Realtime Data Processing at Facebook

Guoqiang Jerry Chen, Janet L. Wiener, Shridhar Iyer, Anshul Jaiswal, Ran Lei
Nikhil Simha, Wei Wang, Kevin Wilfong, Tim Williamson, and Serhat Yilmaz
Facebook, Inc.

ABSTRACT

Realtime data processing powers many use cases at Facebook, including realtime reporting of the aggregated, anonymized voice of Facebook users, analytics for mobile applications, and insights for Facebook page administrators. Many companies have developed their own systems; we have a realtime data processing ecosystem at Facebook that handles hundreds of Gigabytes per second across hundreds of data pipelines.

Many decisions must be made while designing a realtime stream processing system. In this paper, we identify five important design decisions that affect their ease of use, performance, fault tolerance, scalability, and correctness. We compare the alternative choices for each decision and contrast what we built at Facebook to other published systems.

Our main decision was targeting seconds of latency, not milliseconds. Seconds is fast enough for all of the use cases we support and it allows us to use a persistent message bus for data transport. This data transport mechanism then paved the way for fault tolerance, scalability, and multiple options for correctness in our stream processing systems Puma, Swift, and Stylus.

We then illustrate how our decisions and systems satisfy our requirements for multiple use cases at Facebook. Finally, we reflect on the lessons we learned as we built and operated these systems.

1. INTRODUCTION

Realtime data processing systems are used widely to provide insights about events as they happen. Many companies have developed their own systems: examples include Twitter's Storm [28] and Heron [20], Google's Millwheel [9], and LinkedIn's Samza [4]. We present Facebook's Puma, Swift, and Stylus stream processing systems here.

The following qualities are all important in the design of a realtime data system.

- Ease of use: How complex are the processing requirements? Is SQL enough? Or is a general-purpose procedural language (such as C++ or Java) essential? How fast can a user write, test, and deploy a new application?
- Performance: How much latency is ok? Milliseconds? Seconds? Or minutes? How much throughput is required, per machine and in aggregate?
- Fault-tolerance: what kinds of failures are tolerated? What semantics are guaranteed for the number of times that data is processed or output? How does the system store and recover in-memory state?
- Scalability: Can data be sharded and resharded to process partitions of it in parallel? How easily can the system adapt to changes in volume, both up and down? Can it reprocess weeks worth of old data?
- Correctness: Are ACID guarantees required? Must all data that is sent to an entry point be processed and appear in results at the exit point?

In this paper, we present five decisions that must be made while designing a realtime stream processing system. We compare alternatives and their tradeoffs, identify the choices made by different systems in the literature, and discuss what we chose at Facebook for our stream processing systems and why.

Our main decision was that a few seconds of latency (with hundreds of Gigabytes per second throughput) meets our performance requirements. We can therefore connect all of the processing components in our system with a persistent message bus for data transport. Decoupling the data transport from the processing allowed us to achieve fault tolerance, scalability, and ease of use, as well as multiple options for correctness.

We run hundreds of realtime data pipelines in production. Four current production data pipelines illustrate how streaming systems are used at Facebook.

- Chorus is a data pipeline to construct the aggregated, anonymized voice of the people on Facebook: What are the top 5 topics being discussed for the election today? What are the demographic breakdowns (age, gender, country) of World Cup fans?
- Mobile analytics pipelines provide realtime feedback for Facebook mobile application developers. They use this data to diagnose performance and correctness issues, such as the cold start time and crash rate.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD 2016 San Francisco, CA USA
Copyright 2016 ACM 978-1-4503-3531-7/16/06 ...$15.00.
http://dx.doi.org/10.1145/2882903.2904441.

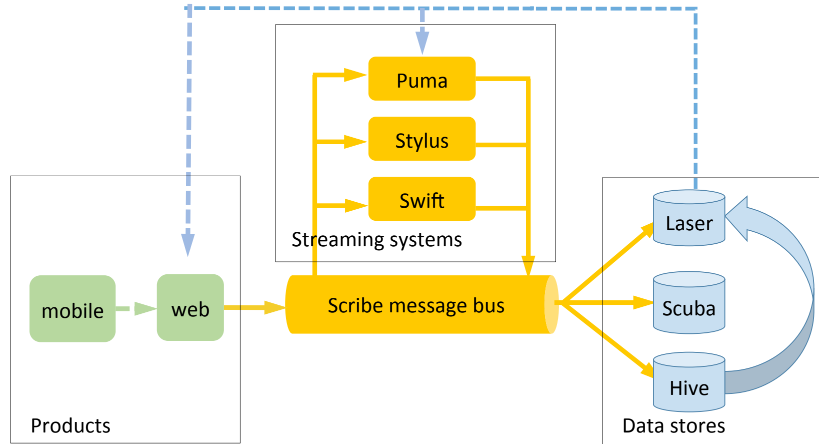

Figure 1: An overview of the systems involved in realtime data processing: from logging in mobile and web products on the left, through Scribe and realtime stream processors in the middle, to data stores for analysis on the right.

- Page insights pipelines provide Facebook Page owners realtime information about the likes, reach and engagement for each page post.
- Realtime streaming pipelines offload CPU-intensive dashboard queries from our interactive data stores and save global CPU resources.

We present several data pipelines in more detail after we describe our systems.

We then reflect on lessons we learned over the last four years as we built and rebuilt these systems. One lesson is to place emphasis on ease of use: not just on the ease of writing applications, but also on the ease of testing, debugging, deploying, and finally monitoring hundreds of applications in production.

This paper is structured as follows. In Section 2, we provide an overview of the realtime processing systems at Facebook and show how they fit together. In Section 3, we present an example application to compute trending events, which we use to illustrate the design choices in the rest of the paper. We discuss the design decisions in Section 4 and show how these decisions shaped realtime processing systems both at Facebook and in related work. We present several different streaming applications in production use at Facebook in Section 5. Then in Section 6, we reflect on lessons we learned about building and deploying realtime systems at Facebook. Finally, we conclude in Section 7.

2. SYSTEMS OVERVIEW

There are multiple systems involved in realtime data processing at Facebook. We present an overview of our ecosystem in this section.

Figure 1 illustrates the flow of data through our systems. On the left, data originates in mobile and web products. The data they log is fed into Scribe, which is a distributed data transport system. All of the solid (yellow) arrows represent data flowing through Scribe.

The realtime stream processing systems Puma, Stylus, and Swift read data from Scribe and also write to Scribe. Puma, Stylus, and Swift applications can be connected through Scribe into a complex DAG (directed acyclic graph), as needed. We overview them here and describe their differences in detail in Section 4.

On the right, Laser, Scuba, and Hive are data stores that use Scribe for ingestion and serve different types of queries. Laser can also provide data to the products and streaming systems, as shown by the dashed (blue) arrows. In this section, we describe each of the data systems.

2.1 Scribe

Scribe [5] is a persistent, distributed messaging system for collecting, aggregating and delivering high volumes of log data with a few seconds of latency and high throughput. Scribe is the transport mechanism for sending data to both batch and realtime systems at Facebook. Within Scribe, data is organized by *category*. A category is a distinct stream of data: all data is written to or read from a specific category. Usually, a streaming application consumes one Scribe category as input. A Scribe category has multiple buckets. A Scribe bucket is the basic processing unit for stream processing systems: applications are parallelized by sending different Scribe buckets to different processes. Scribe provides data durability by storing it in HDFS[23]. Scribe messages are stored and streams can be replayed by the same or different receivers for up to a few days.

2.2 Puma

Puma is a stream processing system whose applications (apps) are written in a SQL-like language with UDFs (user-defined functions) written in Java. Puma apps are quick to write: it can take less than an hour to write, test, and deploy a new app.

Puma apps serve two purposes. First, Puma provides pre-computed query results for simple aggregation queries. For these stateful monoid applications (see section 4.4.2), the delay equals the size of the query result's time window. The query results are obtained by querying the Puma app

```
CREATE APPLICATION top_events;

CREATE INPUT TABLE events_score(
  event_time,
  event,
  category,
  score
)
FROM SCRIBE("events_stream")
TIME event_time;

CREATE TABLE top_events_5min AS
  SELECT
      category,
      event,
      topk(score) AS score
  FROM
      events_score [5 minutes]
```

Figure 2: A complete Puma app that computes the "top K events" for each 5 minute time window. This app can be used for the *Ranker* in Fig 3.

through a Thrift API [8]. Figure 2 shows code for a simple Puma aggregation app with 5 minute time windows.

Second, Puma provides filtering and processing of Scribe streams (with a few seconds delay). For example, a Puma application can reduce a stream of all Facebook actions to only posts, or to only posts that match a predicate, such as containing the hashtag "#superbowl". The output of these stateless Puma apps is another Scribe stream, which can then be the input to another Puma app, any other realtime stream processor, or a data store.

Unlike traditional relational databases, Puma is optimized for compiled queries, not for ad-hoc analysis. Engineers deploy apps with the expectation that they will run for months or years. This expectation allows Puma to generate an efficient query computation and storage plan. Puma aggregation apps store state in a shared HBase cluster.

2.3 Swift

Swift is a basic stream processing engine which provides checkpointing functionalities for Scribe. It provides a very simple API: you can read from a Scribe stream with checkpoints every N strings or B bytes. If the app crashes, you can restart from the latest checkpoint; all data is thus read at least once from Scribe. Swift communicates with client apps through system-level pipes. Thus, the performance and fault tolerance of the system are up to the client. Swift is mostly useful for low throughput, stateless processing. Most Swift client apps are written in scripting languages like Python.

2.4 Stylus

Stylus is a low-level stream processing framework written in C++. The basic component of Stylus is a stream processor. The input to the processor is a Scribe stream and the output can be another Scribe stream or a data store for serving the data. A Stylus processor can be stateless or stateful. Processors can be combined into a complex processing DAG. We present such an example DAG in Figure 3 in the next section.

Stylus's processing API is similar to that of other procedural stream processing systems[4, 9, 28]. Like them, Stylus must handle imperfect ordering in its input streams [24, 10, 9]. Stylus therefore requires the application writer to identify the event time data in the stream. In return, Stylus provides a function to estimate the event time low watermark with a given confidence interval.

2.5 Laser

Laser is a high query throughput, low (millisecond) latency, key-value storage service built on top of RocksDB [3]. Laser can read from any Scribe category in realtime or from any Hive table once a day. The key and value can each be any combination of columns in the (serialized) input stream. Data stored in Laser is then accessible to Facebook product code and to Puma and Stylus apps.

Laser has two common use cases. Laser can make the output Scribe stream of a Puma or Stylus app available to Facebook products. Laser can also make the result of a complex Hive query or a Scribe stream available to a Puma or Stylus app, usually for a lookup join, such as identifying the topic for a given hashtag.

2.6 Scuba

Scuba [7, 18, 15] is Facebook's fast slice-and-dice analysis data store, most commonly used for trouble-shooting of problems as they happen. Scuba ingests millions of new rows per second into thousands of tables. Data typically flows from products through Scribe and into Scuba with less than one minute of delay. Scuba can also ingest the output of any Puma, Stylus, or Swift app.

Scuba provides ad hoc queries with most response times under 1 second. The Scuba UI displays query results in a variety of visualization formats, including tables, time series, bar charts, and world maps.

2.7 Hive data warehouse

Hive [26] is Facebook's exabyte-scale data warehouse. Facebook generates multiple new petabytes of data per day, about half of which is raw event data ingested from Scribe[15]. (The other half of the data is derived from the raw data, e.g., by daily query pipelines.) Most event tables in Hive are partitioned by day: each partition becomes available after the day ends at midnight. Any realtime processing of this data must happen in Puma, Stylus, or Swift applications. Presto [2] provides full ANSI SQL queries over data stored in Hive. Query results change only once a day, after new data is loaded. They can then be sent to Laser for access by products and realtime stream processors.

3. EXAMPLE APPLICATION

We use the example application in Figure 3 to demonstrate the design choices in the next section. This application identifies trending events in an input stream of events. The events contain an event type, a dimension id (which is used to fetch dimension information about the event, such as the language in which it is written), and text (which is analyzed to classify the event topic, such as *movies* or *babies*). The output of the application is a ranked list of topics (sorted by event count) for each 5 minute time bucket.

There are four processing nodes, each of which may be executed by multiple processes running in parallel on disjoint partitions of their input.

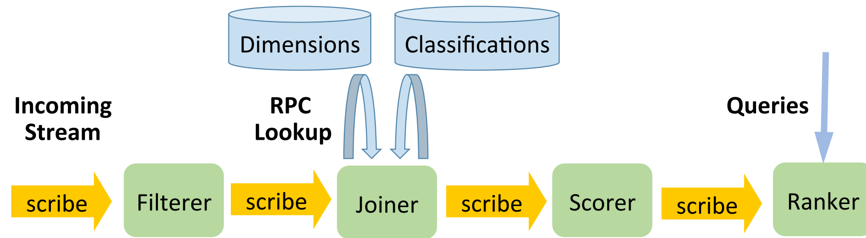

Figure 3: An example streaming application with 4 nodes: this application computes "trending" events.

1. The *Filterer* filters the input stream based on the event type and then shards its output on the dimension id so that the processing for the next node can be done in parallel on shards with disjoint sets of dimension ids.

2. The *Joiner* queries one or more external systems to (a) retrieve information based on the dimension id and (b) classify the event by topic, based on its text. Since each *Joiner* process receives sharded input, it is more likely to have the dimension information it needs in a cache, which reduces network calls to the external service. The output is then resharded by $(event, topic)$ pair so that the *Scorer* can aggregate them in parallel.

3. The *Scorer* keeps a sliding window of the event counts per topic for recent history. It also keeps track of the long term trends for these counters. Based on the long term trend and the current counts, it computes a score for each (event, topic) pair and emits the score as its output to the *Ranker*, resharded by topic.

4. The *Ranker* computes the top K events for each topic per N minute time bucket.

In this example, the *Filterer* and *Joiner* are stateless and the *Scorer* and *Ranker* are stateful. At Facebook, each of the nodes can be implemented in Stylus. Puma apps can implement the *Filterer* and *Ranker*. The example Puma app in Figure 2 contains code for the *Ranker*. Although a Puma app can join with data in Laser, the *Joiner* node may need to query an arbitrary service for the Classifications, which Puma cannot do. Swift can only be used to implement the stateless nodes.

A consumer service queries the *Ranker* periodically to get the top K events for each topic. Alternatively, the *Ranker* can publish its results to Laser and the consumer service can query Laser. Puma is designed to handle thousands of queries per second per app, whereas Laser is designed to handle millions. Querying Laser is also a better choice when the query latency requirements are in milliseconds.

4. DESIGN DECISIONS

In this section, we present five design decisions. These decisions are summarized in Table 4, where we show which capabilities each decision affects. For each one, we categorize the alternatives and explain how they affect the relevant capabilities. Then we discuss the pros and cons of our decision for Facebook's systems.

Table 5 summarizes which alternatives were chosen by a variety of realtime systems, both at Facebook and in the related literature.

4.1 Language paradigm

The first design decision is the type of language that people will use to write applications in the system. This decision determines how easy it is to write applications and how much control the application writer has over their performance.

4.1.1 Choices

There are three common choices:

- *Declarative*: SQL is (mostly) declarative. SQL is the simplest and fastest to write. A lot of people already know SQL, so their ramp-up is fast. However, the downside of SQL is its limited expressiveness. Many systems add functions to SQL for operations such as hashing and string operators. For example, Streambase [27], S-Store [21] and STREAM [12] provide SQL-based stream processing.

- *Functional*: Functional programming models [10, 30, 32] represent an application as a sequence of predefined operators. It is still simple to write an application, but the user has more control over the order of operations and there are usually more operations available.

- *Procedural*: C++, Java, and Python are all common procedural languages. They offer the most flexibility and (usually) the highest performance. The application writer has complete control over the data structures and execution. However, they also take the most time to write and test and require the most language expertise. S4 [22], Storm [28], Heron [20], and Samza [4] processors are all examples of procedural stream processing systems.

4.1.2 Languages at Facebook

In our environment at Facebook, there is no single language that fits all use cases. Needing different languages (and the different levels of ease of use and performance they provide) is the main reason why we have three different stream processing systems.

Puma applications are written in SQL. A Puma app can be written and tested in under an hour, which makes it very ease to use. Puma apps have good throughput and can increase their throughput by using more parallel processing nodes.

Swift applications mostly use Python. It is easy to prototype and it is very useful for low-throughput (tens of Megabytes per second) stream processing apps. Although it is possible to write a high performance processor with Swift, it takes a lot of effort.

Design decision	Ease of use	Performance	Fault tolerance	Scalability	Correctness
Language paradigm	X	X			
Data transfer	X	X	X	X	
Processing semantics			X		X
State-saving mechanism	X	X	X	X	X
Reprocessing	X			X	X

Figure 4: Each design decision affects some of the data quality attributes.

Design decision	Puma	Stylus	Swift	Storm	Heron	Spark Streaming	Millwheel	Flink	Samza
Language paradigm	SQL	C++	Python	Java	Java	Functional	C++	Functional	Java
Data transfer	Scribe	Scribe	Scribe	RPC	Stream Manager	RPC	RPC	RPC	Kafka
Processing semantics	at least	at least at most exactly	at least at most	at least	at least	best effort exactly	at least exactly	at least exactly	at least
State-saving mechanism	remote DB	local DB remote DB				limited	remote DB	global snapshot	local DB
Reprocessing	same code	same code	no batch	same DSL	same DSL	same code	same code	same code	no batch

Figure 5: The design decisions made by different streaming systems.

Stylus applications are written in C++ and a Stylus processor requires multiple classes. While a script will generate boilerplate code, it can still take a few days to write an application. Stylus applications have the greatest flexibility for complicated stream processing applications.

We do not currently provide any functional paradigms at Facebook, although we are exploring Spark Streaming [32].

4.2 Data transfer

A typical stream processing application is composed of multiple processing nodes, arranged in a DAG. The second design decision is the mechanism to transfer data between processing nodes. This decision has a significant impact on the fault tolerance, performance, and scalability of the stream processing system. It also affects its ease of use, particularly for debugging.

4.2.1 Choices

Typical choices for data transfer include:

- Direct message transfer: Typically, an RPC or in-memory message queue is used to pass data directly from one process to another. For example, MillWheel [9], Flink [16], and Spark Streaming [32] use RPC and Storm [28] uses ZeroMQ [6], a form of message queue. One of the advantages of this method is speed: tens of milliseconds end-to-end latency is achievable.

- Broker based message transfer: In this case, there is a separate broker process that connects stream processing nodes and forwards messages between them. Using an intermediary process adds overhead, but also allows the system to scale better. The broker can multiplex a given input stream to multiple output processors. It can also apply back pressure to an input processor when the output processor falls behind. Heron [20] uses a stream manager between Heron instances to solve both of these problems.

- Persistent storage based message transfer. In this case, processors are connected by a persistent message bus. The output stream of one processor is written to a persistent store and the next processor reads its input from that store. This method is the most reliable. In addition to multiplexing, a persistent store allows the input and output processors to write and read at different speeds, at different points in time, and to read the same data multiple times, e.g., to recover from a processor failure. The processing nodes are decoupled from each other so the failure of a single node does not affect other nodes. Samza [4] uses Kafka [19], a persistent store, to connect processing nodes.

All three types of data transfer mechanisms connect consecutive nodes in a DAG.

There are two types of connections between consecutive nodes [31]. Narrow dependency connections link a fixed number of partitions from the sending node to the receiving node. Such connections are often one-to-one and their nodes can be collapsed. Wide dependency connections link every partition of the sender to each partition of the receiver. These connections must be implemented with a data transfer mechanism.

4.2.2 Data transfer at Facebook

At Facebook, we use Scribe [5], a persistent message bus, to connect processing nodes. Using Scribe imposes a minimum latency of about a second per stream. However, at Facebook, the typical requirement for real time stream processing is seconds.

A second limitation of Scribe is that it writes to disk. In practice, the writes are asynchronous (not blocking) and the reads come from a cache because streaming applications read the most recent data. Finally, a persistent store requires additional hardware and network bandwidth.

Accepting these limitations gives us multiple advantages for fault tolerance, ease of use, scalability, and performance.

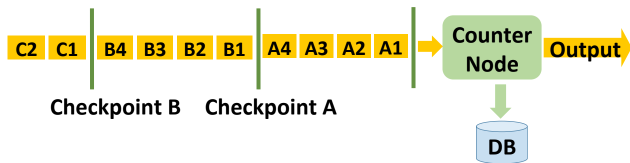

Figure 6: This Counter Node processor counts events from a $(timestamp, event)$ input stream. Every few seconds, it emits the counter value to a $(timewindow, counter)$ output stream.

- Fault tolerance: The independence of stream processing node failures is a highly desirable property when we deploy thousands of jobs to process streams.
- Fault tolerance: Recovery from failure is faster because we only have to replace the node(s) that failed.
- Fault tolerance: Automatic multiplexing allows us to run duplicate downstream nodes. For example, we can run multiple Scuba or Laser tiers that *each* read all of their input streams' data, so that we have redundancy for disaster recovery purposes.
- Performance: If one processing node is slow (or dies), the speed of the previous node is not affected. For example, if a machine is overloaded with too many jobs, we simply move some jobs to a new machine and they pick up processing the input stream from where they left off. In a tightly coupled system [9, 16, 32, 28, 20], back pressure is propagated upstream and the peak processing throughput is determined by the slowest node in the DAG.
- Ease of use: Debugging is easier. When a problem is observed with a particular processing node, we can reproduce the problem by reading the same input stream from a new node.
- Ease of use: Monitoring and alerting are simpler to implement. The primary responsibility of each node is to consume its input. It is sufficient to set up monitoring and alerts for delays in processing streams from the persistent store.
- Ease of use: We have more flexibility in how we write each application. We can connect components of any system that reads or writes data in the same DAG. We can use the output of a Puma application as the input of a Stylus processor and then read the Stylus output as input to our data stores Scuba or Hive.
- Scalability: We can scale the number of partitions up or down easily by changing the number of buckets per Scribe category in a configuration file.

Given the advantages above, Scribe has worked well as the data transfer mechanism at Facebook. Kafka or another persistent store would have similar advantages. We use Scribe because we develop it at Facebook.

4.3 Processing semantics

The processing semantics of each node determine its correctness and fault tolerance.

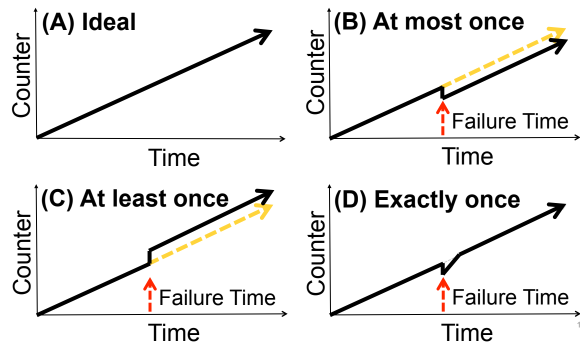

Figure 7: The output of a stateful processor with different state semantics.

4.3.1 Choices

A stream processor does three types of activities.

1. Process input events: For example, it may deserialize input events, query an external system, and update its in-memory state. These activities can be rerun without side effects.
2. Generate output: Based on the input events and in-memory state, it generates output for downstream systems for further processing or serving. This can happen as input events are processed or can be synchronized before or after a checkpoint.
3. Save checkpoints to a database for failure recovery. Three separate items may be saved:
   (a) The in-memory state of the processing node.
   (b) The current offset in the input stream.
   (c) The output value(s).

   Not all processors will save all of these items. What is saved and when determines the processor's semantics.

The implementations of these activities, especially the checkpoints, control the processor's semantics. There are two kinds of relevant semantics:

- State semantics: can each input event count at-least-once, at-most-once, or exactly-once?
- Output semantics: can a given output value show up in the output stream at-least-once, at-most-once, or exactly-once?

Stateless processors only have output semantics. Stateful processors have both kinds.

The different state semantics depend only on the order of saving the offset and in-memory state.

- At-least-once state semantics: Save the in-memory state first, then save the offset.
- At-most-once state semantics: Save the offset first, then save the in-memory state.
- Exactly-once state semantics: Save the in-memory state and the offset atomically, e.g., in a transaction.

	State semantics		
Output semantics	At-least-once	At-most-once	Exactly-once
At-least-once	X		X
At-most-once		X	X
Exactly-once			X

Figure 8: Common combinations of state and output processing semantics.

Output semantics depend on saving the output value(s) in the checkpoint, in addition to the in-memory state and offset.

- At-least-once output semantics: Emit output to the output stream, then save a checkpoint of the offset and in-memory state.
- At-most-once output semantics: Save a checkpoint of the offset and in-memory state, then emit output.
- Exactly-once output semantics: Save a checkpoint of the offset and in-memory state and emit output value(s) atomically in the same transaction.

Figure 6 shows a stateful stream processing node, the "Counter Node", which counts events in its input and outputs the count periodically. We use the Counter Node to illustrate the different state processing semantics with at-least-once output semantics. Figure 7 shows how different semantics affect the possible counter output values after a machine or processor failure.

At-least-once output semantics allow the Counter Node to emit output as it receives events. At-least-once output semantics are desirable for systems that require low latency processing and can handle small amounts of input data duplication.

To achieve at-most-once output semantics, the Counter Node must save its checkpoint before generating output. If the processor can do side-effect-free processing of events $A1 - A4$, then save checkpoint A, and then generate output, it does not need to buffer events $A1-A4$ before saving checkpoint A. This optimization also reduces the chance of losing data, since only failures that happen between checkpointing and emitting output can cause data loss. We illustrate the performance benefits of doing side-effect-free processing between checkpoints in Section 4.3.2. Photon [11] also offers options to reduce data loss with at-most-once output semantics. At-most-once state and output semantics are desirable when data loss is preferable to data duplication.

To get exactly-once output semantics, the Counter Node must save checkpoints after processing events $A1 - A4$, but atomically with emitting the output. Exactly-once output semantics require transaction support from the receiver of the output. In practice, this means that the receiver must be a data store, rather than a data transport mechanism like Scribe. Exactly-once semantics usually impose a performance penalty, since the processor needs to wait for transactions to complete.

Table 8 shows the common combinations of state and output semantics.

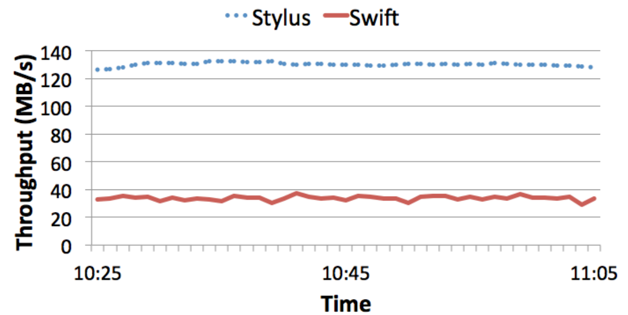

Figure 9: The Stylus implementation of this processor does side-effect-free processing between checkpoints and achieves nearly 4x as much throughput as the Swift implementation.

4.3.2 *Processing semantics used at Facebook*

In Facebook's environment, different applications often have different state and output semantics requirements. We give a few different examples.

In the trending events example in Figure 3, the *Ranker* sends its results to an idempotent serving system. Sending output twice is not a problem. Therefore, we can use at-least-once state and output semantics.

The data ingestion pipeline for Scuba [7] is stateless. Only the output semantics apply. Most data sent to Scuba is sampled and Scuba is a best-effort query system, meaning that query results may be based on partial data. Therefore, a small amount of data loss is preferred to any data duplication. Exactly-once semantics are not possible because Scuba does not support transactions, so at-most-once output semantics are the best choice.

In fact, most of our analysis data stores, including Laser, Scuba, and Hive, do not support transactions. We need to use other data stores to get transactions and exactly-once state semantics.

Getting exactly-once state semantics is also a challenge when the downstream data store is a distributed database such as HBase or ZippyDB. (ZippyDB is Facebook's distributed key-value store with Paxos-style replication, built on top of RocksDB.) The state must be saved to multiple shards, requiring a high-latency distributed transaction. Instead of incurring this latency, most users choose at-most-once or at-least-once semantics.

Puma guarantees at-least-once state and output semantics with checkpoints to HBase. Stylus offers all of the options in Figure 8 to its application writers.

We now examine the benefits of overlapping side-effect-free processing with receiving input for at-most-once output semantics. Figure 9 shows the throughput for two different implementations of the Scuba data ingestion processor. Both implementations have at-most-once output semantics. The Swift implementation buffers all input events between checkpoints, which occur approximately every 2 seconds. Then it processes those events and sends its output to the Scuba servers. While it is waiting for the checkpoint, the processor's CPU is underutilized.

The Stylus implementation does as much side-effect-free work as it can between checkpoints, including deserialization of its input events. Since deserialization is the performance

bottleneck, the Stylus processor achieves much higher CPU utilization. The Stylus processor, therefore, achieves nearly four times the throughput of the Swift processor: in Figure 9 we see 135 MB/s versus 35 MB/s, respectively.

In general, separating out the side-effect-free processing and doing it between checkpoints can be implemented in custom code in any processor, if the developer has a good understanding of processing semantics. Stylus provides this optimization out of the box.

4.4 State-saving mechanisms

The state saving mechanism for stateful processors affects their fault tolerance. For example, in Figure 3, the *Scorer* maintains both long term counters and short term counters for events in order to compute the trending score. After a machine failure, we need to restore the counter values.

4.4.1 Choices

There are multiple ways to save state during processing and restore it after a failure. These solutions include:

- Replication [13]. In a replication based approach, the stateful nodes are replicated with two or more copies. This approach requires twice as much hardware, since many nodes are duplicated.

- Local database persistence. Samza [4] stores the state to a local database and writes the mutation to Kafka at the same time. After a failure, the local state is recovered from the Kafka log. The logs are compacted offline to keep log size bounded. Samza can support at-least-once but not exactly-once state and output semantics, since Kafka does not support transactions. .

- Remote database persistence. In this approach, the checkpoint and states are persisted to a remote database. MillWheel [9] saves fine-grained checkpoints to a remote database and supports exactly-once state and output semantics.

- Upstream backup. In these systems, the events are buffered in the upstream nodes and replayed after a failure [17].

- Global consistent snapshot. Flink [16] uses a distributed snapshot algorithm to maintain a globally consistent snapshot. After a failure, multiple machines must be restored to the consistent state.

4.4.2 State saving mechanisms at Facebook

At Facebook, we have different demands for fault tolerance for stream processing systems. Puma provides fault tolerance for stateful aggregation. Stylus provides multiple out-of-the-box fault tolerant solutions for stateful processing.

We implemented two state-saving mechanisms in Stylus: a local database model and a remote database model. In this section, we will focus only on our general local database implementation and our implementation of a remote database model for monoid processors, defined below.

Saving state to a local RocksDB [3] database is attractive for many users at Facebook. It is easy to set up. The local database writes are fast when a flash or memory-based file system (such as tmpfs) is used, since there is no network cost.

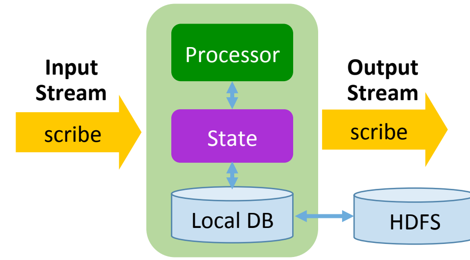

Figure 10: Saving state using a RocksDB local database and HDFS for remote backup.

It also supports exactly-once state and output semantics for stateful processing.

The *Scorer* node in Figure 3 is a good candidate for saving its state to a local database: the overall state is small and will fit into the flash or disk of a single machine.

Figure 10 illustrates RocksDB embedded in the same process as the stream processor and its state. The in-memory state is saved to this local database at fixed intervals. The local database is then copied asynchronously to HDFS at a larger interval using RocksDB's backup engine. When a process crashes and restarts on the same machine, the local database is used to restore the state and resume processing from the checkpoint. If the machine dies, the copy on HDFS is used instead.

HDFS is designed for batch workloads and is not intended to be an always-available system. If HDFS is not available for writes, processing continues without remote backup copies. If there is a failure, then recovery uses an older snapshot. We can parallelize reading from HDFS at recovery time so that HDFS read bandwidth is not a bottleneck.

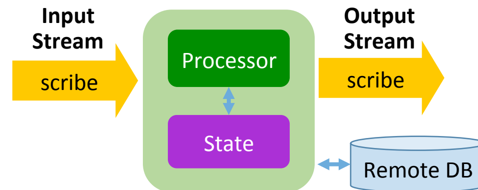

Figure 11: Saving state using a remote database.

A remote database can hold states that do not fit in memory. A remote database solution also provides faster machine failover time since we do not need to load the complete state to the machine upon restart. Often, a distributed database serves as the store for the output of the stream processing system. The same remote database can be used for saving state.

Figure 11 illustrates saving state in a remote database. When an event is received from the input stream, the state is updated. If the needed state is not in memory, it is read from the remote database, modified, and then saved back to the database. This read-modify-write pattern can be optimized if the states are limited to monoid processors.

A monoid [1, 14] is an algebraic structure that has an identity element and is associative. When a monoid processor's application needs to access state that is not in memory, mutations are applied to an empty state (the identity element).

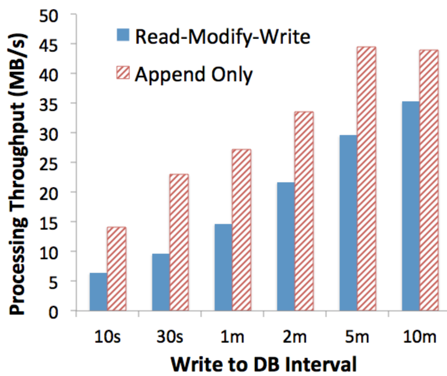

Figure 12: Saving state: read-modify-write vs append-only remote database write throughput.

Periodically, the existing database state is loaded in memory, merged with the in-memory partial state, and then written out to the database asynchronously. This read-merge-write pattern can be done less often than the read-modify-write.

When the remote database supports a custom merge operator, then the merge operation can happen in the database. The read-modify-write pattern is optimized to an append-only pattern, resulting in performance gains.

RocksDB and ZippyDB support custom merge operators. We illustrate the performance gains in Figure 12, which compares the throughput of the same Stylus stream processor application, configured with and without the append-only optimization. The application aggregates its input events across many dimensions, which means that one input event changes many different values in the application state. The remote database is a three machine ZippyDB cluster. Since different applications may have different requirements for how often to save their state remotely, we varied the interval for flushing to the remote database. Figure 12 shows that the application throughput is 25% to 200% higher with the append-only optimization.

The aggregation functions in Puma are all monoid. In Stylus, a user application declares that it is a monoid stream processor. The application appends partial state to the framework and Stylus decides when to merge the partial states into a complete state. This flexibility in when to merge — and the resulting performance gain — is the main difference between our remote database approach and that of Millwheel.

4.5 Backfill processing

We often need to reprocess old data, for several reasons.

- When a user develops a new stream processing application, it is good to test the application against old data. For example, in Figure 3, it is useful to run the trending algorithm on a known events stream and see if it identifies the expected trending events.
- For many applications, when we add a new metric, we want to run it against old data to generate historical metric data.
- When we discover a logging or processing bug, we need to reprocess the data for the period with the bug.

4.5.1 Choices

There are a few approaches to reprocessing data.

- Stream only. In this approach, the retention of the data transport mechanism must be long enough to replay the input streams for reprocessing.
- Maintain two separate systems: one for batch, and one for stream processing. For example, to bootstrap the long term counters in Figure 3, we could develop a separate batch pipeline to compute them. This approach is challenging: it is hard to maintain consistency between two different systems. Summingbird [14] uses a high level DSL to convert one processor specification to each system automatically, but still needs to maintain consistency between two different processing system implementations.
- Develop stream processing systems that can also run in a batch environment. Spark Streaming and Flink are good examples of this approach, which is also the one we take.

4.5.2 Reprocessing data at Facebook

Scribe does not provide infinite retention; instead we store input and output streams in our data warehouse Hive for longer retention. Our approach is different from Spark Streaming and Flink. They use similar fault tolerance and data transfer mechanisms for both their batch and stream processing.

To reprocess older data, we use the standard MapReduce framework to read from Hive and run the stream processing applications in our batch environment. Puma applications can run in Hive's environment as Hive UDFs (user-defined functions) and UDAFs (user-defined aggregation functions). The Puma app code remains unchanged, whether it is running over streaming or batch data.

Stylus provides three types of processors: a stateless processor, a general stateful processor, and a monoid stream processor. When a user creates a Stylus application, two binaries are generated at the same time: one for stream and one for batch. The batch binary for a stateless processor runs in Hive as a custom mapper. The batch binary for a general stateful processor runs as a custom reducer and the reduce key is the aggregation key plus event timestamp. The batch binary for monoid processors can be optimized to do partial aggregation in the map phase.

5. FACEBOOK APPLICATIONS

In this section, we describe several different realtime applications in production use at Facebook. We show how these applications use the different components available to achieve their particular goals.

5.1 Chorus

The Chorus data pipeline [25, 29] transforms a stream of individual Facebook posts into aggregated, anonymized, and annotated visual summaries (that do not reveal any private information). This pipeline enables Facebook's communications and insights teams to report to the public about realtime conversations, as they happen, without needing to know about the underlying data stores or queries. New posts show up in query results in seconds: for example, during the 2015 Superbowl, we watched a huge spike in posts

containing the hashtag "#likeagirl" in the 2 minutes following the TV ad. http://insights.fb.com/2015/02/02/the-big-gathering-xlix/ contains an analysis of Superbowl 2015 conversation in the U.S.

There are many steps in this pipeline. We would like to call attention to two things.

First, this pipeline contains a mix of Puma and Stylus apps, with lookup joins in Laser and both Hive and Scuba as sink data stores for the results. All data transport is via Scribe. The data flow is similar to the example in Figure 3.

Second, this pipeline has evolved substantially over the last two years. The original pipeline had only one Puma app to filter posts. The laser joins were added with custom Python code to perform the join. Later still, a Stylus app replaced the custom code. At each step in the pipeline's evolution, we could add or replace one component at a time and test and deploy incrementally.

5.2 Dashboard queries

Dashboards are popular for observing trends and spotting anomalies at a glance. They run the same queries repeatedly, over a sliding time window. Once the query is embedded in a dashboard, the aggregations and metrics are fixed. Stream processing apps compute their query results as the data arrives. They are ideal for dashboard queries.

Scuba was designed for interactive, slice-and-dice queries. It does aggregation at query time by reading all of the raw event data. When we realized that queries from dashboards consumed much Scuba CPU, we built a framework to migrate dashboard queries from Scuba to Puma.

There were a few challenges. First, Puma apps could read the Scuba input from Scribe, but needed new code to compute Scuba UDFs that were defined in Scuba's visualization layer.

Second, Puma was designed for apps with millions of time series in their results. Those Puma apps shard their data by time series. Most Scuba queries have a limit of 7: it only makes sense to visualize up to 7 lines in a chart. They can not be sharded by time series into $N > 7$ processes: the processes must use a different sharding key and compute partial aggregates. One process then combines the partial aggregates.

Third, while exporting queries to dashboards, people experiment with different queries. We need to detect dead dashboard queries in order to avoid wasting CPU. Overall, the migration project has been very successful. The Puma apps consume approximate 14% of the CPU that was needed to run the same queries in Scuba.

In general, there is a tradeoff between processing at read time, such as Scuba does, and processing at write time, such as the stream processors Puma and Stylus do. Read time processing is much more flexible – you don't need to choose the queries in advance – but generally more CPU intensive.

5.3 Hybrid realtime-batch pipelines

Over half of all queries over Facebook's data warehouse Hive are part of daily query pipelines. The pipelines can start processing anytime after midnight. Due to dependencies, some of them complete only after 12 or more hours. We are now working on converting some of the earlier queries in these pipelines to realtime streaming apps so that the pipelines can complete earlier.

Interesting challenges in these conversions include:

- Validating that the realtime pipeline results are correct. Correct realtime results may not be identical to batch results. For example, the boundaries between days may differ.
- Adding enough common Hive UDFs to Puma and Stylus to support most queries.
- Making the conversion process self-service, so that teams can convert, test, and deploy without help from the Puma and Stylus teams.

In multiple cases, we have sped up pipelines by 10 to 24 hours. For example, we were able to convert a portion of a pipeline that used to complete around 2pm to a set of realtime stream processing apps that deliver the same data in Hive by 1am. The end result of this pipeline is therefore available 13 hours sooner.

6. LESSONS LEARNED

We have learned a lot while building our ecosystem of realtime data systems, as much from comparing them as from building any single system. Many of the lessons below are about service management: it is not enough to provide a framework for users to write applications. Ease of use encompasses debugging, deployment, and monitoring, as well. The value of tools that make operation easier is underestimated. In our experience, every time we add a new tool, we are surprised that we managed without it.

6.1 Multiple systems let us "move fast"

There is no single stream processing system that is right for all use cases. Providing a set of systems at different points in the ease of use versus performance, fault tolerance, and scalability space has worked well in Facebook's culture of "move fast" and "iterate". We have seen multiple applications start out in Puma or Swift and then move to Stylus (weeks or months later) as they needed more expressiveness or higher throughput. Writing a simple application lets our users deploy something quickly and prove its value first, then invest the time in building a more complex and robust application.

Similarly, evolving our own systems has been important. Every major piece of Puma has been rewritten at least once. Using a high-level language allows us to modify the underlying implementation without requiring changes to Puma apps.

Covering more points in the design space has also been beneficial. We can and do create stream processing DAGs that contain a mix of Puma, Swift, and Stylus applications, which gives us a lot of flexibility in satisfying the requirements for each node individually.

Finally, connecting nodes with Scribe streams makes it easy not just to replace a node with a faster or more expressive node, but also to reuse the output of that node in another data pipeline (DAG). We have been able to save developer time and system resources by reusing the same processor output for different purposes.

6.2 Ease of debugging

Databases typically store data and enable users to run queries on it. Such an environment is very conducive to

an iterative development process. A user can program one processing stage and run it. If it doesn't yield the right results, they can modify the processing logic and rerun it on the same data.

When working with stream processing systems, it is harder to iterate during development because the data is not stored. When you update a stream processing operator, it begins running on new streaming data, not the same data as before, so the results may be different. Moreover, you have to handle delayed data. With persistent Scribe streams, we can replay a stream from a given (recent) time period, which makes debugging much easier.

6.3 Ease of deployment

Writing a new streaming application requires more than writing the application code. The ease or hassle of deploying and maintaining the application is equally important.

Laser and Puma apps are deployed as a service. Stylus apps are owned by the individual teams who write them, but we provide a standard framework for monitoring them.

Laser apps are extremely easy to setup, deploy, and delete. There is a UI to configure the app: just choose an ordered set of columns from the input Scribe stream for each of the key and value, a lifetime for each key-value pair, and a set of data centers to run the service. The UI then gives you a single command to deploy an app and another command to delete it.

A Puma app is almost as easy to deploy and delete as a Laser app, but requires a second engineer: the UI generates a code diff that must be reviewed. The app is deployed or deleted automatically after the diff is accepted and committed.

Puma apps were not always this easy to deploy. Originally, a member of the Puma team had to deploy each new app, which created a bottleneck for both the Puma team and the teams that wrote the apps. Making Puma deployment self-service let us scale to the hundreds of data pipelines that use Puma today.

6.4 Ease of monitoring and operation

Once an app is deployed, we need to monitor it. Is it using the right amount of parallelism? With Scribe, changing the parallelism is often just changing the number of Scribe buckets and restarting the nodes that output and consume that Scribe category. However, guessing the right amount of parallelism before deployment is a black art. We save both time and machine resources by being able to change it easily; we can get started with some initial level and then adapt quickly.

We then use alerts to detect when an app is processing its Scribe input more slowly than the input is being generated. We call that "processing lag". The Puma team runs processing lag alerts for all Puma apps, because the Puma service is maintained by the Puma team. Stylus provides its application developers with processing lag alerts. These alerts notify us to adapt our apps to changes in volume over time.

In the future, we would like to provide dashboards and alerts that are automatically configured to monitor both Puma and Stylus apps for the teams that use them. We would also like to scale the apps automatically.

6.5 Streaming vs batch processing

Streaming versus batch processing is not an either/or decision. Originally, all data warehouse processing at Facebook was batch processing [26]. We began developing Puma and Swift about five years ago. As we showed in Section 5.3, using a mix of streaming and batch processing can speed up long pipelines by hours.

Furthermore, streaming-only systems can be authoritative. We do not need to treat realtime system results as an approximation and batch results as the "truth." It is possible to create streaming systems that do not miss data (so counts and sums are exact). Good approximate unique counts (computed with HyperLogLog) are often as actionable as exact numbers.

7. CONCLUSIONS

In the last few years, real time processing has proliferated at Facebook. We have developed multiple, independent yet composable systems. Together, they form a comprehensive platform to serve our diverse needs.

In this paper, we discuss multiple design decisions we made and their effects on ease of use, performance, fault tolerance, scalability, and correctness. We would like to conclude with three points.

First, targeting seconds of latency, not milliseconds, was an important design decision. Seconds is fast enough for all of the use cases we support; there are other systems at Facebook to provide millisecond or microsecond latency in our products. Seconds latency allows us to use a persistent message bus for data transport. This data transport mechanism then paved the way for fault tolerance, scalability, and multiple options for correctness in our stream processing systems Puma, Swift, and Stylus.

Second, ease of use is as important as the other qualities. In our culture of hacking, where "move fast" is printed on posters, having systems available with the right learning curves lets us get prototype applications up and running in hours or days, not weeks. We can then test and iterate. In addition, making debugging, deployment, and operational monitoring easy has greatly increased the adoption rate for our systems, and we plan to continue making it even easier.

Third, there is a spectrum of correctness. Not all use cases need ACID semantics. By providing choices along this spectrum, we let the application builders decide what they need. If they need transactions and exactly-once semantics, they can pay for them with extra latency and hardware. But when they do not – and many use cases are measuring relative proportions or changes in direction, not absolute values – we enable faster and simpler applications.

Our systems continue to evolve to serve our users better. In the future, we plan to work on scaling our system infrastructure. For example, we want to improve the dynamic load balancing for our stream processing jobs; the load balancer should coordinate hundreds of jobs on a single machine and minimize the recovery time for lagging jobs. We are also considering alternate runtime environments for running stream processing backfill jobs. Today, they run in Hive. We plan to evaluate Spark and Flink. We hope to bridge the gap between realtime and batch processing at Facebook.

8. ACKNOWLEDGMENTS

Many people contributed to our systems. KC Braunschweig, Matt Dordal, Li Fang, Eric Hwang, Brett Proctor, Sam Rash, Assaf Sela, Brian Smith, Chong Xie, and Alexander Zhavnerchik built Scribe. Ankur Goenka, Yuhan Hao, Alex Levchuk, Abhishek Maloo, Yuan Mei, Sam Rash, and Gautam Roy contributed to Puma. Ajoy Frank, Daniel Marinescu, and Adam Radziwonczyk-Syta built the predecessor of Stylus. Neil Kodner, Danny Liao, and Jason Sundram provided valuable feedback as they developed Chorus.

9. REFERENCES

[1] Monoid. https://en.wikipedia.org/wiki/Monoid.

[2] Presto. http://prestodb.io.

[3] Rocksdb. http://rocksdb.org.

[4] Samza. http://samza.apache.org.

[5] Scribe. https://github.com/facebook/scribe.

[6] Zeromq. https://zeromq.org/.

[7] L. Abraham, J. Allen, O. Barykin, V. Borkar, B. Chopra, C. Gerea, D. Merl, J. Metzler, D. Reiss, S. Subramanian, et al. Scuba: diving into data at facebook. In *PVLDB*, pages 1057–1067, 2013.

[8] A. Agarwal, M. Slee, and M. Kwiatkowski. Thrift: Scalable cross-language services implementation. Technical report, Facebook, 2007.

[9] 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*, 6(11):1033–1044, Aug. 2013.

[10] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. 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*, 8(12):1792–1803, Aug. 2015.

[11] 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*, pages 577–588, 2013.

[12] 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 2004-20, Stanford InfoLab, 2004.

[13] M. Balazinska, H. Balakrishnan, S. R. Madden, and M. Stonebraker. Fault-tolerance in the borealis distributed stream processing system. *ACM TODS*, 33(1):3:1–3:44, Mar. 2008.

[14] O. Boykin, S. Ritchie, I. O'Connell, and J. Lin. Summingbird: A framework for integrating batch and online mapreduce computations. *PVLDB*, 7(13):1441–1451, Aug. 2014.

[15] N. Bronson, T. Lento, and J. L. Wiener. Open data challenges at facebook. In *ICDE*, pages 1516–1519, 2015.

[16] P. Carbone, G. Fóra, S. Ewen, S. Haridi, and K. Tzoumas. Lightweight asynchronous snapshots for distributed dataflows. *CoRR*, abs/1506.08603, 2015.

[17] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *SIGMOD*, pages 725–736, 2013.

[18] A. Goel, B. Chopra, C. Gerea, D. Mátáni, J. Metzler, F. Ul Haq, and J. L. Wiener. Fast database restarts at Facebook. In *SIGMOD*, pages 541–549, 2014.

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

[20] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter heron: Stream processing at scale. In *SIGMOD*, pages 239–250, 2015.

[21] 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. *PVLDB*, 8(13):2134–2145, Sept. 2015.

[22] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *IEEE Data Mining Workshops*, pages 170–177, 2010.

[23] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST)*, pages 1–10, 2010.

[24] M. Stonebraker, U. Çetintemel, and S. B. Zdonik. The 8 requirements of real-time stream processing. *SIGMOD Record*, 34(4):42–47, 2005.

[25] J. Sundram. Developing data products. Big Data Spain, 2015. https://www.youtube.com/watch?v=CkEdD6FL7Ug.

[26] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sen Sarma, R. Murthy, and H. Liu. Data warehousing and analytics infrastructure at facebook. In *SIGMOD*, pages 1013–1020, 2010.

[27] R. Tibbetts, S. Yang, R. MacNeill, and D. Rydzewski. Streambase liveview: Push-based real-time analytics. *StreamBase Systems (Jan 2012)*, 2011.

[28] 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 *SIGMOD*, pages 147–156, 2014.

[29] J. L. Wiener. Understanding realtime conversations on Facebook. QCON San Francisco, 2015. https://qconsf.com/sf2015/speakers/janet-wiener.

[30] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, volume 8, pages 1–14, 2008.

[31] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 2–2, 2012.

[32] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *SOSP*, pages 423–438, 2013.