Summingbird: A Framework for Integrating Batch and Online MapReduce Computations

Oscar Boykin, Sam Ritchie, Ian O'Connell, and Jimmy Lin
Twitter, Inc.
San Francisco, California
@posco @sritchie @0x138 @lintool

ABSTRACT

Summingbird is an open-source domain-specific language implemented in Scala and designed to integrate online and batch MapReduce computations in a single framework. Summingbird programs are written using dataflow abstractions such as sources, sinks, and stores, and can run on different execution platforms: Hadoop for batch processing (via Scalding/Cascading) and Storm for online processing. Different execution modes require different bindings for the dataflow abstractions (e.g., HDFS files or message queues for the source) but do not require any changes to the program logic. Furthermore, Summingbird can operate in a hybrid processing mode that transparently integrates batch and online results to efficiently generate up-to-date aggregations over long time spans. The language was designed to improve developer productivity and address pain points in building analytics solutions at Twitter where often, the same code needs to be written twice (once for batch processing and again for online processing) and indefinitely maintained in parallel. Our key insight is that certain algebraic structures provide the theoretical foundation for integrating batch and online processing in a seamless fashion. This means that Summingbird imposes constraints on the types of aggregations that can be performed, although in practice we have not found these constraints to be overly restrictive for a broad range of analytics tasks at Twitter.

1. INTRODUCTION

MapReduce, particularly the open-source Hadoop implementation, has given the data management community a powerful "hammer" with which to tackle big data problems. Higher-level dataflow abstractions such as Pig [32] and Cascading provide data scientists with powerful tools to unlock insights from the petabytes residing in modern data warehouses. These solutions focus on high-throughput batch processing, an area that has received substantial attention over the past few years.

More recently, we have seen growing interest in online processing, which represents a different class of problems.[1] Typically, batch analytics are performed on static or slowly changing datasets ranging from terabytes to petabytes. The standard approach for analyzing high-volume data streams (e.g., log data) is to run periodic batch jobs (e.g., hourly). Batch processing frameworks (e.g., Hadoop) primarily focus on job throughput and often have difficult handling latency-sensitive jobs, e.g., from interactive analyses. There is often a need for low latency responses to potentially complex queries over high-volume, infinite streams of data: this demands online processing capabilities. Although data management on streams is not new, how to best integrate batch and online processing in a production environment remains an open question. This is the challenge that we tackle in this paper.

The key insight of this work is that certain algebraic structures provide the theoretical foundation for seamlessly integrating batch and online processing. From this starting point, we have built Summingbird, a data processing framework that supports both batch and online computations formulated in terms of these algebraic structures. Summingbird provides a domain-specific language (DSL) implemented in Scala for expressing analytical queries that transparently generates either Hadoop jobs (batch computations) or Storm topologies (online computations) without requiring any changes to the program logic. Furthermore, Summingbird can operate in a hybrid processing mode that transparently integrates batch and online results to efficiently generate up-to-date views over long time spans. Although aggregations in Summingbird are restricted to certain algebraic structures, in practice, we find that our language is sufficiently expressive to capture large classes of analytical queries in a production environment.

The contribution of this work lies primarily in the design of Summingbird (Section 3) and the use of probabilistic data structures implemented as commutative semigroups for a broad range of analytical tasks (Section 4). Additional features such as left joins and a hybrid processing mode are

[1]A clarification on terminology: the terms *online*, *real-time*, and *streaming* are sometimes used interchangeably, and often without precise definitions. To us, *online* means that data are processed as they are being generated, in contrast to *batch*. Processing in real time suggests low latency responses, although online processing doesn't always mean real time, e.g., an expensive computation. The literature on *streaming* algorithms usually implies limited working space. This is often, but not always, the case in online processing; e.g., a system might have access to large distributed key–value stores for retaining partial results.

also discussed (Section 5). We share our experiences running Summingbird in production at Twitter, reflecting on strengths and future work (Section 6). Since Summingbird executes analytical workflows on either Hadoop or Storm, its performance is no worse than programs that are directly written for each execution framework (but see additional discussion in Section 6). In this respect, the contribution of our work lies not in increasing system performance, but rather in boosting productivity for developers by providing a unified framework for batch and online analytics.

2. BACKGROUND

We begin by describing pain points that we and our colleagues have experienced over the past several years in managing batch and online computations at Twitter. This provides broader context about the problem that Summingbird tries to solve. Our situation is hardly unique, and the analytical queries that we describe are common in domains ranging from social media to online retailing.

Hadoop and batch analytics. A number of large Hadoop clusters comprise the core of Twitter's data warehouse. Data are imported from a variety of sources, including structured databases (e.g., user profiles and the interest graph), unstructured text (e.g., Tweets), and semi-structured interaction logs [23]. For analytics and building data products, data scientists typically use a higher-level dataflow language such as Pig or Scalding (Twitter's Scala API to Cascading). This is a mature production system, aspects of which we have previously described elsewhere [24, 25].

Suppose we are interested in the number of user interactions with some object. This could be clicks on links, impressions of recommendations, numbers of logins on a particular client, etc. Typically, we wish to know the raw counts as well as descriptive statistics such as mean, median, and percentile breakdowns. Often, we want to perform cardinality estimation or focus on the heavy hitters. In most applications, exact answers are not necessary as long as the errors are bounded—as an example, for links that are clicked more than one thousand times, it is sufficient to know the click count to the nearest ten.

Answering these questions on data stored in the Hadoop data warehouse is straightforward using Pig or any other comparable analytical tool. Whatever is not provided as a language primitive can be easily implemented as a UDF. This suffices for retrospective analysis and building offline data products (e.g., machine learning models), but what if we desire these answers in real time on live data, with subsecond latency?

Online analytics. Over the years, Twitter has developed several generations of scalable systems for counting events in real time. From these counters we can answer most of the questions described above, but the issue is that the systems are not well integrated into the rest of our analytics infrastructure. This creates substantial friction during the development cycle: a data scientist would use familiar Hadoop-based tools for retrospective analyses or model construction. However, since a production system must run in an online environment, she must write a separate set of interfaces for gathering and processing real-time signals. To mitigate the pain of working with two separate code bases, the common development practice is to encapsulate core functionality in a library that is agnostic with respect to the processing model, and then separately "hook" the library into either a batch or an online execution framework.

This, however, is an imperfect solution for a few reasons: it still requires the developer to build and maintain code that binds to different execution frameworks. In many cases, two sets of aggregation logics must be created due to the inherent differences between batch and online processing. Moreover, when writing code that is supposedly agnostic to the processing model, it is easy to forget the constraints of the execution environment. For example, scaling out in Hadoop is often as simple as increasing the number of reducers, but the ability to scale out in an online environment by splitting streams is more restrictive. Thus, it is not uncommon to prototype a particular feature in Hadoop and then discover that the implementation is too slow to run in an online production setting. As another example: in batch processing, it is possible to take advantage of disk storage if in-memory data structures grow too large, but in online processing this is usually not possible due to latency requirements. Managing memory limitations is particularly important when trying to track large event spaces that follow Zipfian distributions, due to the presence of long tails.

In addition to the major pain of, essentially, writing everything twice (once for batch processing and once for online processing), there was no standard online processing framework at Twitter until recently. The systems for counting events in real-time were responsible only for gathering signals and offered little support in helping a client manipulate and process them. Over the past several years, the result has been a proliferation of custom one-off processing engines for various specialized tasks. A good example that illustrates all these issues is described in a previous paper about Twitter's real-time related query suggestion architecture [28]. In that paper, we shared the case study of how a batch-oriented system was first built, only to be replaced immediately by an online system that depended on a custom processing system. Because the system was specifically built to implement a particular type of query-suggestion algorithm, it would be difficult to reuse the code for other related tasks.

Framework standardization. Today, Twitter has made progress in addressing the problem of the proliferation of one-off systems. Hadoop remains the standard for batch analytics, although it is mostly accessed via higher-level abstractions such as Scalding and Pig. For online processing and real-time analytics, Storm has emerged as the standard execution framework. Hadoop requires no introduction, but here we provide a quick overview of Storm.

Storm [37] is an open-source stream processing framework released by Twitter in 2011,[2] now an Apache project. A Storm cluster executes user-submitted topologies (the equivalent of Hadoop jobs). A topology represents a directed acyclic dataflow graph comprised of "spouts", which are sources of streams, and "bolts", which perform stream transformations. Spouts are usually connected to message queues, from which they consume sequences of tuples. The execution framework is responsible for running topologies over a cluster, handling task placement, message routing, and ensuring robustness in the presence of failures. The framework provides options for either best-effort message delivery or at-least-once message delivery. Bolts execute in parallel across nodes in the cluster, and Storm provides "groupings"

[2]`http://storm-project.net/`

to specify connections between bolts. For example, a "fields grouping" performs a group-by to ensure that all messages with the same field are delivered to the same bolt.

It might be tempting to eliminate Hadoop and build an analytics platform entirely around online processing, but this is impractical for several reasons. Online processing does not obviate the need to store the raw data: fault tolerant systems for distributed online processing are usually forced to choose between an exactly-once message delivery guarantee that is expensive and slow or looser guarantees (e.g., at-least-once, best-effort) that are simpler and faster. Storm chooses the second option, which is why online processing still needs to be "backed up" by traditional batch processing. Furthermore, to correct errors in the online processing pipeline or to handle upgraded capabilities (e.g., an improved language detector), it is often necessary to rerun analytics on historical data and recompute results. Although it is possible to simply replay archived data through Storm, this would be hugely inefficient. In practice, when replaying historical data through Storm, we are at best a small factor faster than consuming the sources in real time, compared to the arbitrary scale out that we are able to achieve with Hadoop.

Need for batch/online hybrids. It is clear that Hadoop and Storm both have their place, but merely standardizing on the two processing frameworks does not solve the problem of a developer needing to write everything twice. It would be desirable to have an abstraction for expressing analytical queries that is agnostic to batch or online processing, and have a system automatically generate Hadoop jobs or storm topologies as appropriate. Summingbird does exactly this.

Another common use case for Summingbird is when we desire aggregations over large volumes of historical data, but also need up-to-date results. For example, suppose we want to keep track of impression counts for all Tweets over the entire life of the service: this obviously entails processing large volumes of log data, which is more suited to batch processing—but at the same time, we want real-time counts so we can identify "hot content" with minimal latency. One obvious solution would be to run hourly Hadoop jobs, and then "fill in" the latest hour with Storm.

Now take the perspective of a querying client who wishes to consume the results of these analyses. The client would need to implement complex logic for merging results from the batch and online computations in a robust manner. For example, the client needs to ensure that messages are not double counted by both Storm and Hadoop, and also handle the opposite problem, when gaps appear between Hadoop and Storm coverage. The client must also handle various abnormal operating scenarios, the most common of which is batch processing delays. When the Hadoop cluster is operating beyond capacity, jobs may not generate results in a timely fashion, in which case the client must continue gathering results from Storm and wait for Hadoop to "catch up". Summingbird transparently handles these issues to provide an integrated view of the data to querying clients.

We emphasize that the biggest pain point that Summingbird tries to address is developer productivity, not runtime performance, since ultimately, what runs is either a "vanilla" Hadoop job or Storm topology. Thus, the contributions of the language lie in the abstractions it introduces and its balance between simplicity and expressivity with respect to a broad range of analytical queries encountered at Twitter.

3. COMPUTATIONAL MODEL

Summingbird was developed as a solution to the pain points described in the previous section. Although our discussion is couched within the Twitter production environment, we believe that many other organizations face the same challenges. The goal of the project is to provide a MapReduce-like programming model that generalizes over both batch and online computations, thus allowing code to be written once and executed on multiple processing frameworks. Furthermore, the programming model provides abstractions that allow results from batch and online processing to be integrated in a seamless manner. In concrete terms, Summingbird is an open-source domain-specific language implemented in Scala.

To enable efficient aggregations in both batch and online processing, Summingbird takes advantage of commutative semigroups, a type of algebraic structure (more details below). This means that Summingbird imposes constraints on the types of aggregations that can be performed in the "reduce", although in practice we have not found this to be overly restrictive in the types of analytical queries that are possible with the language.

Summingbird consumes and generates two types of data: *streams*, which are (potentially) infinite sequences of tuples, and *snapshots*, which represent the complete state of a dataset at some point in time.

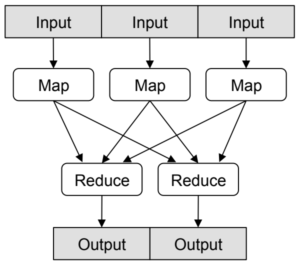
A **producer** is an abstraction over the state of a data transformation in Summingbird (i.e., a node in a dataflow graph). A producer has an associated **source**, which materializes objects of a particular type for processing. At the beginning of a program, in the online setting, sources are usually attached to message queues from which data are consumed in real time. In the batch setting, sources might read from files that are stored in HDFS. From a producer, a workflow can be constructed via transformations described below, each of which creates another producer.

A **platform** describes how Summingbird workflows are instantiated in a specific execution framework. Currently, Summingbird provides platform implementations for Storm, Scalding, and an in-memory execution engine (primarily for testing purposes). Scalding is Twitter's Scala API to Cascading, an open-source framework for building dataflows that can be executed on Hadoop. Thus, Summingbird is connected to Hadoop for batch processing indirectly via Scalding and Cascading.

Summingbird jobs can generate two types of outputs: A **store** represents an abstract model of a key–value store. In the online setting, a store receives partial results that are combined with its present state, and might be backed by memcached, MySQL, or HBase; reasonably low-latency reads and writes are a requirement. In the batch setting, a store contains a snapshot of the aggregated value for each of its keys, which is usually materialized to disk. A **sink** materializes (unaggregated) tuples from the producer, typically after some manipulation. Sinks might populate another message queue for further processing (in the online case) or simply write data to disk (both online and batch processing).

Although Summingbird is capable of creating complex DAG workflows, let us begin by focusing on the canonical use case. Just like in standard MapReduce, a Summingbird job consists of two phases: the "map" phase, where per-record computations are applied in parallel to generate

```
// Running in Hadoop (via Scalding/Cascading)
Scalding.run {
  wordCount[Scalding](
    Scalding.source[Tweet]("source_data"),
    Scalding.store[String, Long]("count_out")
  )
}
```



```
// Running in Storm
Storm.run {
  wordCount[Storm](
    new TweetSpout(),
    new MemcacheStore[String, Long]
  )
}
```

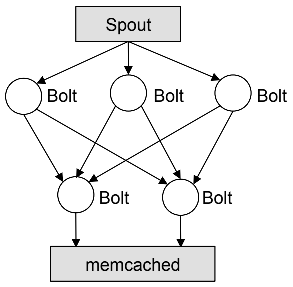


Figure 2: Two separate instantiations of the Summingbird word count program on different execution frameworks. Code snippet on the left uses the Scalding platform implementation, which creates a Hadoop job (via Cascading) running over files stored in HDFS. Code snippet on the right uses the Storm platform, which yields a Storm topology that processes data streaming from a spout in real time.

```
def wordCount[P <: Platform[P]]
  (source: Producer[P, String],
   store: P#Store[String, Long]) =
   source.flatMap { sentence =>
      toWords(sentence).map(_ -> 1L)
    }.sumByKey(store)
```

Figure 1: A simple word count example in Summingbird.

an arbitrary number of intermediate key–value pairs, and the "reduce" phase, where values are aggregated by key. A simple word count example in Summingbird is shown in Figure 1. In the `flatMap` we apply a tokenizer that breaks sentences from the `source` and for each token generates a key–value pair with the token as the key and one as the value. The keys are then aggregated with `sumByKey` (i.e., the "reduce") and materialized in `store`. A lot of complexity is hidden in `sumByKey`, as we detail shortly. To illustrate the difference between stores and sinks, if we insert a `write(sink)` before the call to `sumByKey`, the program will materialize (e.g., to an HDFS file or another message queue) each parsed token and the count one as a key–value pair. In contrast, if the `write` call is inserted after `sumByKey`, the program will materialize a stream of partial aggregations (i.e., deltas) for each word.

Note the Summingbird program in Figure 1 references only the source and store abstractions, and thus remains agnostic to the processing model (whether batch or online). To run the job on Hadoop, we supply the Scalding platform, as in Figure 2 (left), which creates a Hadoop job that processes input data on HDFS and writes results back to HDFS. Alternatively, we can supply the Storm platform, as in Figure 2 (right), yielding a Storm topology that processes Tweets in real time and stores the counts in memcached. To precisely explain the semantics of a group-by on an infinite stream requires a discussion of algebraic structures, which we defer to Section 3.2.

Below, we describe in more detail the "map" and "reduce" computations in Summingbird.

3.1 "Map" Computations

Just as in MapReduce, the "map" phase of a Summingbird program specifies per-record computations. These embarrassingly parallel computations are amenable to scale out in both batch and online execution. Three types of per-record computations are provided for a producer:

`flatMap[T, U](fn: T => List[U]): List[U]`
The `flatMap` method takes a function that generates a list of values, applies the function to each object consumed from the source, and then flattens together all returned lists. The result is another producer. The word count example uses `flatMap` because the tokenizer generates a list of tokens for each input sentence.

`map[T, U](fn: T => U): List[U]`
The `map` method takes a function and applies it to each object consumed from the source. The result is another producer.

`filter[T](fn: T => Boolean): List[T]`
The `filter` method takes a binary predicate and discards objects from the source for which the predicate returns false. The result is another producer.

In addition to the basic map and map-like computations described above, the producer API provides a few other additional features such as the ability to merge sources from two producers and to perform left joins (more details in Section 5.1).

3.2 "Reduce" Computations

In Summingbird, a call to `sumByKey` triggers a reduce operation where key–value pairs from the mapper stage are grouped by keys and all values associated with the same key are aggregated. The results are typically materialized in a store such as a file on HDFS or an in-memory key–value store. To enable efficient aggregation and the integration of batch and online processing, values in Summingbird are limited to certain algebraic structures. First, a few definitions:

Definition 1. A **semigroup** is a tuple comprised of a set $\mathcal{M}$ and an associative binary operation $\diamond : \mathcal{M} \times \mathcal{M} \to \mathcal{M}$.

That is, for all $m_1, m_2, m_3 \in \mathcal{M}$,

$$(m_1 \diamond m_2) \diamond m_3 = m_1 \diamond (m_2 \diamond m_3)$$

The $\diamond$ operation can be viewed as a generalization of arithmetic addition (sum) over arbitrary types.

Definition 2. A **monoid** is a semigroup that also contains an identity element ϵ, such that for all $m \in \mathcal{M}$,

$$\epsilon \diamond m = m \diamond \epsilon = m$$

The identity element can be viewed as a generalization of zero for arithmetic addition (sum).

Definition 3. A **commutative** semigroup (or monoid) is a semigroup (or monoid) where the associated binary operation is also commutative, that is, $m_1, m_2 \in \mathcal{M}$,

$$m_1 \diamond m_2 = m_2 \diamond m_1$$

Commutativity allows us to reorder operands in a computation without affecting results.

In the general case, to guarantee the correctness of computations, values in Summingbird must be at least commutative semigroups (and of course, they can be algebraic types with more properties such as commutative monoids). In the reduce phase of Summingbird, `sumByKey` aggregates all values with the same key using the associative binary operation belonging to the semigroup ($\diamond$).

The introduction of semigroups allows us to precisely define the semantics of `sumByKey` on an infinite stream of key–value pairs. Formally, each aggregation is `get(K)` $\diamond$ `V`, i.e., we incrementally apply the associative operation to the current value held in the store and each new value. In practice, however, this approach does not scale to the volumes of data that Twitter processes. Our solution is to buffer output key–value pairs after the group-by and perform aggregations in batches: buffer sizes are entirely user configurable, and a typical setting might be to process batches of 10K key–value pairs, but no less frequently than every 30 seconds. Processing in small batches also allows us to take advantage of efficient multi-get and multi-put operations that are supported by many key–value stores. Summingbird also provides options for buffering key–value pairs prior to network shuffling on Storm and performing aggregations on the map side; this is entirely analogous to combiners in MapReduce and reduces network traffic.

Note that pre-shuffling and post-shuffling batching should be considered optimizations, since they do not affect result correctness (due to the commutative guarantees provided by the semigroup). It is more accurate to think of these as knobs that the developer can tune to control the tradeoff between latency and resource consumption. For example, if an application really demands updates for each key–value pair as it is encountered, batching can be turned off, and we would simply need to allocate sufficient resources in the store implementation to support the query load.

In practice, most of the value types used in analytical processing tasks at Twitter are commutative monoids. The next section describes them in more detail, but here we discuss how the properties of associativity and commutativity relate to online and batch processing more generally.

In the online case when a single processor is consuming from a single message queue or in the batch case when a single processor is consuming a single file, semigroups are sufficient for correctness, i.e., we do not need commutativity, since input key–value pairs will be processed sequentially. Note, however, that even in this simple scenario commutativity is needed to handle transient data glitches that are often encountered in online processing such as out-of-order message delivery.

In both the online case and the batch processing case, we need commutativity to guarantee correctness as soon as we introduce partitioned input, that is, multiple processors consuming from a single message queue or mapping over multiple files on HDFS. This is in fact the common case, since there is no other easy way to build scale-out distributed processing systems. Commutativity is needed because partial results from each partition may be aggregated in arbitrary order.[3] In the online processing case, commutativity lets us handle issues like out-of-order message delivery. For both online and batch processing, commutativity further enables certain optimizations such as combiners in MapReduce and partial aggregations in Storm (as discussed above).

We have recently been considering cases where commutativity may be relaxed. With respect to an error function $\mathcal{E}$, one way to formalize this might be:

$$\Pr[\mathcal{E}(m_1 \diamond m_2, m_2 \diamond m_1) < \epsilon] > \delta$$

for some given ϵ and δ that are specified by the algorithm designer. Of course, errors are likely to compound as more operands are swapped, but in an online setting, monoids that obey this property will be resilient to a certain (quantifiable) amount of transient data glitches without needing to be fully commutative. Nevertheless, this relaxed notion of commutativity is mostly a curiosity at this point, since the data structures that are commonly used in analytics tasks at Twitter form commutative monoids, as we discuss next.

4. ALGEBRAIC STRUCTURES

Although values in Summingbird must be at least commutative semigroups to enable efficient aggregation in both the online and batch contexts while maintaining correctness, this has not been a limitation in practice at Twitter. We have implemented a number of types that capture a broad range of analytical tasks, described below.

One simple monoid is the set of integers with addition ($+$) as the binary operation and the identity element zero. This means that Summingbird is capable of any analytical query that involves counting events. Integers also form a monoid under multiplication ($\times$) with the identity element one, so Summingbird can handle queries involving products, such as computing probabilities (although in most cases we would prefer working with log probabilities). Sets of arbitrary objects are also monoids with respect to set union and the empty set as the identity—these are useful for keeping track of set membership, e.g., the unique users who have clicked on a link. The operators `max` and `min` form semigroups over the set of integers (because the types lack a distinguished identity element). These types are obviously useful for a variety of analytical queries. All of the types described above are also commutative.

Monoids can also be composed from other monoids in more complex data structures. For example, the set of tuples

[3]One possible way around this in the batch processing case is to sort intermediate data by timestamp (if available), but this introduces potential scalability bottlenecks and eliminates the possibility of intermediate aggregations.

```
def wordCount[P <: Platform[P]]
  (source: Producer[P, Query],
   store: P#Store[Long, Map[String, Long]]) =
   source.flatMap { query =>
      (query.getHour, Map(query.getQuery -> 1L))
    }.sumByKey(store)
```

```
def wordCount[P <: Platform[P]]
  (source: Producer[P, Query],
   store: P#Store[Long, SketchMap[String, Long]])
  (implicit countMonoid: SketchMapMonoid[String, Long]) =
   source.flatMap { query =>
      (query.getHour,
       countMonoid.create((query.getQuery, 1L)))
    }.sumByKey(store)
```

Figure 3: Example of counting query frequency by hour using hashmaps (left) and count-min sketches (right).

of monoids are themselves monoids under element-wise application of the $\diamond$ operation. For example, the set of all pairs of integers forms a monoid with element-wise addition, i.e., $(a, b) \diamond (c, d) = (a + c, b + d)$ and the identity element $(0, 0)$. This can be useful for concurrently keeping track of multiple counts over a series of observations. One specific example is an algebraic group for computing moments (e.g., mean, variance, skewness, kurtosis, etc.), where we keep track a tuple consisting of $(1, x, x^2, x^3, \ldots)$. These tuples can be aggregated by element-wise addition, from which standard moments can be straightforwardly derived.

Another useful monoid is the set of all hashmaps (i.e., associative array) that map from arbitrary keys to values that are monoids. Similar to complex tuples, the associative operation is key-wise application of the $\diamond$ operation with the empty hashmap as the identity element. If the $\diamond$ operation is commutative, then so is the hashmap monoid. The common use case for this type is to compute histograms

One more non-trivial example worth a passing mention is our monoid treatment of minhash [6], a fast probabilistic algorithm for computing similarity between arbitrary objects. In this case, $\diamond$ computes the minimum of two signatures (i.e., the "min" in minhash).

Beyond these examples, much of the power of Summingbird derives from monoid implementations of common probabilistic data structures. These types were motivated by the realization that in many analytics scenarios, exact counts are not necessary—one might argue that exact counts are not even desirable due to the noise inherent in human behavior, e.g., accidental clicks. In concrete terms, it doesn't matter much if a retweet counter displays 141 or 142. Any count is "good enough" as long as the value is within, say, 1% of the true count; in fact, front-end designers might choose to round the value to the nearest ten anyway. We hasten to emphasize that for some analytics scenarios (e.g., for billing advertisers), there is *no* tolerance for error and exact counts are absolutely necessary—for those tasks, probabilistic data structures are not appropriate.

Summingbird leverages the tolerance for errors in many analytical tasks by the use of probabilistic data structures, which are primarily based on hashing, and where the source of error often comes from hash collisions. The use of such data structures means that Summingbird processes every input key–value pair, unlike a strategy based on sampling. Of course, there is no reason why Summingbird cannot also incorporate sampling, although proper sampling requires some knowledge of the underlying distribution. In our experience, probabilistic data structures can be used as "black boxes" by engineers who have no understanding of the underlying implementations. In contrast, proper sampling usually requires engineers to have a much more sophisticated knowledge of statistics. Below, we describe a few useful commutative monoids:

Bloom Filters [4] are compact probabilistic data structures for keeping track of set membership, i.e., whether an element is a member of a set or not. False positive matches are possible (e.g., a Bloom filter might assert that an element is in the set when in reality it is not), but false negatives are not possible. One canonical use case for Bloom filters is to keep track of users who have been exposed to a certain event (e.g., Tweet, recommendation, etc.) in order to avoid duplicate impressions. For Twitter, this involves keeping track of $O(10^8)$ objects/day for $O(10^8)$ users. The asymmetric error properties of Bloom filters (i.e., no false negatives) means that a user will never be exposed to the same treatment twice. Bloom filters provide accuracy/space tradeoffs: given a desired error rate and a given capacity (determined a priori by the developer based on different application scenarios), we can appropriately size the filter.

Hyperloglog counters [10, 15] are compact probabilistic data structures for cardinality estimation (i.e., size of a set). A canonical use case of hyperloglog counters is to keep track of the number of *unique* users who have performed a certain action, e.g., retweeted or favorited a Tweet, clicked on a link, etc. For Twitter, a naïve exact solution based on sets would be impractical for $O(10^8)$ users, particularly for events with high cardinalities (e.g., retweets of celebrities' Tweets or Tweet impressions). These counters are also useful for computing graph statistics such as the size of a node's second-degree neighborhood (e.g., followers of followers [29]). A hyperloglog counter occupies $O(\log \log n)$ space for cardinalities of up to n. These data structures are tunable within the (ϵ, δ) framework—that is, achieving $(1 \pm \epsilon)$-approximation with probability δ. The choice of parameters affects the constant factor in the size of the counters.

Count-Min Sketches [9] are compact probabilistic data structures for keeping track of frequencies (i.e., counts) associated with events. A canonical use case of count-min sketches is to keep track of the number of times a query was issued to Twitter search within a span of time. In general, count-min sketches can be used for building histograms of events. The data structure is based on hashing objects into a two dimension array of counts using a series of hash functions. Given a desired error bound in the (ϵ, δ) model based on the application scenario, we can compute the size of the data structure that underlies the count-min sketch.

As a concrete example, Figure 3 shows two versions of a Summingbird program to generate hourly counts of search queries. The left version provides an exact solution by keeping track of the query counts in a hashmap (which is a commutative monoid, as previously discussed). This solution will not scale because there is not enough memory to keep track of all unique queries and their counts. The version on the right replaces the hashmap with a count-min sketch: note that the program has exactly the same logic. The couple of

Task	Exact	Approximate
Set membership	set	Bloom filter
Set cardinality	set	hyperloglog counter
Frequency count	hashmap	count-min sketch

Table 1: A summary of common analytical queries and the exact/approximate versions of the monoid used in Summingbird.

```
def urlCount[P <: Platform[P]]
  (tweets: Producer[P, Tweet],
   urlExpander: P#Service[String, String],
   store: P#Store[String, Long]) =
   source.flatMap { tweet =>
      extractUrls(tweet.getText)
    }.map { url => (url, 1L) }
     .leftJoin(urlExpander)
     .map {
      case (shortUrl, (count, optResolvedUrl)) =>
        (optResolvedUrl.getOrElse("unknown"), count)
    }.sumByKey(store)
```

Figure 4: Summingbird join example for counting resolved URLs in Tweets using a URL expander.

extra lines are necessary to properly initialize the count-min sketch (type `SketchMap`), and the `(implicit countMonoid: ...)` statement declares that whatever is calling this code must have the monoid available in its implicit scope.

In summary, the three most commonly-used probabilistic data structures in Summingbird are shown in Table 1 with their exact equivalents. In our experience, the algebraic types discussed above appear to be sufficient for large classes of analytical queries that Twitter data scientists issue on a daily basis.

5. ADDITIONAL FEATURES

5.1 Joins

Summingbird can perform left joins via the **service** abstraction. As described in Section 3, a producer has a source which materializes the input; in addition, it can have a service, which can be viewed as a mapping from values of type `T` to values of type `U`. Possible implementations for a service include gets from in-memory key–value stores, database queries, remote procedure calls, etc. In data warehousing parlance, a service would hold a dimension table to be joined with the fact table (i.e., the source).

An example is shown in Figure 4, which computes counts of URLs that are contained in Tweets. Since the URLs contained in the text are always shortened, we need to look up the expanded URLs: this is accomplished by the service. In the first `flatMap` call, we extract all URLs from the Tweet text. These URLs are then left joined with the service. More precisely, `leftJoin` takes a producer that generates `(K, V1)` pairs and a service `K` $\rightarrow$ `V2` to yield a producer that generates `(K, (V1, Option[V2]))`, where `Option` is the Scala monad for an optional value of type `V2`. In this case, both types are strings. After the left join, a call to `map` converts each joined result into a key–value pair with the expanded URL as the key (or the special token "unknown" if the URL expander does not find an expansion) and the value one. Finally, a call to `sumByKey` computes the aggregate counts.

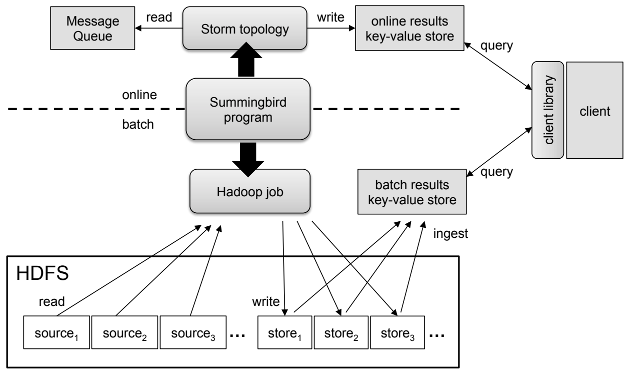


Figure 5: The Summingbird architecture for hybrid online/batch processing.

5.2 Hybrid Online/Batch Processing

Summingbird's ability to integrate batch and offline analytics supports a hybrid processing model where we are able to efficiently and seamlessly provide access to aggregations across long time spans while maintaining up-to-date values with minimal latency. One common use case is to keep track of counts for a very large event space (e.g., all Tweets) across a long time span (e.g., for the life of the service).

The basic idea behind hybrid processing is to periodically "roll up" aggregates using Hadoop and to "fill in" results from real-time data using Storm. While this general design pattern is decades old [22], our architecture illustrates the role of semigroups in allowing Summingbird to integrate batch and online results while transparently preserving correctness. We begin by highlighting two key features:

- Hybrid processing does not require changing the logic of Summingbird programs—the same exact program runs in either batch or online mode. The only additional requirements from the developer's perspective are a few metadata extractors to define how inputs are grouped in batches and a modest number of hooks into other parts of Twitter's infrastructure. Any additional "bookkeeping" is performed behind the scenes without the developer's knowledge.
- Downstream clients are completely shielded from the details of the hybrid processing. Integration of results from Hadoop and Storm are transparently handled by the client library, which presents a simple key–value interface.

The complete architecture of Summingbird running in hybrid mode is shown in Figure 5, but we begin by introducing a few basic concepts. In addition to the Summingbird program, we assume that the developer has defined two additional classes: the `TimeExtractor` extracts timestamps out of sources, and the `Batcher` maps a timestamp onto an integer batch id. The batch id defines disjoint batches (e.g., hourly), the level of granularity at which aggregations are computed, and internally, Summingbird will never partially aggregate across two different batches. In most cases, the implementations of these classes are trivial, as nearly all log events are annotated with a timestamp, which can be straightforwardly mapped (for example) to the nearest hour.

Summingbird in hybrid mode requires integration with other infrastructure at Twitter: On the source end, we assume the existence of message queues that deliver event data in real-time and that the same data are also deposited onto HDFS (for example, see Lee et al. [23] for a description of Twitter's logging infrastructure). On the store end, we assume the existence of two separate key–value stores: one for the batch results, and the other for the online results (although the client library transparently handles results merging). Twitter uses open-source software for most of these components, but even in cases where we rely on an internal implementation, comparable open-source equivalents are readily available.

Periodically, a Summingbird job on the Scalding (Hadoop) platform is triggered to compute aggregates on the next incremental source batch that has been deposited in HDFS. The process of physically launching these jobs is accomplished through Mesos [16], although conceptually it is no different from cron. The mapping from batch ids to physical HDFS paths can be deterministically computed since data are structured according to a `YYYY/MM/DD/HH/` physical layout. The data import pipeline is engineered so that a directory does not appear until all the files contained in that directory have arrived [23], so it is not possible to process partially-imported results.

A minor detail here is worth noting: many log messages that are generated near the end of an hour appear in the directory for the next hour due to unavoidable latencies in the log pipeline. We address this issue by running the Hadoop jobs across a moving window of two hours, but discarding events that do not belong to the relevant batch. The size of the moving window is configurable, but in practice, we have found that a negligible fraction of events arrive more than an hour late. The results of these Summingbird jobs are materialized in HDFS files (at known locations) that hold the aggregated values for the relevant batch.

The batch results key–value store polls HDFS periodically for the appearance of newly-created stores, and when one appears, the contents are ingested. Since the key–value pairs on HDFS capture results for only that source batch, the ingestion process requires applying the semigroup associative operator (i.e., $\diamond$) to aggregate those key–value pairs with the current contents of the batch results store. However, instead of storing `(K, V)` pairs directly, the contents are transformed into `(K, (batchId, V))` pairs—this data structure captures the value of a particular key up to and including the specified batch id. This transformation is performed "behind the scenes" without the developer's knowledge. Note that folding the batch id into the value to form a tuple is necessary because our key–value store implementation does not have an atomic ingest feature, or otherwise it would be sufficient to store a global batch id for the entire store.

In parallel with the batch jobs, the same Summingbird program is continuously executed in a Storm topology, and the results are deposited in an online results key–value store. Instead of aggregating by key `K`, however, the system automatically builds a compound key `(K, batchId)` for performing the grouping. These represent the online partial results for each batch.

Let us now turn our attention to the client. The client library maintains connections to both the online and batch results store. All queries first go to the batch results store: by comparing the wall clock time and the batch id from the result, the system knows how "far behind" the value is. Based on this, the client can figure out how many values need to be "filled in" from the online store, which is keyed by `(K, batchId)`. It can then issue appropriate requests to the online results store. The final, up-to-date value is arrived at by aggregating all the partial values—once again, the validity of these operations is licensed by the fact that the values are (at least) semigroups. Note that all this logic is handled by the client library, and thus downstream systems are presented with a simple key–value interface.

Typically, the batch results key–value store is much larger and backed by durable storage, whereas the online results are kept in memory-resident key–value stores (e.g., memcached). To prevent memory overflow, keys are pruned based on a time-to-live (TTL) setting. The TTL is tuned such that, under normal operating circumstances (and within a "margin of safety"), there will be no "gap" between the batch and online results—that is, largest batch id in the batch results store will be greater than the smallest batch id in the online results store. However, during times of excessive load on the Hadoop cluster or outages, gaps may appear when the online results coverage is not sufficient to fill in where the batch results end. In this case, a client lookup will fail.

6. PRODUCTION EXPERIENCE

As previously discussed in Section 2, Summingbird evolved from several generations of systems within Twitter for counting and processing events in real-time that date back at least 2009. The project itself, however, began in late 2012. Summingbird has been in production since early 2013, and it was open-sourced in September of that year.[4]

The adoption of Summingbird has grown over time, and here we provide a snapshot of its deployment status within Twitter. Currently, there are a few dozen Summingbird jobs that are running in production, which account for roughly half of the jobs that depend on online analytics. A typical Summingbird job might process 1–20 MB/s (around 100-250K events per second) in the online mode and execute hourly in batch mode, translating to several hundred Hadoop jobs daily. The results of these jobs typically feed online dashboards for internal monitoring purposes or generate signals that serve as input to user-facing products.

Overall, there are roughly a dozen teams at Twitter who run production jobs on Storm, and half of them use Summingbird exclusively. Many of the other teams were early adopters of Storm, before Summingbird was sufficiently mature, and are considering migrating to it. In the same way that Pig or Hive emerged a few years ago as the preferred interface for Hadoop (as opposed to writing Java programs), Summingbird is becoming the standard tool to access online analytics capabilities within the company.

We have found that most online analytics needs at Twitter can be handled by Summingbird, but admittedly, there is a large amount of self-selection in users. However, since the framework evolved out of several years of experience building and running disparate online analytics systems, we are fairly confident to have captured at least a large fraction of the cumulative needs of the organization.

In many ways, the current state of online analytics processing on high-volume event data is similar to the state of Hadoop-based data warehousing several years ago. Data infrastructure builders discovered that simple primitives such

[4]`https://blog.twitter.com/2013/streaming-mapreduce-with-summingbird`

as selections, joins, aggregations were sufficient to encompass most use cases (at the time), which made simple domain-specific languages such as Pig highly successful. Of course, as time went on, the types of analyses data scientists wish to perform increased in sophistication to include advanced analytics such as data mining, machine learning, etc. In the same way, we see a co-evolution of needs and capabilities for online analytics: being able to handle "the easy cases" is an essential first step that Summingbird solves, but we have no doubt that new needs will arise in the future.

One particular limitation we have encountered so far is the need for generic folds (that are available in functional programming languages): we wish to retain some state, which is altered by an incoming event to produce new state. Unless the updates themselves form a semigroup, this use case cannot be efficiently handled by Summingbird.[5] One specific example of this is online learning, say with a linear model using stochastic gradient descent [5]. In the general case, a model learned online will be different from a model learned using a batch algorithm; even the order in which the training examples are presented to the online learner affects the final model parameters. Thus, it is difficult to develop an approach that generalizes across online learning, batch learning, and distributed learning (over different partitions) while maintaining stability in the output models and theoretical guarantees. We are aware of recent work by Izbicki [17] in formulating standard machine learning constructs in terms of algebraic types, and it would be interesting future work to see how those results can be integrated into Summingbird.

In general, Summingbird users are able to take advantage of probabilistic data structures (Bloom filters, hyperloglog counters, count-min sketches) with relative ease, since there is a fairly straightforward mapping between exact data structures and their probabilistic equivalents (see Figure 1). Most engineers do take the effort to learn, at least at a high level, how the data structures work, but this is not an absolute pre-requisite for writing Summingbird programs. However, the use of probabilistic data structures requires us to educate developers about their proper usage and limitations—in particular, setting parameters to achieve the desired compactness/accuracy tradeoff and understanding the effects of certain skewed distributions. To assist developers, we explicitly expose the approximation errors for structures where it is easy to do via an API that provides error bounds and the probability that the true value lies within the error bound. With these metadata exposed, errors can be monitored and the data structures can be retuned when necessary.

Although the goal of Summingbird is to increase developer productivity as opposed to analytics performance, we see opportunities for faster job execution as well. Our design follows a traditional separation of logical plan from physical plan, with many opportunities for query optimization during plan compilation. One simple example currently implemented is that in the absence of developer-specified constraints, the system tries to heuristically tune batch sizes in Storm. Another trivial example is to coerce the first set of bolts in a topology to be co-located with the spouts (sources), thus saving a serialization step and a network hop. The upshot is that, even now, Summingbird jobs are often faster than hand-written Storm topologies. This is a nascent aspect of the project, although the applicability of common optimizations in the data management literature is fairly evident. Furthermore, the logical/physical plan separation allows us to explore different platforms beyond Hadoop and Storm—possibilities include Spark [39] and Akka.[6]

[5]A trivial solution would be to retain a log of all events, but this is obviously not scalable.

7. RELATED WORK

Dataflow languages. Summingbird is a high-level dataflow language for analytical processing and thus it shares common features with other languages such as Pig, Cascading/Scalding, and DryadLINQ [38]. Their overall approach is to provide developers with high-level primitives and user-defined functions (UDFs) in a manner that is abstracted from the underlying execution engine. Summingbird is reminiscent of Sawzall [33], which supports arbitrary map-side procedural code but has only a small set of aggregators on the reduce side. However, Sawzall aggregators do not take explicit advantage of algebraic structures to help developers reason about correctness. In contrast to all these languages, which are suitable only for batch processing, Summingbird can target execution on Hadoop and Storm without any modifications to the program logic.

MapReduce extensions to online processing. Previous attempts to extend MapReduce to online processing include DEDUCE [20], the system of Grinev et al. [13], and Muppet [21]. These systems are similar to Summingbird programs running on Storm, but lack the seamless integration between batch and online processing that we provide. DEDUCE is an extension of IBM's System S stream processing middleware that provides support for MapReduce as embedded jobs within a larger dataflow, and hence has a different architecture than Summingbird. The system of Grinev et al. [13] builds on the distributed key–value store Cassandra and implements a reduce function that incrementally applies a new input value to an already existing aggregated value. This is similar to Summingbird on Storm without any batching, which as we have noted does not scale to Twitter's data volume. Furthermore, Grinev et al. describe aggregations without explicit references to algebraic structures, which is a critical component of Summingbird since it provides guarantees of correctness. Muppet [21] describes a modified MapReduce model called "MapUpdate": since streams may never end, "updaters" use storage called *slates* to summarize the data they have seen so far, serving as "memories" of updaters, distributed across multiple machines and persisted in a key–value store for later processing. This is quite similar to one way in which Summingbird programs can be executed, but we provide other processing options as well. Finally, despite its name, Hadoop Online Prototype (HOP) [8] is primarily about pipelining intermediate results between the map and reduce phases of a batch job; however, it does support continuous queries via batching on the reduce end.

Another related thread of research explores MapReduce extensions for *incremental* batch processing. Examples include the continuous bulk processing (CBP) model [26], Incoop [3], and Hourglass [14]. The focus of these systems is on techniques to minimize reprocessing of old data for reoccurring jobs on data that are being ingested continuously (e.g., log files). Summingbird's solution to the same class of problems is the hybrid processing mode described in Section 5.2, but in addition to efficient incremental batch aggregations

[6]`http://akka.io/`

(using Hadoop), we are able to provide up-to-date results with minimal latency (using Storm).

Other online and stream processing systems. Online processing of data streams in real time is of course not a new problem. Below we discuss other systems for online and stream processing beyond MapReduce extensions.

Stream-oriented databases have a long history [7, 12, 11, 19]. Typically, users issue standing queries in a variant of SQL with temporal extensions and results are returned via some sort of callback. One advantage of these systems is that they build on widespread familiarity with SQL. In addition, these systems usually have built-in primitives representing various temporal constructs such as sliding windows, which makes large classes of queries easy to write (e.g., counting clicks and clickthrough frequencies). In the last few years, stream processing engines have received renewed interest. In addition to Storm, other systems in this space include S4 [31], Samza,[7] BlockMon [36], TimeStream [34], Spark Streaming [40], MillWheel [1], and Photon [2]. Summingbird is different from these systems in its integration of online and batch processing, its dependence on algebraic structures to allow developers to reason about the correctness of computations, and its use of approximate data structures.

Online data processing frameworks often have close relationships to publish-subscribe systems such as Hedwig[8] or Kafka [18] and queuing systems such as RabbitMQ, Amazon's Simple Queue Service,[9] or Twitter's Kestrel.[10] These systems often serve as sources from which online data processing systems consume data—for example, Summingbird is able to read messages from Kestrel queues and from Kafka. These systems, however, focus on issues related to the physical transport and delivery of messages, as opposed to how the messages are processed by consumers.

Category theory and data analytics. We have seen a number of researchers explore the relationship between algebraic structures, category theory, and large-scale data analytics. For example, Meijer and Bierman [27] showed that a single generalization of relational algebra over sets—namely, monads and monad comprehensions—forms the basis of a common query language for both SQL and noSQL. In the machine learning domain, Izbicki [17] described how online training, parallel training, and cross-validation of classifiers can be understood in terms of monoids and homomorphisms. From the distributed systems literature, work on Convergent and Commutative Replicated Data Types (CRDTs) [35], which are distributed data structures that are guaranteed to eventually converge on the same state under asynchronous replication, are enabled by certain algebraic structures. The application of CRDTs to stream processing has been illustrated in a prototype [30]. Like Summingbird, these works share in the insight that abstract algebra provides a formal framework for thinking about and potentially resolving many thorny issues in distributed processing.

8. CONCLUSIONS

The nature of analytics within Twitter has evolved over the past few years, perhaps leading trends in the broader community. This paper focuses on a specific pain point that arose from the need for both batch and online analytics, which previously resulted in duplicate and difficult-to-maintain code. We describe efforts to address this challenge and reflect on the impact within the organization. Early on, we also made the decision to open source Summingbird so that the community can benefit from our experiences and build on our efforts. In terms of exploiting the formal properties of algebraic structures to seamlessly integrate different modes of distributed processing, we have only begun to scratch the surface and hope to stimulate further work.

[7]`http://samza.incubator.apache.org/`

[8]`http://wiki.apache.org/hadoop/HedWig`

[9]`http://aws.amazon.com/sqs/`

[10]`https://github.com/twitter/kestrel`

9. ACKNOWLEDGMENTS

We'd like to thank everyone at Twitter who has contributed to Summingbird, particularly, Ashutosh Singhal, Wen-Hao Lue, Bill Darrow, Joe Nievelt, Brian Wallerstein, Dmitriy Ryaboy, and Alex Roetter.

10. REFERENCES

[1] 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. *VLDB*, 2013.

[2] 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. *SIGMOD*, 2013.

[3] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquini. Incoop: MapReduce for incremental computations. *SoCC*, 2011.

[4] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[5] L. Bottou. Large-scale machine learning with stochastic gradient descent. *COMPSTAT*, 2010.

[6] A. Z. Broder. On the resemblance and containment of documents. *SEQUENCES*, 1997.

[7] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams—a new class of data management applications. *VLDB*, 2002.

[8] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce online. *NSDI*, 2010.

[9] G. Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[10] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. *Conference on Analysis of Algorithms*, 2007.

[11] B. Gedik, H. Andrade, K.-L. Wu, P. Yu, and M. Doo. SPADE: The System S declarative stream processing engine. *SIGMOD*, 2008.

[12] J. Gehrke. Special issue on data stream processing. *Bulletin of the Technical Committee on Data Engineering*, 26(1):2, 2003.

[13] M. Grinev, M. Grineva, M. Hentschel, and D. Kossmann. Analytics for the real-time web. *VLDB*, 2011.

[14] M. Hayes and S. Shah. Hourglass: A library for incremental processing on Hadoop. *Big Data*, 2013.

[15] S. Heule, M. Nunkesser, and A. Hall. HyperLogLog in practice: Algorithmic engineering of a state of the art cardinality estimation algorithm. *EDBT*, 2013.

[16] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. *NSDI*, 2011.

[17] M. Izbicki. Algebraic classifiers: A generic approach to fast cross-validation, online training, and parallel training. *ICML*, 2013.

[18] J. Kreps, N. Narkhede, and J. Rao. Kafka: A distributed messaging system for log processing. *NetDB*, 2011.

[19] S. Krishnamurthy, M. Franklin, J. Davis, D. Farina, P. Golovko, A. Li, and N. Thombre. Continuous analytics over discontinuous streams. *SIGMOD*, 2010.

[20] V. Kumar, H. Andrade, B. Gedik, and K.-L. Wu. DEDUCE: At the intersection of MapReduce and stream processing. *EDBT*, 2010.

[21] W. Lam, L. Liu, S. Prasad, A. Rajaraman, Z. Vacheri, and A. Doan. Muppet: MapReduce-style processing of fast data. *VLDB*, 2012.

[22] B. Lampson. Hints for computer system design. *SOSP*, 1983.

[23] G. Lee, J. Lin, C. Liu, A. Lorek, and D. Ryaboy. The unified logging infrastructure for data analytics at Twitter. *VLDB*, 2012.

[24] J. Lin and A. Kolcz. Large-scale machine learning at Twitter. *SIGMOD*, 2012.

[25] J. Lin and D. Ryaboy. Scaling big data mining infrastructure: The Twitter experience. *SIGKDD Explorations*, 14(2):6–19, 2012.

[26] D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum. Stateful bulk processing for incremental analytics. *SoCC*, 2010.

[27] E. Meijer and G. Bierman. A co-relational model of data for large shared data banks. *Communications of the ACM*, 54(4):49–58, 2011.

[28] G. Mishne, J. Dalton, Z. Li, A. Sharma, and J. Lin. Fast data in the era of big data: Twitter's real-time related query suggestion architecture. *SIGMOD*, 2013.

[29] S. A. Myers, A. Sharma, P. Gupta, and J. Lin. Information network or social network? The structure of the Twitter follow graph. *WWW Companion*, 2014.

[30] D. Navalho, S. Duarte, N. Preguiça, and M. Shapiro. Incremental stream processing using computational conflict-free replicated data types. *CloudDP*, 2013.

[31] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. *KDCloud Workshop at ICDM*, 2010.

[32] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. *SIGMOD*, 2008.

[33] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming Journal*, 13(4):277–298, 2005.

[34] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang. TimeStream: Reliable stream computation in the cloud. *EuroSys*, 2013.

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

[36] D. Simoncelli, M. Dusi, F. Gringoli, and S. Niccolini. Stream-monitoring with BlockMon: Convergence of network measurements and data analytics platforms. *ACM SIGCOMM Computer Communication Review*, 43(2):30–35, 2013.

[37] 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. *SIGMOD*, 2014.

[38] 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. *OSDI*, 2008.

[39] 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. *NSDI*, 2012.

[40] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. *HotCloud*, 2012.