. In *ICDE*, pages 817–828, 2013.

[32] A. Lerner and D. Shasha. AQuery: Query language for ordered data, optimization techniques, and experiments. In *VLDB*, pages 345–356, 2003.

[33] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. No pane, no gain: Efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Rec.*, 34(1):39–44, Mar. 2005.

[34] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. Semantics and evaluation techniques for window aggregates

in data streams. In *SIGMOD*, pages 311–322, 2005.

[35] M. Liu, E. A. Rundensteiner, D. J. Dougherty, C. Gupta, S. Wang, I. Ari, and A. Mehta. High-performance nested CEP query processing over event streams. In *ICDE*, pages 123 – 134, April, 2011.

[36] M. Liu, E. A. Rundensteiner, K. Greenfield, C. Gupta, S. Wang, I. Ari, and A. Mehta. E-Cube: Multi-dimensional event sequence analysis using hierarchical pattern query sharing. In *SIGMOD*, pages 889–900, 2011.

[37] E. Lo, B. Kao, W.-S. Ho, S. D. Lee, C. K. Chui, and D. W. Cheung. OLAP on sequence data. In *SIGMOD*, pages 649–660, 2008.

[38] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *SIGMOD*, pages 49–60, 2002.

[39] J. Meehan, N. Tatbul, S. Zdonik, C. Aslantas, U. Cetintemel, J. Du, T. Kraska, S. Madden, D. Maier, A. Pavlo, M. Stonebraker, K. Tufte, and H. Wang. S-Store: Streaming meets transaction processing. *In VLDB*, 8(13):2134–2145, 2015.

[40] Y. Mei and S. Madden. ZStream: A Cost-based query processor for adaptively detecting composite events. In *SIGMOD*, pages 193–206, 2009.

[41] I. Motakis and C. Zaniolo. Temporal aggregation in active database rules. In *SIGMOD*, pages 440–451, 1997.

[42] Y. Qi, L. Cao, M. Ray, and E. A. Rundensteiner. Complex event analytics: Online aggregation of stream sequence patterns. In *SIGMOD*, pages 229–240, 2014.

[43] M. Ray, C. Lei, and E. A. Rundensteiner. Scalable pattern sharing on event streams. In *SIGMOD*, 2016. (To appear).

[44] M. Ray, E. A. Rundensteiner, M. Liu, C. Gupta, S. Wang, and I. Ari. High-performance complex event processing using continuous sliding views. In *EDBT*, pages 525–536, 2013.

[45] E. Rundensteiner, D. Wang, and R. Ellison. Active complex event processing or infection control and hygiene monitoring, Oct. 6 2011. US Patent App. 13/077,401.

[46] R. Sadri, C. Zaniolo, A. Zarkesh, and J. Adibi. Expressing and optimizing sequence queries in database systems. *ACM Trans. Database Syst.*, 29(2):282–318, June 2004.

[47] N. P. Schultz-Møller, M. Migliavacca, and P. Pietzuch. Distributed complex event processing with query rewriting. In *DEBS*, pages 4:1–4:12, 2009.

[48] P. Seshadri, M. Livny, and R. Ramakrishnan. SEQ: Design and implementation of a sequence database system. In *VLDB*, pages 99–110, 1996.

[49] D. Wang, E. A. Rundensteiner, and R. T. Ellison, III. Active complex event processing over event streams. *In VLDB*, 4(10):634–645, July 2011.

[50] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *SIGMOD*, pages 407–418, 2006.

[51] D. Zhang, D. Gunopulos, V. J. Tsotras, and B. Seeger. Temporal aggregation over data streams using multiple granularities. In *EDBT*, pages 646–663, 2002.

Handling Shared, Mutable State in Stream Processing with Correctness Guarantees

Nesime Tatbul[1,2], Stan Zdonik[3], John Meehan[3], Cansu Aslantas[3],
Michael Stonebraker[2], Kristin Tufte[4], Chris Giossi[4], Hong Quach[4]
[1]Intel Labs [2]MIT [3]Brown University [4]Portland State University
{tatbul,stonebraker}@csail.mit.edu, {sbz,john,cpa}@cs.brown.edu, {tufte,cgiossi,htquach}@pdx.edu

Abstract

S-Store is a next-generation stream processing system that is being developed at Brown, Intel, MIT, and Portland State University. It is designed to achieve very high throughput, while maintaining a number of correctness guarantees required to handle shared, mutable state in streaming applications. This paper explores these correctness criteria and describes how S-Store achieves them, including a new model of stream processing that provides support for ACID transactions.

1 Introduction

Stream processing has been around for a long time. Over a decade ago, the database community explored the topic of near-real-time processing by building a number of prototype systems [6, 9, 15]. These systems were based on a variant of the standard relational operators that were modified to deal with the unbounded nature of streams.

Additionally, streaming applications require support for storage and historical queries. In our view, the early systems did not properly address storage-related issues. In particular, they largely ignored the handling of shared, mutable state. They were missing the guarantees that one would expect of any serious OLTP DBMS. These correctness guarantees are needed in addition to those that streaming systems typically provide, such as exactly-once processing (which requires that, upon recovery, the system will not lose or duplicate data).

We believe that it is time to take a look at streaming through the lens of these processing guarantees. In this paper, we present S-Store, which is designed to address the correctness aspects of streaming applications. We show that it is possible to support correctness without serious performance degradation. We also show that the only way to achieve good performance is by tightly integrating storage management with the streaming infrastructure. Some modern streaming systems require the use of an external storage manager to provide needed services [2, 3, 22, 27]. As we will show, using external storage comes at a cost.

We begin with describing a motivating use case, and proceed to discuss S-Store's correctness guarantees, computational model and implementation to achieve these guarantees, followed by an experimental comparison with the state of the art.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

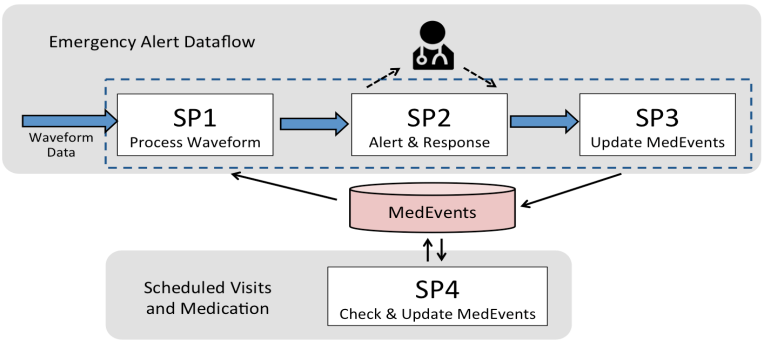


Figure 1: Multiple Streaming Dataflows Sharing State: A MIMIC-based Example [16, 26]

2 Example Use Case

In this section, we present a selected use case of S-Store, based on a recent demonstration of the BigDAWG polystore system [16], in which S-Store was used for real-time alert monitoring over streaming patient waveforms from an ICU (intensive care unit). This use case illustrates the need for consistently managing state shared among multiple streaming dataflows.

In a hospital environment, proper tracking of patient medications is critical to avoid overdoses and bad drug interactions. Studies have estimated that preventable "adverse drug events" (with patient injury) in hospitals to be between 380,000 and 450,000 per year [20]. We observe that different types of events may trigger medication administration: an emergency alert, a periodic doctor visit, or a periodic medication administration schedule. These events all require reading and updating of the list of medication administrations for a patient. In the MIMIC ICU data set [26], this data is stored in a medication events (*MedEvents*) table. Thus, separate dataflow graphs must update a single shared table, which requires transactional consistency to avoid patient injury.

Figure 1 diagrams potential S-Store dataflow graphs that update the *MedEvents* table. The upper dataflow represents an emergency alert notification, while the lower dataflow represents periodic doctor visits and medication administrations. In the emergency alert dataflow, a stored procedure (*SP1*) reads incoming patient waveform data (e.g., Pulmonary Arterial Pressure (PAP)), and calculates a windowed average over it. When this average is elevated, a doctor must be notified and medication may be recommended; however, medication must not be recommended if the medication has been recently administered. The doctor then either accepts or declines the recommendation, and the *MedEvents* table is updated appropriately. In the periodic-visits dataflow, a doctor or a schedule decides which medication is advisable. Before administering the medication, the caregiver enters the medication to be administered. The system then checks for potential drug interactions with recent medication administrations using the *MedEvents* table, and then updates *MedEvents* appropriately. This simpler dataflow is similar in nature to an OLTP transaction.

For ensuring correct semantics, this example requires ordered execution of its dataflows and transactional (ACID) access to the *MedEvents* table. More specifically, *SP1* must read the *MedEvents* table before an alert is sent to the doctor; the *MedEvents* table needs to remain locked so that other updates – such as from *SP4* – cannot interfere. Thus, *SP1*, *SP2*, and *SP3* must be part of an ordered dataflow within a single nested transaction. Furthermore, *SP1*, *SP2*, *SP3* cannot be a single stored procedure due to the human interaction. Note that this example could be extended with other similar emergency-alert dataflows, as different types of analysis are needed on different waveform streams, e.g., cardiac anomalies to be detected from ECG-waveform streams.

A similar workload pattern can be found in other domains such as transportation, wherein one or more shared tables must be read and updated by multiple dataflows, as might be seen in the display of messages on Variable Message Signs and Adaptive Signal Control. In this case, transactional processing support would be

required to avoid inconsistencies, garbled messages, and incorrect signal timing. We note that in most of these examples, the dataflows are fully automated (i.e., human-in-the-loop is not necessarily a critical requirement as in the medical setting).

3 Correctness

Transaction-processing systems normally provide *ACID (Atomicity, Consistency, Isolation, and Durability)* guarantees. These guarantees broadly protect against data corruption of two kinds: (i) interference of concurrent transactions, and (ii) transaction failures. Consistency and Isolation primarily address interference, while Atomicity and Durability address failures. It is widely understood that failures can cause data inconsistencies. Thus, most stream processing engines also cover this case by incorporating failure-recovery facilities. However, it is less widely acknowledged that any streaming computation that shares mutable data with other computations (e.g., a separate streaming dataflow graph) must guard against interference from those computations as in standard OLTP.

In addition to ACID, there are other correctness requirements from stream processing that must be considered. First, a transaction execution must conform to some logical *order* specified by the user. The scheduler should be free to produce a schedule that interleaves transactions in a variety of ways, but the results must be equivalent to the specified logical order. Secondly, it has been shown that, in streaming systems, failures may lead to lost or duplicated tuples. It puts a burden on the application to detect and react to such problems appropriately. Thus, streaming systems typically strive to provide *exactly-once* semantics as part of their fault-tolerance mechanisms.

For correctly handling hybrid workloads, S-Store provides efficient scheduling and recovery mechanisms that maintain three complementary correctness guarantees that are needed by both streaming and transactional processing. In what follows, we discuss these guarantees.

3.1 ACID Guarantees

We regard a transaction as the basic unit of computation. As in conventional OLTP, a transaction T must take a database from one consistent state to another. In S-Store, the database state consists of streaming data (*streams* and *windows*) in addition to non-streaming data (*tables*). Accordingly, we make a distinction between two types of transactions: (i) *OLTP transactions* that only access tables, and are activated by explicit transaction requests from a client, and (ii) *streaming transactions* that access streams and windows as well as tables, and are activated by the arrival of new data on their input streams. Both types of transactions are subject to the same interference and failure issues discussed above. Thus, first and foremost, S-Store strives to provide ACID guarantees for individual OLTP and streaming transactions in the same way traditional OLTP systems do. Furthermore, access to streams and windows require additional isolation restrictions, in order to ensure that such streaming state is not publicly available to arbitrary transactions that might endanger the streaming semantics.

3.2 Ordered Execution Guarantees

Stream-based computation requires ordered execution for two primary reasons: (i) streaming data itself has an inherent order (e.g., timestamps indicating order of occurrence or arrival), and (ii) processing over streaming data has to follow a number of consecutive steps (e.g., expressed as directed acyclic dataflow graphs as illustrated in Figure 1). Respecting (i) is important for achieving correct semantics for order-sensitive operations such as sliding windows. Likewise, respecting (ii) is important for achieving correctness for complex dataflow graphs as a whole.

Traditional ACID-based models do not provide any order-related guarantees. In fact, transactions can be executed in any order as long as the result is equivalent to a serial schedule. Therefore, S-Store provides an ad-

ditional correctness guarantee that ensures that every transaction schedule meets the following two constraints: (i) for a given streaming transaction T, atomic batches of an input stream S must be processed in order (a.k.a., stream order constraint), and (ii) for a given atomic batch of stream S that is input to a dataflow graph G, transactions that constitute G must be processed in a valid topological order of G (a.k.a., dataflow order constraint).

For coarser-grained isolation, S-Store also allows the user to define nested transactions as part of a dataflow graph (e.g., see the *Emergency Alert Dataflow* in Figure 1), which may introduce additional ordering constraints [23]. S-Store's scheduler takes all of these constraints into account in order to create correct execution schedules.

3.3 Exactly-once Processing Guarantees

Failures in streaming applications may lead to lost state. Furthermore, recovering from failures typically involves replicating and replaying streaming state, which, if not applied with care, may lead to redundant executions and duplicated state. To avoid these problems, streaming systems strive to provide fault tolerance mechanisms that will ensure "exactly-once" semantics. Note that exactly-once may refer either (i) to external delivery of streaming results, or (ii) to processing of streams within the system. The former typically implies the latter, but the latter not necessarily implies the former. In this work, we have so far mainly focused on the latter (i.e., exactly-once processing, not delivery), as that is more directly relevant in terms of database state management.

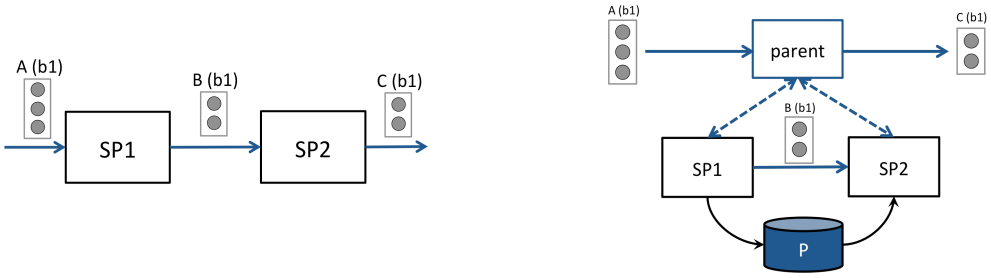
Exactly-once processing is not a concern in traditional OLTP. Any failed transaction that was partially executed is undone (Atomicity), and it is up to the user to reinvoke such a transaction (i.e., the system is not responsible for loss due to such transactions). On the other hand, any committed transaction that was not permanently recorded must be redone by the system (Durability). State duplication is not an issue, since successful transactions are made durable effectively only once. This approach alone is not sufficient to ensure exactly-once processing in case of streaming transactions, mainly because of the order and data dependencies among transaction executions. First, any failed transaction must be explicitly reinvoked to ensure continuity of the execution without any data loss. Second, it must be ensured that redoing a committed transaction does not lead to redundant invocations on others that depend on it.

S-Store provides exactly-once processing guarantees for all streaming state kept in the database. This guarantee ensures that each atomic batch on a given stream S that is an input to a streaming transaction T is processed exactly once by T. Note that such a transaction execution, once it commits, will likely modify the database state (streams, windows, or tables). Thus, even if a failure happens and some transaction executions are undone or redone during recovery, the database state must be "equivalent" to one that is as if S were processed exactly once by T.

Note that executing a streaming transaction may have an external side effect other than modifying the database state (e.g., delivering an output tuple to a sink that is external to S-Store). It is possible that such a side effect may get executed multiple times during recovery. Thus, our exactly-once processing guarantee applies only to state that is internal to S-Store. This is similar to other exactly-once processing systems such as Spark Streaming [28]. Exactly-once delivery might also be important in some application scenarios (e.g., dataflow graphs that involve a human-in-the-loop computation as in the medical use case described in Section 2). We plan to investigate this guarantee in more detail as part of our future work.

4 Model Overview

We now describe our model, which allows us to seamlessly mix OLTP transactions and streaming transactions. The basic computational unit in S-Store is a transaction, and all transactions are pre-declared as stored procedures. A stored procedure is written in both SQL (to interact with tables that store database state) and in Java (to allow arbitrary processing logic). Streaming transactions are those that take finite batches of tuples from streams as input and may produce finite batches of tuples as output. As one would expect, all transactions (streaming or



(a) Processing of Atomic Batches

(b) Nested Transactions

Figure 2: Example Dataflows

not), preserve the standard ACID properties of OLTP database systems.

As mentioned earlier, S-Store manages three kinds of state: (i) streams, (ii) windows, and (iii) tables. S-Store models a stream as an unbounded sequence of tuples. These tuples arrive in some order and are processed in chunks (called *atomic batches*). An atomic batch is a contiguous, non-overlapping subsequence of a stream in which all tuples in the batch share a common batch-id. A typical example is to group tuples with a common application timestamp or time-interval into the same batch [9, 28]. We assume that batches over a stream should be processed in ascending order of their batch-id's; however the order of tuples within a single batch does not matter since each batch is always processed as an indivisible atomic unit.

A window over a stream is also a contiguous subsequence of that stream, but different from atomic batches, windows come with a set of rules for deriving a new window from an old one. Windows are defined in units of batches (as opposed to time or tuple count), and can slide and tumble much as in previous generations of streaming systems [11], so, we will not go into detail here. It is important to note that defining windows in batch units ensures that windows are processed in a deterministic way, avoiding the "evaporating tuples" problem discussed in previous work [9, 12].

Streams, windows, and tables differ in terms of which transactions are allowed to access them. Tables can be publicly read or written by any transaction, while windows are private to the transaction in which they are defined, and streams are private to their "producer" and "consumer" transactions.

Streaming systems typically push data from input to output. This arrangement reduces latency, since there is no need to poll the inputs to determine if the next input batch is ready. S-Store, like other systems, captures the notion of the next action to perform through a dataflow graph. In the case of S-Store, the actions are transactions, making the dataflow graph a DAG with transaction definitions as nodes, and a directed arc from node T_i to node T_j if T_j should follow T_i in the processing order. That is to say, when T_i commits, T_j should be triggered next.

Figure 2a shows a two-stored-procedure (i.e., *SP1* and *SP2*) dataflow graph. The batch of tuples labeled *A* is the input to *SP1*, all with the same batch-id *b1*. *SP1* begins execution as a transaction with the 3-tuple batch as input. Suppose that *SP1* commits with the batch labeled *B* as output. The tuples in batch *B* would be assigned the batch-id of the inputs that they were derived from (*b1*), and the process repeats with batch *B* as input to *SP2* and batch *C* as the output batch for *SP2*.

Stored procedures that take multiple streams as input or emit multiple streams as output are processed in a similar way. In this case, a stored procedure begins execution with atomic batches from all of its input streams with a common batch-id and the same batch-id carries over to any output batches that result from this execution.

For each transaction definition, there could be many transaction executions (TEs). If stream *S* is the input to transaction *T*, a TE is created every time a new batch of tuples arrives on stream *S*. Windows are created in TEs. Since they are the principal data structure that reacts to the unbounded nature of a stream, the i^{th} TE for a transaction *T* will inherit any window state that is active in the $(i-1)^{st}$ TE for *T*. Aside from this exception,

windows are private and cannot be shared with TEs for other transactions, since that would break the isolation requirement for ACID transactions. Similarly, streams can only be shared by the TE's of their producer and consumer transactions in a dataflow (e.g., only TE's of *SP1* and *SP2* can share the stream that flows between them in Figure 2a).

We also provide a nested transaction facility that allows the application programmer to build higher-level transactions out of smaller ones, giving her the ability to create coarser isolation units among transactions, as illustrated in Figure 2b. In this example, two streaming transactions, *SP1* and *SP2*, in a dataflow graph access a shared table *P*. *SP1* writes to the table and *SP2* reads from it. If another OLTP transaction also writes to *P* in a way to interleave between *SP1* and *SP2*, then *SP2* may get unexpected results. Creating a nested transaction with *SP1* and *SP2* as its children will isolate the behavior of *SP1* and *SP2* as a group from other transactions (i.e., other OLTP or streaming). Note that nested transactions also isolate multiple instances of a given streaming dataflow graph (or subgraph) from one another.

S-Store transactions can be executed in any order as long as this order obeys the ordering constraints imposed by: (i) the relative order of atomic batches on each input stream, (ii) the topological ordering of the stored procedures in the dataflow graph, (iii) any additional constraints due to nested transactions. Assuming that transaction definitions themselves are deterministic, this is the only source of potential non-determinism in S-Store transaction schedules. For example, for the simple dataflow in Figure 2a, both of the following would be valid schedules: $[TE1(b1); TE1(b2); TE2(b1); TE2(b2)]$ or $[TE1(b1); TE2(b1); TE1(b2); TE2(b2)]$. On the other hand, for the dataflow in Figure 2b, the former schedule would not be allowed due to the nesting.

A more detailed description of our model can be found in a recent publication [23].

5 Implementation

Our S-Store implementation seeks to prove that we can provide all of the correctness guarantees mentioned above without major loss of performance. Implementation of the mechanisms to provide these guarantees must be native to the system to minimize overhead.

S-Store is built on top of H-Store [21], a high-throughput main-memory OLTP system, in order to take advantage of its extremely light-weight transaction mechanism. Thus, like H-Store, S-Store follows a typical two-layer distributed DBMS architecture (see Figure 3). Transactions are initiated in the partition engine (PE), which is responsible for managing transaction distribution, scheduling, coordination, and recovery. The PE manages the use of another layer called the execution engine (EE), which is responsible for the local execution of SQL queries. As mentioned earlier, transactions are predefined as stored procedures which are composed of Java and SQL statements. When a stored procedure is invoked with input parameters, a transaction execution (TE) is instantiated and runs to completion before committing. A client program connects to the PE via a stored-procedure execution request. If the stored procedure requires SQL processing, then the EE is invoked with those sub-requests.

While we chose H-Store to serve as our foundation, our architectural extensions and mechanisms could be implemented on any main-memory OLTP engine, thereby directly inheriting the required ACID guarantees discussed in Section 3.1. We are able to achieve our desired correctness guarantees due to the implementation additions described in the following subsections.

5.1 ACID Implementation

In order to maintain the transactional properties inherited from H-Store, we implement our dataflow graph as a series of stored procedures connected by streams. All streaming state, including both **streams** and **windows**, are implemented as time-varying tables, which are accessed within stored procedures. Thus, it is impossible to access streaming state in a non-transactional manner.

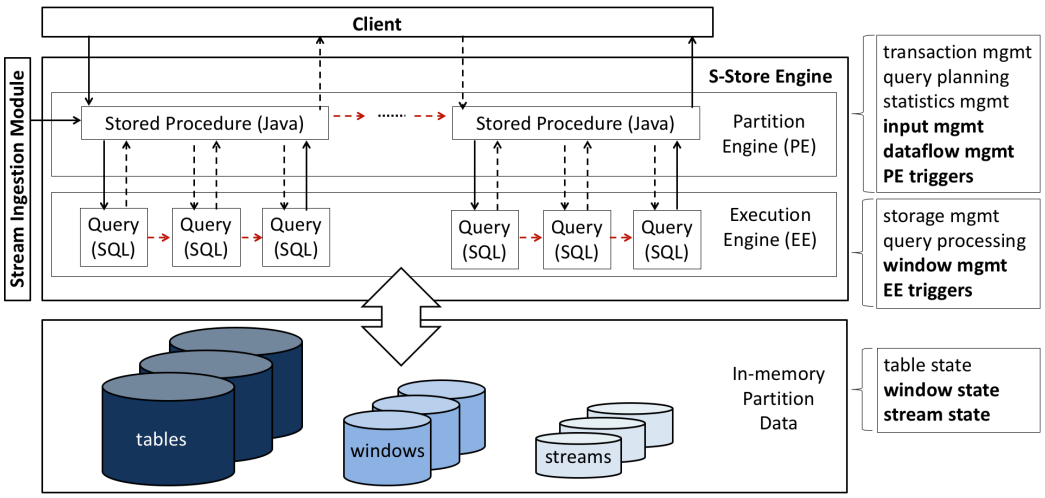


Figure 3: S-Store Architecture

The stored procedures within the dataflow are connected by streams, and activated via **partition engine (PE) triggers**. When a transaction commits and places a new batch of tuples into its output stream, any downstream transactions in the dataflow are immediately scheduled for execution using that output batch as their input.

In addition to PE triggers, S-Store includes **execution engine (EE) triggers**. These allow SQL statements to be invoked upon the availability of a new tuple in a stream or the slide of a window. Unlike PE triggers, EE triggers execute within the same transaction as the insertion that activated them.

5.2 Ordering Implementation

Because S-Store breaks down a dataflow into multiple discrete stored procedures, multiple simultaneous transaction requests must be scheduled in such a way that ordering is maintained between stored procedures within a dataflow, and between dataflow instantiations. S-Store provides such a **streaming scheduler**.

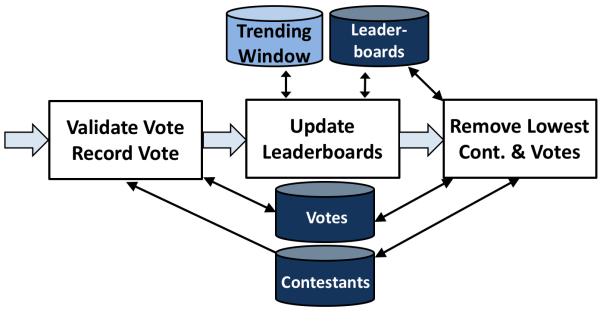
In single-node S-Store, transactions are scheduled serially, meaning that a batch will be processed to completion within a dataflow graph before the next batch is considered. This simple scheduling policy ensures that both stream and dataflow order constraints will always be satisfied for a given dataflow graph. In our ongoing work, we are extending the streaming scheduler to operate over multiple nodes.

5.3 Exactly-Once Implementation

Within single-node S-Store, our primary concern regarding exactly-once processing lies within internal message passing via streams, so we provide the guarantee primarily through fault tolerance. We provide two alternative fault-tolerance mechanisms, both of which guarantee exactly-once processing semantics.

In **strong recovery**, each transaction execution is logged using H-Store's command-logging mechanism. When recovering in this mode, the original execution order of the transactions will be replayed in exactly the same way as in the log. To ensure the exactly-once processing guarantee, PE triggers are turned off during recovery; all transactions are replayed from the log, but no transactions will be repeated.

In **weak recovery**, only "border" transactions (i.e., transactions that begin a dataflow graph) are command-logged. Upon recovery, these transactions are re-executed, but with PE triggers kept turned-on. The streaming scheduler will execute the full dataflow graph in a *legal* order according to ordering and data isolation rules, but not necessarily in the *exact* order that they were originally executed before the failure. This alternative recovery mode improves both run-time and recovery performance, while still providing the ordered execution (via the



(a) Leaderboard Maintenance Dataflow

System	ACID	Order	Exactly-Once	Max Tput (batches/sec)
H-Store (async)	✓	✗	✗	5300
H-Store (sync)	✓	✓	✗	210
Esper + VoltDB	✓	✓	✗	570
Storm + VoltDB	✓	✓	✓	600
S-Store	✓	✓	✓	2200

(b) Experiment Result

Figure 4: Performance vs. Correctness Guarantees

scheduler) and exactly-once processing guarantees.

For more information about the implementation of S-Store, please refer to our PVLDB paper [23].

6 State-of-the-Art Comparison

When evaluating S-Store's performance, it is once again important to consider the three guarantees described in Section 3. In modern state-of-the-art systems, it is challenging to provide all three processing guarantees. More specifically, OLTP systems are able to process ACID transactions with high performance, but have no concept of dataflow graphs, and thus no inherent support for ordering or exactly-once processing. In contrast, stream processing systems are able to provide dataflow ordering and exactly-once processing, but do not support ACID transactions. Thus, in both cases, achieving all three guarantees with high performance is a major challenge.

To test S-Store's performance in comparison to current state of the art, we created a simple leaderboard-maintenance benchmark. This benchmark mimics a singing competition in which users vote for their favorite contestants, and periodically, the lowest contestant is removed until a winner is selected. As shown in Figure 4a, the benchmark's dataflow graph is composed of three stored procedures that each access shared table state, and thus requires data isolation (i.e., a nested transaction) across all three. For the purposes of simplifying comparison across systems, we considered a batch to be a single vote, and we record our throughput numbers in terms of "input batches per second."

The leaderboard-maintenance benchmark requires all three of S-Store's processing guarantees to be executed correctly. We first compared S-Store's performance to its OLTP predecessor, H-Store. As an OLTP system, by default H-Store only provides the first guarantee, ACID, and thus maintains an impressive throughput (over 5000 input batches per second, as shown in the first row of Figure 4b). However, the results it provides are incorrect; a wrong candidate may win the contest since votes may be processed in a different order than the one that is required by the benchmark. For H-Store to provide correct results, the ordering guarantee must also be provided.

We can force H-Store to provide an ordering guarantee across the dataflow graph by insisting that H-Store process the whole dataflow graph serially. In this case, the client has to manage the order in which the transactions are executed, by waiting for a response from the engine before it can submit the next transaction request (i.e., submitting requests in a synchronous manner). As one would expect, performance suffers drastically as a result. H-Store's throughput plummets to around 200 input batches per second, when ordering constraints are enforced via synchronous requests.

Both single-node streaming engines (e.g., Esper [3]) and distributed stream processing engines (e.g., Storm [27]) also struggle to provide all three processing guarantees. In the case of streaming engines, dataflow graphs

are core functionality, and the ordering guarantee is provided. Exactly-once processing can also be added to many systems possibly with some loss in performance (e.g., Storm with Trident [4]). However, ACID transactions are not integrated into streaming systems. Instead, they must use an additional OLTP database to store and share the mutable state consistently. For our experiments, we used VoltDB [5] (the commercial version of H-Store) to provide this functionality to Esper and Storm.

Similarly to H-Store, providing all three processing guarantees degrades throughput. To provide both ordering and ACID, the streaming systems must submit requests to the OLTP database and wait for the response to move on. Even with a main-memory OLTP system such as VoltDB, this additional communication takes time and prevents the stream system from performing meaningful work in the meantime. As shown in Figure 4b, both Esper and Storm with Trident were only able to manage about 600 input batches per second, when providing ACID guarantees through VoltDB.

By contrast, S-Store is able to maintain 2200 input batches per second on the same workload, while natively providing all three processing guarantees. S-Store manages both dataflow graph ordering and consistent mutable state in the same engine. This combination allows S-Store to handle multiple asynchronous transaction requests from the client and still preserve the right processing order within the partition engine. Meanwhile, each operation performed on any state is transactional, guaranteeing that the data is consistent every time it is accessed – even in presence of failures.

7 Related Work

First-generation streaming systems provided relational-style query processing models and system architectures for purely streaming workloads [3, 6, 9, 15]. The primary focus was on low-latency processing over push-based, unbounded, and ordered data arriving at high or unpredictable rates. State management mostly meant efficiently supporting joins and aggregates over sliding windows, and correctness was only a concern in failure scenarios [10, 19].

Botan et al. proposed extensions to the traditional database transaction model to enable support for continuous queries over both streaming and stored data sources [13]. While this work considered ACID-style access to shared data, its focus was limited to correctly ordering individual read and write operations for a single continuous query rather than transaction-level ordering for complex dataflow graphs as in S-Store.

More recently, a new breed of streaming systems has emerged, which commonly aim at providing a MapReduce-like distributed and fault-tolerant framework for real-time computations over streaming data. Examples include S4 [25], Storm [27], Twitter Heron [22], Spark Streaming [28], Samza [2], Naiad [24], Flink [1], and MillWheel [7]. These systems significantly differ in terms of the way they manage persistent state and the correctness guarantees that they provide, but none of them is capable of handling streaming applications with shared mutable state with sufficient consistency guarantees as provided by S-Store.

S4, *Storm*, and *Twitter Heron* neither support fault-tolerant persistent state nor can guarantee exactly once processing. Storm when used with *Trident* can ensure exactly-once semantics, yet with significant degradation in performance [4]. Likewise, *Google MillWheel* can persist state with the help of a backend data store (e.g., BigTable or Spanner), and can deal with out-of-order data with exactly once processing guarantees using a low-watermark mechanism [7].

Several recent systems adopt a *stateful dataflow model* with support for in-memory state management. *SEEP* decouples a streaming operators state from its processing logic, thereby making state directly manageable by the system via a well-defined set of primitive scale-out and fault-tolerance operations [17, 18]. *Naiad* extends the MapReduce model with support for structured cycles and streaming based on a timely dataflow model that uses logical timestamps for coordination [24]. *Samza* isolates multiple processors by localizing their state and disallowing them from sharing data, unless data is explicitly written to external storage [2]. Like S-Store, all of these systems treat state as mutable and explicitly manageable, but since they all focus on analytical and cyclic

dataflow graphs, they do not provide inherent support for transactional access to shared state.

There are a number of systems have explicitly been designed for handling *hybrid workloads* that include streaming. *Spark Streaming* extends the Spark batch processing engine with support for discretized streams (D-Streams) [28]. All state is stored in partitioned, immutable, in-memory data structures called Resilient Distributed Datasets (RDDs). Spark Streaming provides exactly-once consistency semantics, but is not a good fit for transactional workloads that require many fine-grained update operations. *Microsoft Trill* is another hybrid engine designed for a diverse spectrum of analytical queries with real-time to offline latency requirements [14]. Trill is based on a tempo-relational query model that incrementally processes events in batches organized as columns. Like Spark Streaming, its focus lies more on OLAP settings with read-mostly state. Last but not least, the *Google Dataflow Model* provides a single unified processing model for batch, micro-batch, and streaming workloads [8]. It generalizes the windowing, triggering, and ordering models found in MillWheel [7] in a way to enable programmers to make flexible tradeoffs between correctness and performance.

8 Conclusion

In this paper, we have described an approach to stream processing for applications that have shared, mutable state. These applications require guarantees for correct execution. We discussed ACID guarantees as in OLTP systems. We also described the idea of exactly-once processing, exactly-once delivery, and transactional workflows that obey ordering constrains as expressed in a dataflow graph. The paper also describes how we implement these guarantees on top of the H-Store OLTP main-memory database system.

In the future, we intend to look at extending our single-node prototype to run in a multi-node environment. This, of course, will preserve the guarantees mentioned above. We will re-examine recovery for our distributed extensions.

We are also studying how to adapt S-Store to effectively act as a real-time ETL system. Rather than loading data from flat files, S-Store will accept batches of tuples and install them transactionally in a persistent data store (either within S-Store or externally). During this process, its stored procedures can perform data cleaning and alerting. Each batch, possibly from multiple sources, must be processed to completion or not all. Furthermore, as tuples are being loaded, other transactions should not be allowed to see a partially loaded state. S-Store's ability to manage shared state makes it an ideal candidate for real-time ETL.

Acknowledgments. This research was funded in part by the Intel Science and Technology Center for Big Data, and by the NSF under grants NSF IIS-1111423 and NSF IIS-1110917.

References

[1] Apache Flink. `https://flink.apache.org/`.
[2] Apache Samza. `http://samza.apache.org/`.
[3] Esper. `http://www.espertech.com/esper/`.
[4] Trident Tutorial. `https://storm.apache.org/documentation/Trident-tutorial.html`.
[5] VoltDB. `http://www.voltdb.com/`.
[6] D. Abadi et al. Aurora: A New Model and Architecture for Data Stream Management. *VLDB Journal*, 12(2), 2003.
[7] T. Akidau et al. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. *PVLDB*, 6(11), 2013.
[8] T. Akidau et al. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *PVLDB*, 8(12), 2015.
[9] A. Arasu et al. STREAM: The Stanford Data Stream Management System. In *Data Stream Management: Processing High-Speed Data Streams*, 2004.
[10] M. Balazinska et al. Fault-tolerance in the Borealis Distributed Stream Processing System. *ACM TODS*, 33(1), 2008.

[11] I. Botan et al. SECRET: A Model for Analysis of the Execution Semantics of Stream Processing Systems. *PVLDB*, 3(1), 2010.
[12] N. Jain et al. Towards a Streaming SQL Standard. *PVLDB*, 1(2), 2008.
[13] I. Botan et al. Transactional Stream Processing. In *EDBT*, 2012.
[14] B. Chandramouli et al. Trill: A High-Performance Incremental Query Processor for Diverse Analytics. *PVLDB*, 8(4), 2014.
[15] S. Chandrasekaran et al. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR*, 2003.
[16] A. Elmore et al. A Demonstration of the BigDAWG Polystore System. *PVLDB*, 8(12), 2015.
[17] R. C. Fernandez et al. Integrating Scale-out and Fault-tolerance in Stream Processing using Operator State Management. In *SIGMOD*, 2013.
[18] R. C. Fernandez et al. Making State Explicit for Imperative Big Data Processing. In *USENIX ATC*, 2014.
[19] J.-H. Hwang et al. High-Availability Algorithms for Distributed Stream Processing. In *ICDE*, 2005.
[20] Institute of Medicine of the National Academies. Preventing Medication Errors . https://iom.nationalacademies.org/̃media/Files/Report%20Files/2006/Preventing-Medication-Errors-Quality-Chasm-Series/medicationerrorsnew.pdf.
[21] R. Kallman et al. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *PVLDB*, 1(2), 2008.
[22] S. Kulkarni et al. Twitter Heron: Stream Processing at Scale. In *SIGMOD*, 2015.
[23] J. Meehan et al. S-Store: Streaming Meets Transaction Processing. *PVLDB*, 8(13), 2015.
[24] D. G. Murray et al. Naiad: A Timely Dataflow System. In *SOSP*, 2013.
[25] L. Neumeyer et al. S4: Distributed Stream Computing Platform. In *KDCloud*, 2010.
[26] PhysioNet. MIMIC II Data Set. `https://physionet.org/mimic2/`.
[27] A. Toshniwal et al. Storm @Twitter. In *SIGMOD*, 2014.
[28] M. Zaharia et al. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *SOSP*, 2013.

"The Event Model" for Situation Awareness

Opher Etzion[1], Fabiana Fournier[2], and Barbara von Halle[3]

[1]Information Systems Department, Yezreel Valley College, Israel, opher.etzion@gmail.com
[2] IBM Research – Haifa, Haifa University Campus, Haifa 3498825, Israel, fabiana@il.ibm.com
[3] Sapiens International Corporation , Barbara.vonHalle@sapiens.com

Abstract

The Event Model (TEM) is a novel computation-independent model targeted at helping non-programmers to define and manage the logic of event-driven applications. The model design is based on a collection of building blocks that comprise a set of diagrams and normalized tables to define the event business logic of an application, a set of principles that define the set of assertions that a correct model should satisfy, and a glossary to express all the business concepts. The validity of the TEM model created is checked and guaranteed through a related set of integrity principles, and automatically translated to execution by the code generator. In this paper we concentrate on the model itself. The concepts and facilities of the model are demonstrated through an example taken from the Cold Chain Management (CCM) domain. Preliminary tests in the scope of transport and logistics indicate that the tables and diagrams in TEM are well accepted and embraced by non-technical people, who stress the ease and friendly manner of defining the event logic as the main benefit of TEM.

Keywords: Event-driven applications, model driven engineering, computational independent model, conceptual modeling, real-time business intelligence.

1 Introduction

In this paper we present The Event Model (TEM), a novel way to model, develop, validate, maintain, and implement event-driven applications. The Event Model follows the Model Driven Engineering approach [1, 3] and can be classified as a CIM (Computation-Independent Model), providing independence in the physical data representation and implementation details, omitting details that are obvious to the designer. This model can be directly translated to an execution model (PSM-Platform-Specific Model in the Model Driven Architecture terminology) through an intermediate generic representation (PIM-Platform-Independent Model).

TEM is based on a set of well-defined principles and building blocks, and does not require substantial programming skills, therefore targets non-technical people. In this paper we bring a high overview of TEM and focus on the main building blocks that constitute a TEM model, that is TEM diagrams and logic tables. In TEM, the event derivation logic is expressed through a collection of normalized tables. These tables can be automatically validated and transformed into code. This idea has already been successfully proven in the domain of business rules by The Decision Model (TDM) [20]. The Decision Model groups the rules into natural logical groups to create a structure that makes the model relatively simple to understand, communicate, and manage.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

We illustrate the model throughout this paper using a scenario from the Cold Chain Management (CCM) domain. The example employed is a simplified version, yet representative, of a real-word use case in that domain. A *cold chain* is defined as a temperature controlled supply chain. One of the major issues in this field is the transportation of temperature sensitive products through thermal and refrigerated packaging methods and the logistical planning to protect the quality of these shipments. Examples of cold chain products are fruits and vegetables, pharmaceuticals, and technology products. The cold chain serves the function of keeping food fresh for extended periods and eliminating doubts over the quality of the food products. Unfortunately, about 25% of all food products transported in the cold chain are wasted each year due to breaches in integrity that cause fluctuations in temperature and product degradation[1]. *In our scenario, John Cool is the quality control officer at NeverRotten Ltd. He is in charge of setting control policies for the online monitoring of the company's cold chain products. John's task is to detect a potentially dangerous condition of a container before an actual product quality disqualification takes place, thus remediation actions can be taken, saving time and money. To this end, John wants to define two main policy rules:*

- *Alert me when, inside a container, the temperature is in the permitted range constantly increases for the last 5 minutes.*
- *Alert me whenever a delay longer than permitted occurs.*

We show how TEM can help John Cool to easily create the logic needed to monitor any delays and temperature changes in a cold chain container to achieve the goals stated above.

2 TEM in a Nutshell

This section provides a high level view of The Event Model. We discuss its origins, design goals, building blocks, and basic concepts.

2.1 TEM and Concept Computing

TEM follows the paradigm of concept computing [6], according to which all model artifacts are concepts. A *concept* is a meaningful term within the user's domain of discourse. The model consists of concepts and *semantic relationships* among concepts. These concepts are based on the user's cognitive terms, and are independent of the IT terms or implementation. The vision is to strive for automatic transformation along with model-driven engineering; this approach contrasts with the current state of practice in which the transformations between the three levels of models are mostly done manually. The vision is to have a concept-oriented model and transform it in a mostly automated fashion to create an execution model. Concept computing belongs to the family of executable specifications, which has been studied in different domains [1]. While the concept computing vision aims at simplification, the model still needs to be expressive enough to allow this automatic transformation. The success of such a model in the event-driven domain depends on the level of simplification relative to existing event-driven models. In the construction of TEM we employed some simplification goals, as discussed below.

2.2 TEM Simplification Goals

After observing and experiencing the relative complexity of event processing tools, we wanted to define simplification goals for the design of TEM so it can used by non-IT experts. In this section we outline these simplification goals.

[1] http://people.hofstra.edu/geotrans/eng/ch5en/appl5en/ch5a5en. html

1. **Stick to the basics** by eliminating technical details. Looking at designs and implementations of event-driven applications, we observe two types of logic: the *application logic*, which directly states how derived events are generated and how the values of their attributes are assigned, and *supporting logic,* which is intended to enrich events or query databases as part of the processing. In our CCM example, the temperature range can be reported as part of an event and is either produced by the sensor or enriched later by an external database. Alternatively, it may not be part of an event but rather a result of a query executed during the evaluation of a pattern from either a database or a global variable store. The first simplification design goal is to view the concept of "temperature range" as a concept that is **obvious** in the designer's terminology and thus eliminate the supporting logic of where its value resides and how it should be fetched; we move that aspect "behind the scenes". These details can be inferred automatically during the code generation phase.

2. **Employ top down, goal-oriented design**. Many design tools require logical completeness (such as referential integrity) at all times. This requires building the model in a bottom-up fashion; namely, all the meta-data elements must be defined (events, attributes, data elements) before using them in the logic definition. Our second simplification design goal is to support top down design, and allow temporary inconsistency. We allow work in the "forgive" mode [9], in which some details may be completed at a later phase. This design goal complements the "stick to the basics" goal, by concentrating on the business logic first, and completing the data aspects later.

3. **Reduce the number of logical artifacts**. In a typical event processing application, there may be multiple logical artifacts, including event processing agents, queries, or processing elements, depending on the programming model that specifies the derivation logic of a single derived event. This variety arises when there are multiple ways to create a single derived event. In our CCM example there might be different circumstances in which a delay is detected. Our design goal is to have a single logic artifact for every derived event that accumulates all the ways to derive this event. This goal reduces the number of logical artifacts and bounds it by the number of derived events. It also eases the verifiability of the system, since possible logical contradictions are resolved by the semantics of this single logical artifact.

4. **Use fact types as first class citizens in the model**. In many of the conceptual models that are descendants of the Entity Relationship model [12], terms are modeled as attributes that are subordinates of entities or relationships. In some cases, it is more intuitive to view these concepts as "fact types" and make them first class citizens of the model, so the entity or event they are associated with is secondary (and may be a matter of implementation decisions). This requirement is again consistent with the "stick to the basics" goal.

2.3 TEM building blocks

TEM is composed of two main building blocks that relate to the model itself and are the main focus of this paper. These are the diagrams (Section 3) and the logical concepts (Section 4). Additional building blocks of the model are:

- TEM Glossary: The concept dictionary used for the interpretation of a specific application.
- Integrity principles: The principles that govern the model integrity.
- Code generator: The automatic translator of a model to executable code. The code generator is able to infer information that is not explicitly stated in the model, according to the stick to the basics principle.

3 TEM Diagrams

One way to simplify the model is to apply a top-down methodology that provides a high level logical view and understanding of the system at hand.

A TEM diagram illustrates the structure of the logic by showing a situation along with the flow direction of derivations in a top-down manner. At the top of the diagram there is a goal, which is the situation that is required to be derived. This goal is connected with the raw and derived events that are identified as participants in the situation derivation. This representation is done in a recursive way until raw events or facts are encountered , as depicted in Figure 2 for our CCM example.

A TEM diagram includes nine icons that express all the relevant terms (Figure 1).

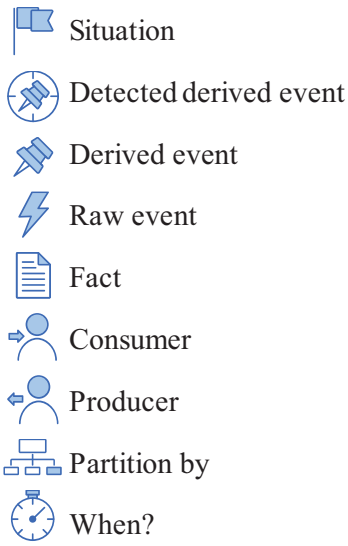


Figure 1: Product quality deterioration logic EDT

Each block in the diagram (a set of rectangle shapes, separated by connecting lines) represents a specific piece of logic with a single corresponding *Event Derivation Table* as explained in Section 4.1. The red rectangles in the background of each block represent the context for the block. The contexts can be collapsed or expanded. Dotted lines specify event flows to and from the event-driven system.

Figure 2 depicts the TEM diagram for the *Product quality deterioration* situation in our CCM example. The situation to be derived is a potential risk to the product quality, which requires alert notification and possible intervention. We have one consumer of the situation (*Quality control officer*, who gets the system alerts) and two producers: *Sensors* that emit the Sensor input; and *Shipment operations system*, which emits *Shipment starts* and *Shipment planned* raw events. The *Context* part of the *Shipment delay* derived event is expanded in the diagram to show a temporal context that is initiated when a shipment starts and ends at the *shipment planned time*, incremented by a *delay tolerance*. The *delay tolerance* indicates a grace period that is calibrated according to the specific situation. Sometimes a delay of a minute can be considered a problem, while in other cases, only a delay of a few days from the planned time is considered a situation that requires an action. We partition the events according to the *Shipment ID* domain fact type since we are looking for delays at the level of the shipment ID. Domain Fact Types serve as abstract fact types to enable segmentation contexts.

For each situation in TEM, there is a corresponding TEM diagram. The diagrams serve as a major design tool that provides a top down view. All blocks that describe situations or derived events require the definition of logical concepts.

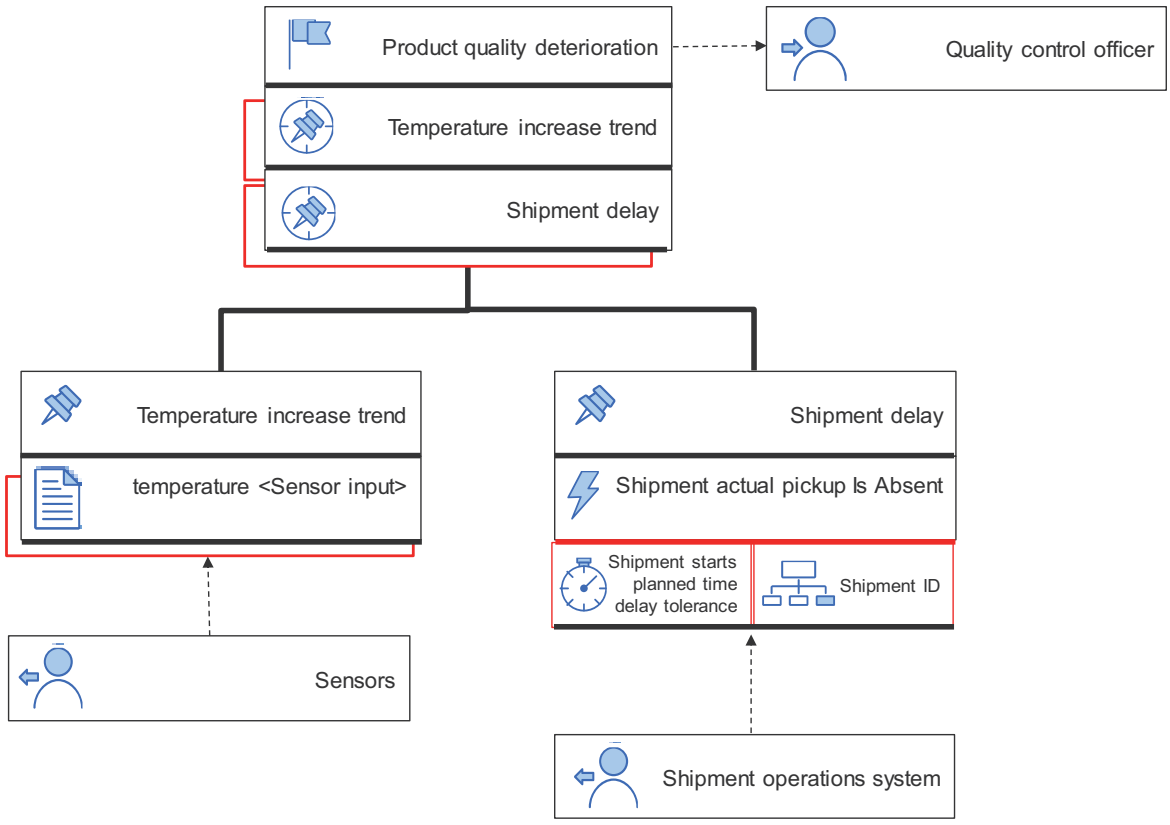


Figure 2: Temperature increase trend logic EDT

4 TEM Logical Concepts

Logical concepts are descriptions of concepts that are computed by the described application. The Event Model Logic consists of two logical concept types which are represented as tables.

Event Derivation: A single logical artifact for each derived event. The derived event mentioned in the name is associated with the table in the sense that the table specifies the conditions for generation of new instances of this event type.

Computation logic: A logical artifact that specifies the computation of assignments of the values of fact types (attributes) associated with a derived event. The derived fact type mentioned in the name is associated with the table in the sense it describes the value assignment for its fact types. Note that if the value of a derived fact type can be implicitly inferred, then the computation table for this derived fact type can be omitted.

Although the names of concepts in TEM can be determined freely by the system designer, we use some naming conventions in the logic tables for the sake of clarity. For example, domain fact types as well as event types start with a capital letter; fact types start with a lowercase letter. We also underline event types in condition columns that have an Event Derivation Table of their own (hyperlinks), to stress the fact that these events are themselves derived from another piece of logic, and enabling users to follow paths of inference by clicking these links.

We describe TEM logic tables in more detail in the following sections.

4.1 TEM Event Derivation Tables

An *Event Derivation Table (EDT)* is a two-dimensional representation of logic leading to a derived event, based on events and facts. Thus, an EDT designates the circumstances under which a derived event of interest is

reached. In our CCM scenario there are three EDTs shown in Table 1, Table 2, and Table 3 that correspond to the same names in the TEM diagram.

4.1.1 Event Derivation Tables Structure

The first row in an EDT indicates its name. The EDT name is the *derived event name* + "Logic", for example, *Product quality deterioration* Logic in Table 1. The table consists of two parts, context and conditions, separated by a red line. The context part consists of two logical sections. The temporal context, represented by *When expression, When start,* and *When end* columns; and the segmentation context represented by the *Partition by* column. For example, Table 2 describes a non-overlapping sliding fixed interval temporal context [10] of 5 minutes' length and a segmentation context that partitions the events by Container ID domain.

Table 1: Product quality deterioration logic EDT

Product quality deterioration **Logic**												
Row #	**When Expression**	**When Start**	**When End**	**Partition by**	**Filter on event**		**Pattern**				**Filter on pattern**	
				Shipment ID			*Temperature increase trend*		*Shipment Delay*			
1	*always*			*same*			*is*	*Detected*				
2	*always*			*same*					*is*	*Detected*		

Table 2: Temperature increase trend logic EDT

Temperature increase trend **Logic**										
Row #	**When Expression**	**When Start**	**When End**	**Partition by**	**Filter on event**		**Pattern**		**Filter on pattern**	
				Container ID	*temperature*		*temperature*			
1	*for every 5 minutes*			*same*	*is between*	*lower bound, upper bound*	*is*	*Increasing*		

Table 3: Shipment delay logic EDT

Shipment delay **Logic**									
Row #	**When Expression**	**When Start**	**When End**	**Partition by**	**Filter on event**	**Pattern**			**Filter on pattern**
				Shipment ID		*Shipment actual pickup*			
1		*Shipment starts*	*planned time + delay tolerance*	*same*		*is*	*Absent*		

4.1.2 Event Derivation Tables Conditions

The conditions part consists of three types of conditions. The conditions are logically applied in the following order.

Filter conditions are expressions evaluated against the content of a single event instance. The role of filter conditions is to determine whether an event instance satisfies the filtering condition and should participate in the derivation. For example, the *Filter on event* column in

Table 2 describes a condition on a fact type *temperature*, which belongs to the *Sensor input* event type. The temperature value must be between predefined bounds in a certain range.

Pattern conditions are expressions on related event types' instances such as Detected, Absent, Thresholds over Aggregations, or Fact Type value changes [10]. The role of pattern conditions is to detect the specified relationships among event instances. For example, in Table 3, the Pattern condition describes an absence detection of event type *Shipment actual pickup,* which means that no event instance of that event type is detected within the specified context.

Filter on pattern conditions are expressions on multiple event occurrences, including comparisons, memberships, and time relationships. The role of the filter on patterns conditions is to filter the pattern result based on conditions among the different events that participates in this pattern. Following the CCM example, let us assume the following scenario: we want to identify whether a shipment was picked up more than two hours after the planned time. We name this derived event *Significant shipment delay*. In this case, the pattern is *Shipment actual pickup* ***occurs after*** *Shipment planned pickup*. The filter on the pattern condition will be expressed as the difference between *shipment planned pickup time* and the *shipment actual pickup time* is greater than two hours (see Table 4).

Table 4: Example of a filter on pattern conditions

Significant shipment delay **Logic**											
Row #	**When Expres**	**When Start**	**When End**	**Partition by**	**Filter on event**		**Pattern**		**Filter on pattern**	
				Shipment ID			*Shipment actual pickup*		*occurrence time of Shipment actual pickup*	
1	*always*			*same*			*occurs after*	*Shipment planned pickup*	*is greater than*	*planned time+2*

The three types of conditions are optional, meaning they can either appear or not in an EDT, however a TEM model is valid if it contains at least one condition. We also do not restrict the number of conditions per condition type. For example in Table 2, we can add a new condition to the Pattern which specifies that in addition to checking whether the temperature value is increasing, we also check that we have at least three Sensor input events in the same Context.

The EDTs have disjunctive normal form (DNF) semantics. Each row in the table indicates a different set of circumstances in which the same event can be derived; therefore, the derived event logic is the union of the rows (logical OR relationship). On the other hand, in each row all conditions in the columns must be satisfied, therefore the columns satisfy an AND logical relationships. For example, as described in Table 1, the *Product quality deterioration* event can be derived when either a *Temperature increase trend* event is detected or a *Shipment delay* event is detected.

TEM connection is a dependency among EDTs when the conclusion, i.e., derived event, of one EDT is referenced in another EDT. Connections are shown in the TEM tables as underlines or hyperlinks. For example, *Temperature increase trend* and *Shipment delay* events are underlined in Table 1 since they are conclusions of *Temperature increase trend logic* and *Shipment delay logic EDTs*, respectively.

4.2 TEM Computation Tables

A derived event, like any event in TEM, is a container that contains facts (attributes) which are instances of the fact types contained in the derived event's event type. Part of the derivation is the assignment of values to these facts. Some of the computed facts are mere copies of values. Thus, according to the simplification goal of **stick to the basics**, their computation details may be omitted and their computation assignment is implicit. A *Computation Table* is a two-dimensional representation of logic leading to a *computed fact type* that needs to be explicitly specified. Let's assume that the *Shipment delay* derived event type has two associated fact types: *Shipment ID* and *Delay message*. The value of *Shipment ID* is computed in an obvious way, namely, by copying the value of the specific partition argument. The *Delay message* has to be explicitly computed, as shown in Table 5. Likewise, Table 6 shows the computation of the two possible alert messages associated with the *Product quality deterioration* situation (see explanation below). Note that the "+" sign denotes string concatenation.

Table 5: Delay message computation table

delay message **Computation**		
Row #		**Row in Event derivation Table**
1	*"Shipment " + Shipment ID+ " pickup time is delayed in " +delay tolerance+ "minutes "*	*1*

Table 6: Alert message computation table

alert message **Computation**		
Row #		**Row in Event derivation Table**
1	*"the temperature in container" + Container ID + "constantly increases within the last 5 minutes"*	*1*
2	*delay message*	*2*

4.2.1 Structure of Computation Tables

The first row in a computation table indicates its name, composed of the *fact type name* + "Computation". For example, Table 6 is a computation table that describes the logic to compute the *alert message* fact type associated with the *Product quality deterioration* event type. The second row is the headings row. The third row and on, include the row number, the expression value of the computed fact type, and a reference to the row number in the corresponding EDT.

Looking at *Product quality deterioration* EDT in Table 1, there are two cases in which the *Product quality deterioration* event type can be derived. One is Shipment delay and the other is Temperature trend increase. Each case dictates a different value to the computed fact type *alert message*. Table 6 contains the two possible values that can be assigned. The first row refers to the case in which a *Temperature increase trend* occurred, since the "row in event derivation table" *Shipment Delay* equals 1.

There is only one case in which the *Shipment delay* event type can be derived as shown in Table 3. In this case, the alert includes the delay elapsed time as computed in Table 5.

While the logic artifacts may be defined first, the glossary concepts eventually need to be completed at a later phase, prior to the model's validation

5 Related Work

In this section we briefly survey work related to TEM in several areas: event processing modeling, semantic modeling of events, and executable specifications.

In the area of event processing modeling, Cugola and Margara provide [5] a comprehensive survey and comparison of models, including aspects of the functional model, processing model, deployment model, interaction model, data model, time model, and rule model. In general, the event processing models contain "programming in the large" modeling, which is typically an event flow model [10] or stream processing model [8]. The "programming in the small" model is closely related programming models such as stream modeling [15] and rule based modeling [2]. Some of the modeling languages employ visualization (i.e. of the event flows) [16]. Another branch of event modeling is based on logic programming. Models in this area follow Kowalski's event calculus model [14].

The main novelties of TEM relative to existing event models are mainly two. First, it is targeted to non-technical people. This is enabled by applying a top-down approach that satisfies the simplification goals and supporting the creation of a specification without providing technical and "obvious" details, such as location of data-items. Second, TEM provides direct path to automatic implementation. This is a departure from current event models that are closely related to the implementation scheme.

The area of semantic data models [17] deals with the semantics of data and relationships among data elements. Most models follow the entity-relationship approach (ER) and its descendent methods (EER). Fidalgo et al., present a recent work [12] in which entities and relationships are first class citizens and attributes are secondary. Fact models [18] take business concepts as first class citizens, and data as containers for these facts. Our model follows the fact modeling approach, which has not been investigated yet in the area of event modeling.

The idea of executable specification was introduced in the early days of software engineering, for example by Urban et al. [19]. TEM can be considered an instance of this concept.

The Decision Model (TDM) [20] is an instance of a model that has similar goals in a different domain (decision management). The main difference between TDM and TEM is that TDM models the inference of computed values of facts as a function of other facts, while TEM models the logic of derivation of events in an event-driven context-based fashion.

6 Conclusions and Future Work

This paper presents The Event Model (TEM). TEM is a novel way to develop and implement event-driven applications. The friendly, yet rigorous, representation of the event logic enables the model to be simpler relative to existing models and accessible to people lacking IT skills. We illustrated the main logic concepts and artifacts of TEM using an example from the CCM domain. Experiments conducted in the scope of transport and logistics indicate that the tables and diagrams in TEM are well accepted and embraced by non-technical people, who stress the ease and friendly manner of defining the event logic as the main benefit of TEM. We believe that these preliminary tests are a good indicator of TEM's potential to open a new era for the consumption and pervasiveness of event-driven applications. In order to prove this statement, further experimentation is required including different domain areas and more complex scenarios.

The simplification design goals stated at the beginning of this paper have been realized as summarized in Table 7.

There are several model extensions, which are either progress or planned:

1. Support for current missing functionality, such as spatial patterns and contexts, pattern policies, and temporal correctness guards.

Table 7: Realization of simplification design goals

	Simplification goal	Realized by
1	Stick to the basics by eliminating technical details	The derivation and computation logic does not contain any logic of data fetching. This is either inferred or completed at a later phase. Assignments of values to attributes of derived events, whose assignment is obvious since they are copied from the context data, can be inferred by the system and does not have to be explicitly defined as part of the logic.
2	Employ top down, goal oriented design	The methodology supports top down, goal-oriented design by making the goal-oriented diagram a starting point. The logic tables are built in "forgive" mode, enabling reference to glossary artifacts prior to their definition.
3	Reduce the quantity of logic artifacts	The normalization principle, according to which there is a single EDT for each derived event, bounds the number of logic artifacts.
4	Use fact types as first class citizens in the model	Fact type is the fundamental basic unit in the model.

2. Support for non-functional requirements: The idea is to extend TEM to model non-functional requirements. Note that there have been some studies of high level modeling of non-functional requirements [4].

3. Extend the model to tangent activities: modeling the process of instrumentation and modeling goals for optimization based decisions.

4. Extend the model to support artifact based business state-oriented processing [13].

In addition, we are carrying out more work in model validation using constraint satisfaction techniques [7], and in code generation for various languages.

7 Acknowledgments

Fabiana Fournier has received funding from the European Union's Seventh Framework Programme FP7/2007-2013 under grant agreement 619491 (FERARI).

References

[1] Bodenstein C., Lohse F., and Zimmermann A. 2010. *Executable Specifications for Model-Based Development of Automotive Software*. SMC 2010, 727-732.

[2] Bragaglia S., Chesani F., Mello P., and Sottara D. 2012. *A Rule-Based Calculus and Processing of Complex Events*. RuleML 2012, 151-166.

[3] Brambilla M., Cabot J., and Wimmer M. 2012. *Model Driven Software Engineering in Practice*. Morgan & Claypool.

[4] Chung L and Leite C.J.P. 209. *On Non-Functional Requirements in Software Engineering*. Conceptual Modeling: Foundations and Applications (2009), 363-379.

[5] Cugola G., and Margara A. 2012. *Processing flows of information: From data stream to complex event processing*. ACM Comput. Surv. (CSUR) 44(3).

[6] Davis M. 2012. *Concept Computing: Bringing Activity-Context Aware Work & Play Spaces into the mainstream*. Keynote presentation from the Association for the Advancement of Arti?cial Intelligence 2012 conference (AAAI 12). URL: http://www.slideshare.net/Mills/understanding-concept-computing

[7] Dechter R. 2003. *Constraint Processing*. Elsevier.

[8] Dindar N, Tatbul N., Miller R.J., Haas L.M., and Botan I. 2013. *Modeling the execution semantics of stream processing engines with SECRET*. VLDB J. (VLDB) 22(4), 421-446.

[9] Etzion O. 1993. *Flexible consistency modes for active databases applications*. Inf. Syst. (IS) 18(6), 391-404.

[10] Etzion O. and Niblet P. 2010. *Event processing in action*. Manning.

[11] Farahbod R., Gervasi V., and Glässer U. 2014. *Executable formal specifications of complex distributed systems with Core ASM*. Sci. Comput. Program. (SCP) 79, 23-38.

[12] Fidalgo R., Alves E., España S., Castro, and Pastor J.O. 2013. *Metamodeling the Enhanced Entity-Relationship Model*. JIDM 4(3), 406-420.

[13] Heath F., Boaz D., Gupta M., Vaculín R., Sun Y., Limonad L., and Hull R. (2013) *Barcelona: A Design and Runtime Environment for Declarative Artifact-Centric BPM*. ICSOC 2103, 705-709.

[14] Kowalski R.A. 1991. *Logic Programing in Artificial Intelligence*. IJCAI (1991), 596-604.

[15] Jacques-Silva G., Kalbarczyk Z., Gedik B., Andrade H., Wu K-L., and Iyer R.K. 2011. *Modeling stream processing applications for dependability evaluation*. DSN 2011, 430-441.

[16] Marquardt N., Gross T., Sheelagh M., Carpendale T., and Greenberg S. 2010. *Revealing the invisible: visualizing the location and event flow of distributed physical devices*. Tangible and Embedded Interaction, 41-48.

[17] Peckham J. and Maryanski F.J. 1988. *Semantic Data Models*. ACM Comput. Surv. (CSUR) 20(3), 153-189.

[18] Ross R.G. 2000. *What Are Fact Models and Why Do You Need Them* (Part 1). Business Rules Journal, 1(5) URL: http://www.BRCommunity.com/a2000/b008a.html.

[19] Urban S.D., Urban J.E and Dominick W.D. 1985. *Utilizing an Executable Specification Language for an Information System*. IEEE Trans. Software Eng. (TSE) 11(7), 598-605.

[20] Von Halle, B., and Goldberg L. 2010. The Decision Model. CRC Press.

Towards Adaptive Event Detection Techniques for the Twitter Social Media Data Stream

Michael Grossniklaus, Marc H. Scholl, and Andreas Weiler
Department of Computer and Information Science
University of Konstanz, Germany
firstname.lastname@uni-konstanz.de

Abstract

Social media data streams are an invaluable source for timely and up-to-date information about current events. As a consequence, several event detection techniques have been proposed in the literature in order to tap this information source. However, most of these proposals focus on the information extraction aspect of the problem and tend to ignore the streaming nature of the input. The work conducted in our research group therefore intends to address these stream-related challenges, such as detecting events incrementally, reporting them in (near) real-time, and coping with fluctuations and spikes in the social media data stream. In this article, we report on the results that we obtained so far and outline our research agenda for the remainder of this work.

1 Introduction

Twitter currently has 320 million monthly active users who author over 500 million *tweets* per day that consist of up to 140 characters each.[1] These impressive usage statistics make Twitter the most popular and fastest-growing microblogging service on the planet. In the domain of social media, microblogging enables users to send short messages, links, and audiovisual content to a network of *followers*, as well as to their own public timeline. Due to their brevity, tweets are an ideal mobile communication medium, which is evidenced by the fact that 80% of Twitter's active users are on mobile devices. As a consequence, several proposals have been made to leverage social media data streams as "social sensors" [15] in order to obtain information about current events as they unfold. For example, Twitter data has been used to alert people in case of an outbreak of an infectious disease [9], to quickly respond to natural disasters [15], and to monitor political elections [21].

The problem of detecting events in text-based corpora is not a novel one and has been addressed by research from the area of Topic Detection and Tracking (TDT) for traditional media such as newspaper archives and news websites. In these domains, an *event* is defined as a real-world occurrence that takes place in a certain geographical location and over a certain time period [3]. In comparison to these information sources, social media data streams such as Twitter introduce additional challenges. First, tweets are much shorter than traditional documents and therefore harder to classify. Second, tweets do not undergo an editorial process and can thus

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

[1] `https://about.twitter.com/company` (November 24, 2015)

contain a substantial amount of spam, typos, slang, *etc*. Finally, the rate at which tweets are produced is very bursty and continually increases as more people adopt Twitter every day.

The techniques that have been proposed for event detection in social media and, in particular, for Twitter have more or less focused exclusively on the information extraction aspect of the problem. Because of this research direction, the challenges that are related to the streaming nature of the input data have so far been largely ignored by these approaches. For example, many techniques use (large) tumbling windows to process the stream, rather than online or streaming algorithms, and are therefore often unable to report events in (near) real-time. Furthermore, event detection often depends on a complex set of parameters, such as thresholds that control what is considered to be an event. Existing approaches typically assume that these parameters can be calibrated empirically by running the technique on sample data until it produces the desired result. Since the data in the stream may change both qualitatively and quantitatively over time, we argue that techniques that are based on fixed parameters are neither realistic nor feasible.

The work that our research group conducts on this topic intends to address this need for *streaming* and *adaptive* event detection techniques for Twitter. Due to this focus, our work is situated in the area of Data Stream Management Systems (DSMS) research. Since event detection and tracking is a vast field of research in itself, we concentrate on the specific task of first story detection, *i.e.*, the detection of general (unknown) events, which has been defined as a subtask of TDT [3]. In this article, we report results that we obtained so far and outline future research directions. We begin in Section 2 by giving a brief overview over the state of the art in event detection techniques for Twitter, including our own. Section 3 presents an evaluation platform that supports the systematic study and comparison of such techniques. In our work, we use this platform in order to gain a better understanding of how different parameter settings affect the trade-off between processing time and result quality in existing event detection techniques. In Section 4, we outline how this empirical research will contribute to building event detection techniques that can adapt to content and volume changes in the social media data stream. Finally, we give concluding remarks in Section 5.

2 Event Detection Techniques

In recent years, numerous techniques to detect events in social media data streams and, in particular, Twitter have been proposed. Rather than presenting a comprehensive survey of event detection techniques, we introduce five examples in this section. The first three examples are existing approaches that we studied in detail in previous work [18, 19]. The remaining two examples are approaches that we proposed ourselves in an effort to develop techniques that process their input in a fully streaming and incremental manner. For a more detailed discussion of the state of the art, we refer the interested reader to one of the existing surveys on this subject. For example, the survey of Nurwidyantoro and Winarko [14] summarizes 11 techniques to detect disaster, traffic, outbreak, and news events. The survey of Madani *et al.* [12] presents 13 techniques that each address one of the four challenges of health epidemics identification, natural events detection, trending topics detection, and sentiment analysis. A more general survey with a wide variety of research topics related to sense making in social media data is presented by Bontcheva and Rout [7]. Finally, Farzindar and Khreich [10] conducted an extensive survey of techniques that are specifically intended to detect events in the Twitter social media data stream.

EDCoW (Event Detection with Clustering of Wavelet-based Signals) [21] is one of the most-cited event detection techniques. In the first step, this algorithm applies a time-based tumbling window of size s to the stream to partition it into non-overlapping segments. For each window instance, it then builds the DF-IDF signals for each distinct term in the segment. The DF-IDF is similar to the TF-IDF that is commonly used in information retrieval to measure the importance of a word (term). Since multiple occurrences of the same term in one document (tweet) are typically associated with the same event, the DF-IDF only counts the number of documents that contain the term. On each of these signals, a discrete wavelet analysis is performed in order to build a second signal in which each data point summarizes a sequence of values of length Δ from the first

signal. Trivial terms are filtered out in the next step by checking the corresponding signal auto-correlations against a threshold γ. A modularity-based graph partitioning technique is then applied to the remaining terms in order to form events by clustering them. Finally, another threshold ϵ is used to filter out insignificant events. In the original paper, EDCoW is evaluated on a month's worth of Twitter data that was gathered in June 2010 by collecting the tweets from the top 1000 Singapore-based users and their friends within two hops. The initial window size s was set to a whole day.

The *WATIS* (Wavelet Analysis Topic Inference Summarization) [8] event detection technique is similar to *EDCoW* in that it first segments the stream into time-based windows of size s and then builds the DF-IDF signals for each distinct term. However, before these signals are further analyzed, they are smoothed using an Adaptive Kolmogorov-Zurbenko (KZA) [22] low-pass filter that calculates a moving average with i_{kz} iterations over n intervals. Based on these smoothed signals a time-frequency representation is constructed using continuous wavelet transformation. On this representation two wavelet analyses are performed in order to detect unexpected shifts in the frequency of a term: the tree map of the continuous wavelet extrema and the local maxima detection. Finally, Latent Dirichlet Allocation (LDA) [6] with i_{lda} iterations is used to enrich event terms with co-occurring terms. The technique is evaluated by applying it to a dataset consisting of 13.6 million tweets, which were gathered over a period of eight days. In this evaluation, the technique was used to process the entire dataset at once, *i.e.*, the initial window has a size s of 192 hours.

As the previous approaches, *enBlogue* [4] uses a time-based tumbling window of size s to segment the stream before processing it.[2] For each window, so-called "seed tags" are identified based on their popularity, which is computed as the relative frequency of a term in a window. Topics are modeled as pairs of tags, which are formed by measuring the correlation between two documents that contain the tags using the Jaccard coefficient. A topic is considered to be an emergent event if its current behavior is different from its previous behavior, *i.e.*, if there is an unexpected shift in its popularity. All topics are then ranked according to their degree of emergence and the top k topics are reported as events. In the original evaluation, the size s of the initial window is set to one hour and the result quality of the detected events is assessed based on a user study.

To conclude this section, we present two simple event detection techniques that we developed in previous work. The goal of both techniques is to reduce the latency with which events can be reported, but each technique follows a different approach to do so. In contrast to the techniques described above, *LLH* [20] reduces the processing required to detect events. It simply calculates a log-likelihood measure for the frequency of all distinct terms in the current time-based tumbling window ($s = 1$ hour) against their frequency in the previous window. For the top N terms ranked according to this ratio, the corresponding top four most co-occurring terms are computed and the resulting term set is reported as an event. Our second technique, *Shifty* [17], aims to reduce latency by using both shorter and sliding windows to segment the stream. It detects events by monitoring the IDF values of distinct terms in successive sliding windows. For each term in a (tumbling) window of size $s = 1$ minute, Shifty computes the IDF value and filters out terms with an IDF value above the window average. In order to calculate the IDF shift for each remaining term from one window to the next, a window with size s_1 that slides with range r_1 is built in the next step. Only terms with a shift above the average shift are retained. In the last step, another sliding window with size s_2 that slides with range r_2 is built. This window is used to calculate the total shift value as the sum of all shift values of the sub-windows. Terms with a total shift value greater than Ω are detected as events and reported together with their top four co-occurring terms.

[2] In their original paper, Alvanaki *et al.* [4] state that *enBlogue* uses sliding windows. However, only the value for the size of the window is given, while the value for the slide range is never mentioned. Personal communication with one of the authors confirmed that indeed a tumbling window is used.

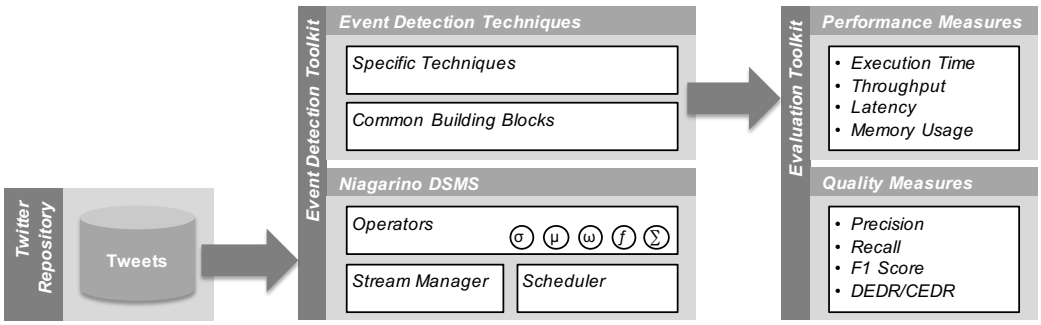


Figure 1: Overview of the evaluation platform for Twitter event detection techniques.

3 Evaluation Platform

In order to understand how our own approaches compete with the current state of the art, we designed and developed an evaluation platform for event detection techniques. Figure 1 gives a schematic overview of this platform and its components. The first component is a tweet repository that we host on our servers, which contains a randomly sampled 10% sub-stream of the public live stream of Twitter. The repository is continuously updated with new tweets that we have been gathering since 2012 using the Twitter Streaming API[3] with the so-called "Gardenhose" access level. At the moment, the repository contains about 10 TB of data, which corresponds to over 50 billion tweets at an average rate of 2.5 million tweets/hour.

The next component of our evaluation platform is a toolkit that can be used to experiment with existing and new event detection techniques in a controlled environment. In order to obtain reliable performance measurements that can be compared fairly, we propose to realize all studied event detection techniques in a DSMS. For this purpose, we currently use *Niagarino*[4], a lightweight and extensible DSMS that we develop and maintain in our research group. The main purpose of Niagarino is to serve as an easy-to-use research platform for streaming applications such as the ones presented in this article. Many of its concepts can be traced back to a series of pioneering data stream management systems, such as Aurora [2], Borealis [1], and STREAM/CQL [5]. In particular, Niagarino is an offshoot of NiagaraST [11], with which it shares the most common ground. The representation of event detection techniques as query plans is one of the key benefits of our approach. Using Niagarino's textual plan description format or the graphical plan builder that we are currently developing, new techniques can be easily developed by modifying existing plans or by creating new ones. In order to further simplify this task, our toolkit already provides a number of building blocks that are common to many event detection techniques, such as operators to tag tweets with their languages, to filter tweets that contain spam, and to remove terms that are considered noise or stop-words. Finally, additional operators that cannot be assembled from already existing ones can be added to our toolkit with limited programming overhead due to Niagarino's modular architecture.

The last component of our platform is a toolkit to evaluate event detection techniques. By providing this toolkit, we address two shortcomings of the current state of the art. First, very few authors of existing event detection techniques have evaluated the performance of their approach in comparison to other techniques. Nevertheless, factors such as throughput, latency, and memory usage are particularly crucial to the feasibility of an approach in a highly volatile streaming setting such as Twitter. Our toolkit therefore provides a number of measures that can be used to study and compare these performance characteristics of event detection techniques. Second, the quality of the results, *i.e.*, the validity of the detected events, is another factor that is paramount to the usefulness of an approach. While some authors of previous approaches have evaluated the results of their technique using a manually crafted ground truth or based on a user study, very few have compared their results

[3] `https://dev.twitter.com` (November 24, 2015)

[4] `http://www.informatik.uni-konstanz.de/grossniklaus/software/niagarino/` (November 24, 2015)

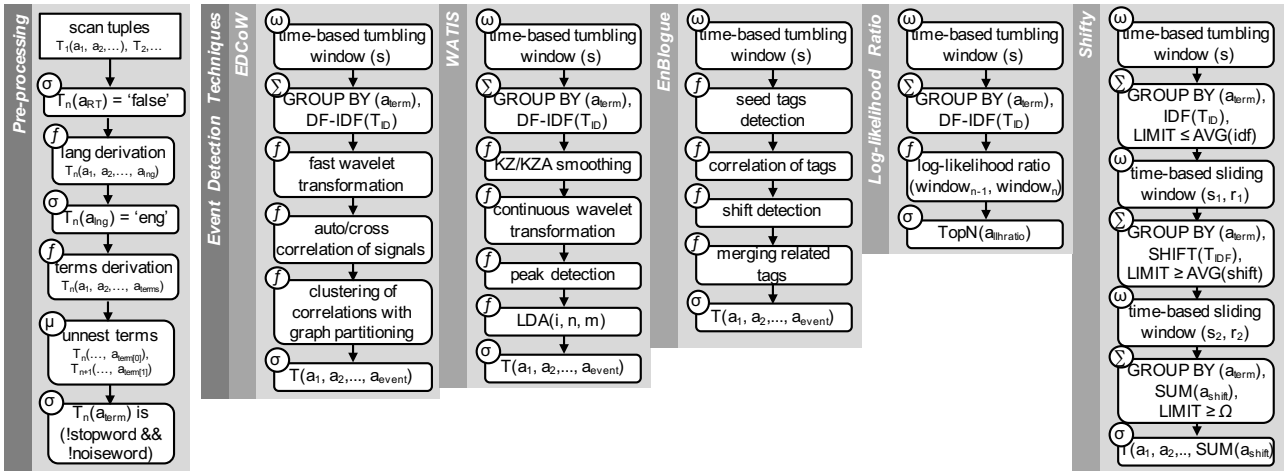


Figure 2: Niagarino query plans for the preprocessing and the five example event detection techniques.

to competing approaches. One reason for this lack of comparative and systematic evaluation is that crafting a ground truth manually does not scale to the volume of the Twitter data stream and conducting user studies is time-consuming and expensive. In our work [18, 19], we have therefore focused on quality measures that can be applied automatically. For example, we propose to measure precision by matching detected events to a combination of Web search-engine results and knowledge bases such as DBpedia[5]. We follow a similar approach to measure recall by crawling the daily headlines of new archives such as Bloomberg and the New York Times. Based on precision and recall, we are able to calculate the F_1 score for a studied technique. It is important to note that values computed by these measures cannot be used to support any absolute conclusions about a single technique. However, they can be used to draw relative conclusions by comparing different techniques or multiple configurations of the same technique.

We have used this platform to conduct an extensive study of the event detection techniques introduced in the previous section. Figure 2 shows the corresponding Niagarino query plans as well as the preprocessing subplan that is common to all approaches. As a complete discussion of the results is out of the scope of this article, we refer the interested reader to our previous work. Weiler *et al.* [18] presents the evaluation measures that we defined. In order to demonstrate that these measure are useful, we apply them to both well-known event detection techniques and baseline approaches. The comparison of the results clearly show that our measures can discriminate between actual event detection techniques and approaches that, for example, simply select random or most frequently occurring terms. In Weiler *et al.* [19], we use these measures to study a number of event detection techniques in terms of performance and result quality. With respect to result quality (F_1 score) our study confirms that the status of both *EDCoW* and *WATIS* as frequently cited event detection techniques is well-deserved as they detect events more reliably than other techniques. However, this result quality comes at the price of lower throughput (tweets/second). In particular, *WATIS* would not be capable of handling the full 100% stream of Twitter on current server hardware, owing to the expensive LDA operator towards the end of the query network. In contrast, our own techniques, *LLH* and *Shifty*, score very well with respect to this performance measure. While *LLH* scores quite low in terms of result quality, *Shifty* is a close runner-up behind the more complex event detection techniques. We therefore conclude that *Shifty* represents an interesting trade-off between performance and result quality that we will investigate further in the future.

[5] `http://dbpedia.org` (November 24, 2015)

4 Future Work

Building on the work presented in this article, we are currently conducting research to address the need for *adaptive* event detection techniques for the Twitter social media data stream. In order to do so, we follow two lines of work.

First, we are studying methods to automatically determine the parameter settings of event detection techniques. As outlined in Section 2, current techniques depend on a number of parameters that directly affect performance and result quality of an approach. The ability to determine and adjust these parameters automatically is important for several reasons. On the one hand, it is unrealistic to assume that such parameter values can be determined based on a small sample of the stream during the design of the technique. This assumption has often been criticized before, for instance by Farzindar and Khreich [10]. On the other hand, the social media data stream may undergo qualitative and quantitative changes, which require parameter adjustments. Using our implementations of existing techniques that we described in this article, we study the effects of different parameter settings for each technique on a number of segments of the real-life Twitter data stream. The goal of this initial empirical study is to develop quality-of-service models for selected techniques that describe the relationship between performance and result quality. Based on these quality-of-service models, we envision that adaptive techniques can trade-off result quality for performance in case of changes in the volume of tweets that need to be processed. In the past, quality-of-service models have been used successfully to control load shedding [16]. Rather than shedding load, we are interested in using such models to shed processing time, *i.e.*, to dynamically reconfigure techniques to perform, for example, fewer LDA iterations or low-pass filter steps.

Our second line of work researches new forms of content-based stream segmentation for event detection techniques. All existing techniques use (large) time-based windows to process the unbounded stream of tweets. In previous and ongoing work [13], we criticized the use of simple time and tuple-based windows in today's complex data-stream applications and instead proposed data-driven windows, so-called frames. We are interested in studying whether frames as a method to segment streams can contribute to better result quality of event detection techniques. The quality improvements that can be obtained with frames stem from the fact that frames adapt the segmentation of the stream to the observed data rather than segmenting it into predefined intervals as windows do. Therefore, in order to use frames in the setting of streaming social media data analysis, the data that can drive the framing of the stream need to be identified. Since a portion of the Twitter stream contains GPS coordinates, it could, for example, make sense to use a position grid to segment the stream to track how (information about) an event spreads geographically.

5 Summary and Conclusion

Since their inception, DSMSs have been used to realize ever more complex stream processing applications, which often demanded new or extended functionality at the system level. In this article, we focussed on event detection in social media data streams, a relatively new application domain for DSMSs. Unfortunately, most existing event detection techniques have been developed without the support of a DSMS, which makes it difficult to reason about their practical feasibility, in particular with respect to their performance. Therefore, we introduced some well-known event detection techniques in this article and showed how they can be realized as query plans in a DSMS. This representation is one of the key benefits of our approach as it greatly simplifies the creation and modification of event detection techniques. In order to further promote the use of DSMSs in researching such techniques, we have designed and developed a platform that provides toolkits for both the implementation and evaluation of existing and novel approaches. Finally, we outlined open research challenges in this area, such as the need for fully streaming and adaptive event detection techniques. We believe that tackling these challenges will again require new DSMS concepts as, for example, new methods to deal with changes in data volume or to segment the stream in a more flexible manner.

Acknowledgments

The research presented in this article is funded in part by the Deutsche Forschungsgemeinschaft (DFG), Grant No. GR 4497/4: "Adaptive and Scalable Event Detection Techniques for Twitter Data Streams". We would also like to thank our students Christina Papavasileiou, Harry Schilling, and Wai-Lok Cheung for their contributions to the implementations of the *WATIS*, *EDCoW*, and *enBlogue* event detection techniques in Niagarino.

References

[1] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley B. Zdonik. The Design of the Borealis Stream Processing Engine. In *Proc. Intl. Conf. on Innovative Data Systems Research (CIDR)*, pages 277–289, 2005.

[2] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stand Zdonik. Aurora: A New Model and Architecture for Data Stream Management. *The VLDB Journal*, 12(2):120–139, 2003.

[3] James Allan. *Topic Detection and Tracking: Event-based Information Organization*. Kluwer Academic Publishers, 2002.

[4] Foteini Alvanaki, Sebastian Michel, Krithi Ramamritham, and Gerhard Weikum. See What's enBlogue: Real-time Emergent Topic Identification in Social Media. In *Proc. Intl. Conf. on Extending Database Technology (EDBT)*, pages 336–347, 2012.

[5] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal*, 15(2):121–142, 2006.

[6] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent Dirichlet Allocation. *J. Mach. Learn. Res.*, 3:993–1022, 2003.

[7] Kalina Bontcheva and Dominic Rout. Making Sense of Social Media Streams through Semantics: A Survey. *Semantic Web*, 5(5):373–403, 2014.

[8] Mário Cordeiro. Twitter Event Detection: Combining Wavelet Analysis and Topic Inference Summarization. In *Proc. Doctoral Symposium on Informatics Engineering (DSIE)*, 2012.

[9] Aron Culotta. Towards Detecting Influenza Epidemics by Analyzing Twitter Messages. In *Proc. Workshop on Social Media Analytics (SOMA)*, pages 115–122, 2010.

[10] Atefeh Farzindar and Wael Khreich. A Survey of Techniques for Event Detection in Twitter. *Computational Intelligence*, 31(1):132–164, 2015.

[11] Jin Li, Kristin Tufte, Vladislav Shkapenyuk, Vassilis Papadimos, Theodore Johnson, and David Maier. Out-of-Order Processing: A New Architecture for High-Performance Stream Systems. *PVLDB*, 1(1):274–288, 2008.

[12] Amina Madani, Omar Boussaid, and Djamel Eddine Zegour. What's Happening: A Survey of Tweets Event Detection. In *Proc. Intl. Conf. on Communications, Computation, Networks and Technologies (INNOV)*, pages 16–22, 2014.

[13] David Maier, Michael Grossniklaus, Sharmadha Moorthy, and Kristin Tufte. Capturing Episodes: May the Frame Be with You (Invited Paper). In *Proc. Intl. Conf. on Distributed Event-Based Systems (DEBS)*, pages 1–11, 2012.

[14] Arif Nurwidyantoro and Edi Winarko. Event Detection in Social Media: A Survey. In *Proc. Intl. Conf. on ICT for Smart Society (ICISS)*, pages 1–5, 2013.

[15] Takeshi Sakaki, Makoto Okazaki, and Yutaka Matsuo. Earthquake Shakes Twitter Users: Real-time Event Detection by Social Sensors. In *Proc. Intl. Conf. on World Wide Web (WWW)*, pages 851–860, 2010.

[16] Nesime Tatbul, Ugur Çetintemel, Stanley B. Zdonik, Mitch Cherniack, and Michael Stonebraker. Load Shedding in a Data Stream Manager. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, pages 309–320, 2003.

[17] Andreas Weiler, Michael Grossniklaus, and Marc H. Scholl. Event Identification and Tracking in Social Media Streaming Data. In *Proc. EDBT Workshop on Multimodal Social Data Management (MSDM)*, pages 282–287, 2014.

[18] Andreas Weiler, Michael Grossniklaus, and Marc H. Scholl. Evaluation Measures for Event Detection Techniques on Twitter Data Streams. In *Proc. British Intl. Conf. on Databases (BICOD)*, pages 108–119, 2015.

[19] Andreas Weiler, Michael Grossniklaus, and Marc H. Scholl. Run-time and Task-based Performance of Event De-

tection Techniques for Twitter. In *Proc. Intl. Conf. on Advanced Information Systems Engineering (CAiSE)*, pages 35–49, 2015.

[20] Andreas Weiler, Marc H. Scholl, Franz Wanner, and Christian Rohrdantz. Event Identification for Local Areas Using Social Media Streaming Data. In *Proc. Workshop on Databases and Social Networks (DBSocial)*, pages 1–6, 2013.

[21] Jianshu Weng and Bu-Sung Lee. Event Detection in Twitter. In *Proc. Intl. Conf on Weblogs and Social Media (ICWSM)*, pages 401–408, 2011.

[22] Wei Yang and Igor G. Zurbenko. Kolmogorov-Zurbenko Filters. *Wiley Interdisciplinary Reviews: Computational Statistics*, 2(3):340–351, 2010.

Data Engineering

It's FREE to join!

TCDE

tab.computer.org/tcde/

The Technical Committee on Data Engineering (TCDE) of the IEEE Computer Society is concerned with the role of data in the design, development, management and utilization of information systems.

- Data Management Systems and Modern Hardware/Software Platforms
- Data Models, Data Integration, Semantics and Data Quality
- Spatial, Temporal, Graph, Scientific, Statistical and Multimedia Databases
- Data Mining, Data Warehousing, and OLAP
- Big Data, Streams and Clouds
- Information Management, Distribution, Mobility, and the WWW
- Data Security, Privacy and Trust
- Performance, Experiments, and Analysis of Data Systems

The TCDE sponsors the International Conference on Data Engineering (ICDE). It publishes a quarterly newsletter, the Data Engineering Bulletin. If you are a member of the IEEE Computer Society, you may join the TCDE and receive copies of the Data Engineering Bulletin without cost. There are approximately 1000 members of the TCDE.

Join TCDE via Online or Fax

ONLINE: Follow the instructions on this page:

www.computer.org/portal/web/tandc/joinatc

FAX: Complete your details and fax this form to **+61-7-3365 3248**

Name _____

IEEE Member # _____

Mailing Address _____

_____

Country _____

Email _____

Phone _____

TCDE Mailing List

TCDE will occasionally email announcements, and other opportunities available for members. This mailing list will be used only for this purpose.

Membership Questions?

Xiaofang Zhou
School of Information Technology and Electrical Engineering
The University of Queensland
Brisbane, QLD 4072, Australia
zxf@uq.edu.au

TCDE Chair

Kyu-Young Whang
KAIST
371-1 Koo-Sung Dong, Yoo-Sung Ku
Daejeon 305-701, Korea
kywhang@cs.kaist.ac.kr

IEEE Computer Society
1730 Massachusetts Ave, NW
Washington, D.C. 20036-1903

Non-profit Org.
U.S. Postage
PAID
Silver Spring, MD
Permit 1398