Robust, Scalable, Real-Time Event Time Series Aggregation at Twitter

Peilin Yang,[1] Srikanth Thiagarajan,[1] and Jimmy Lin[2]
[1] Twitter Inc.
[2] David R. Cheriton School of Computer Science, University of Waterloo
{peiliny,srikantht}@twitter.com,jimmylin@uwaterloo.ca

ABSTRACT

Twitter's data engineering team is faced with the challenge of processing billions of events every day in batch and in real time, and we have built various tools to meet these demands. In this paper, we describe TSAR (TimeSeries AggregatoR), a robust, scalable, real-time event time series aggregation framework built primarily for engagement monitoring: aggregating interactions with Tweets, segmented along a multitude of dimensions such as device, engagement type, etc. TSAR is built on top of Summingbird, an open-source framework for integrating batch and online MapReduce computations, and removes much of the tedium associated with building end-to-end aggregation pipelines—from the ingestion and processing of events to the publication of results in heterogeneous datastores. Clients are provided a query interface that powers dashboards and supports downstream *ad hoc* analytics.

ACM Reference Format:
Peilin Yang, Srikanth Thiagarajan, and Jimmy Lin. 2018. Robust, Scalable, Real-Time Event Time Series Aggregation at Twitter. In *SIGMOD'18: 2018 International Conference on Management of Data, June 10–15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3183713.3190663

1 INTRODUCTION

The analytical needs of data-driven organizations have been evolving to increasingly demand insights generated in real time. Internet companies wish to adapt machine-learned prediction models in response to changing user behavior, public safety officials want to respond to incidents without delay, and financial institutions seek to stop fraud as it's being committed—just to give a few examples. However, organizations desire real-time capabilities without sacrificing efficient and scalable retrospective analytics over large volumes of historic data, which data scientists have become accustomed to. Like many organizations that have built substantial batch analytics infrastructure for data science, Twitter has been evolving to meet increasing demands for real-time analytics.

In this paper, we describe the architecture and design of TSAR (TimeSeries AggregatoR), a robust, scalable, real-time event time series aggregation framework at Twitter designed for applications such as site traffic, service health, and user engagement monitoring. Specifically, we showcase the features of TSAR using a simplified example of the Twitter interaction counter service that computes time series of Tweet impressions and engagements both historically and in real time, segmented by a multitude of dimensions including users, geographic location, user demographics, device type, engagement type (like, reply, etc.), and many more. TSAR provides a high-level domain-specific language (DSL) to specify aggregations declaratively and automates much of the tedium (boilerplate generation, job coordination, error handling, etc.) necessary to build robust and scalable data processing pipelines that handle both historic and live data. TSAR, which has been in production since around 2013, provides an integrated query service that powers dashboards and supports downstream *ad hoc* analytics.

As with previous papers that Twitter has written about its analytics infrastructure—for example, event logging pipelines [12], machine learning tools [14], and recommendation services [6, 17]—the goal is to share our experiences with an academic audience. We offer a "real-word" production perspective that complements the existing literature on real-time processing and aggregation queries. Although aspects of our solution are constrained by existing infrastructure investments, the event aggregation problem we tackle is pervasive and we believe that our experiences are valuable for both researchers and practitioners.

2 LAMBDA AND KAPPA

The terms "Lambda Architecture" and "Kappa Architecture" have entered the vernacular as design patterns for real-time analytics [13]. Distilled to its essence, the Lambda Architecture, which traces back to 2011, consists of a batch processing layer (platform) and a transient real-time processing layer (platform), plus a merging layer on top.[1] In contrast, in the Kappa Architecture, everything is a stream (for example, physically stored in a messaging platform like Kafka), and therefore there is no distinction between batch and stream processing (e.g., batch processing is simply streaming through historic data).[2]

Although these terms capture the evolution of data processing architectures, we believe that these designations are unhelpful today because they primarily focus on physical execution. Early implementations of real-time event aggregation services at Twitter were built using the Lambda Architecture, with two separate code paths (in some cases, managed by different teams) leading to job execution on Hadoop MapReduce and Storm [19]. However, early

ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3190663

[1] http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html
[2] https://www.oreilly.com/ideas/questioning-the-lambda-architecture

on we realized that building, maintaining, and coordinating jobs in this manner was not sustainable, which led to the development of Summingbird [5], an open-source Scala DSL designed to integrate online and batch MapReduce computations in a single framework. Summingbird exploits certain algebraic structures to provide the theoretical foundation for seamlessly integrating online and batch processing. Programs are written using dataflow abstractions such as sources, sinks, and stores, and can run on different execution platforms: Hadoop MapReduce for batch processing and Storm [19] (later Heron [11]) for online processing. Lin [13] provides a recent discursive account of the evolution of this infrastructure.

Is Summingbird an instance of the Lambda or Kappa Architecture? It provides a unified model for expressing computations, so in this respect it is "Kappa-like"; yet Summingbird programs are executed (transparently) in different frameworks, making it "Lambda-like". As computing abstractions evolve, we believe that the Lambda and Kappa distinctions are increasingly irrelevant. Recent developments such as Apache Beam [3] support this argument, as it also provides a unified computational abstraction over what Akidau et al. call "bounded" and "unbounded" datasets. However, physical execution is handled by different "runners" that may perform batch or streaming computations.

At Twitter, Summingbird provides an expressive programming model for a broad range of analytics queries, but the amount of work required to build service infrastructure and end-to-end data processing pipelines remains substantial, especially those involving both online and batch computations. TSAR, which provides a set of abstractions on top of Summingbird, simplifies and automates much of these processes for event aggregations.

3 DESIGN CONSIDERATIONS

Using engagement monitoring as the canonical application of TSAR, let us begin by characterizing the scope of the problem: approximately half a billion Tweets are created every day, and these Tweets are viewed tens of billions of times. For the interaction counter application, we desire an *end-to-end* latency of ten seconds. Other than the obvious desiderata of robustness and scalability, there are a number of less obvious issues to consider:

- How do we coordinate schemas to keep all physical representations consistent? For example, data might be stored on HDFS for use by downstream analytics pipelines but replicated in heterogeneous datastores to support different applications.
- How do we provide support for flexible schema evolution? Aggregation dimensions, business rules, and data cleaning processes change over time.
- How do we painlessly update historical data? Common cases include updates to existing queries or new queries, for which we need to backfill historic data.

TSAR adopts the following design principles:

- Hybrid computation. Every event is processed twice—first in real time and then again (at a later time) by a batch job. Since all processing is orchestrated by Summingbird, there is a single code base. This hybrid model confers all the advantages of batch (efficiency, determinism) and streaming (low latency). However, see Section 5 for more discussion.

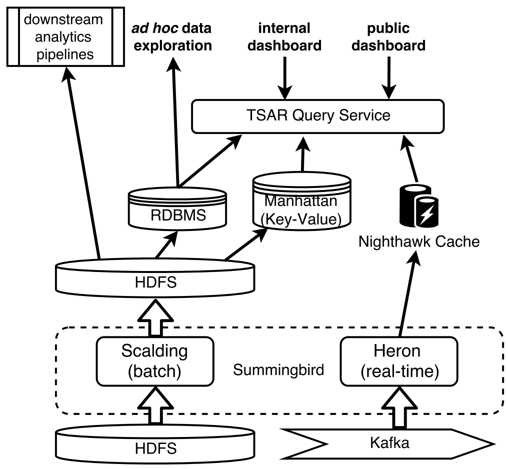

Figure 1: Overall architecture of TSAR.

- Separation of event production from event aggregation. Event production concerns the generation of events to be aggregated from raw sources (e.g., log data on HDFS or a Kafka topic). This is specified imperatively to support data cleaning and custom business logic. Event aggregation is specified declaratively.
- Unified schema architecture. The schema for a TSAR service is specified in a datastore-independent way. TSAR automatically maps the schema onto diverse datastores and transforms the data as necessary when the schema evolves.
- Integrated service infrastructure. TSAR integrates with other Twitter services that provide data processing, storage, scheduling, observability, alerting, etc. The tedium of configuring and orchestrating different components is largely automated.

4 TSAR

At a high-level, TSAR is a robust and scalable framework built on top of Summingbird for specifying event aggregations to compute time series. It abstracts the most common design patterns for computing event aggregations, exposes them in a simple Scala DSL, and automates associated tooling and service infrastructure, which includes boilerplate generation, job scheduling, error handling, etc. TSAR provides a structured representation of time series data that supports data replication to a heterogeneous mix of datastores, e.g., HDFS for long-term persistence and downstream processing, RDBMSes or key–value stores for dashboard visibility and data exploration. The overall architecture is shown in Figure 1.

From the perspective of a developer, there are two points of contact with TSAR:

- The developer expresses the production and aggregation of events in a Scala DSL, specifies the event schema, and defines associated configuration data. These elements comprise a TSAR job.
- TSAR provides a query service for consuming the event time series data (e.g., feeding frontend dashboards).

TSAR automates much of the "plumbing" associated with building end-to-end data pipelines. The batch processing pipeline is shown on the left side of Figure 1 and uses Hadoop MapReduce (via Scalding, Twitter's high-level Scala API). The preprocessing

```
aggregate {
  onKeys(
    (clientApplicationId, engagementType),
    (clientApplicationId),
    (engagementType)
  )
  produce(Count, Unique(userId))
  sinkTo(Manhattan, NightHawk)
  fromProducer(
    Source.map {
        (e.timestamp,
         EngagementAttributes(
           Some(clientApplicationId),
           Some(engagementType),
           Some(userId)
        )
      }
  )
}
```

Figure 2: An example TSAR aggregation query.

```
struct EngagementAttributes {
  1: optional i64 client_application_id,
  2: optional EngagementType engagement_type,
  3: optional i64 user_id
}
```

Figure 3: An example Thrift definition of an event over which aggregation is performed.

stage ingests source events (typically, raw Thrift-encoded service logs), performs job-specific processing (which might include joining together event streams or applying custom transformations), and emits another stream of TSAR events, which are constrained to be tuples of elementary attributes that conform to a fixed, job-specific Thrift schema. These events are then aggregated along various combinations of attribute dimensions (as specified by the job) and the aggregates are written to HDFS in a long-lived, reusable intermediate representation. The intermediates are reprocessed, aggregated further, and replicated to multiple datastores. The online processing pipeline, shown on the right side of Figure 1, largely follows the same execution pattern, defined by the same job, except with Heron as the execution engine (bypassing HDFS). An integrated query service presents a uniform interface to access both historic as well as up-to-date values.

4.1 Job Specification

As a concrete example, let us walk through a hypothetical TSAR job called `Engagement` that counts engagement events (likes, replies, etc.) by client application. The query (slightly simplified) is shown in Figure 2.

As discussed in Section 3, one key design principle is the separation of how events are produced (which is imperative) with how they are aggregated (which is declarative).

The `fromProducer` block contains imperative code to convert raw data (e.g., service logs) into a stream of events, always in the form of `(time, event)` tuples. The event itself, in this example called `EngagementAttributes`, is defined as a Thrift struct, shown in Figure 3. Note that all fields must be declared optional to cope with incomplete records in the raw source. In this case we are simply mapping over a source to generate `(time, event)` tuples, but in reality the event-production logic can include any transformations (e.g., `map`, `flatMap`, `filter`, etc.) supported by Summingbird, including joins based on lookup into external sources. This flexibility allows the developer to implement data cleaning algorithms and apply custom business logic. The translation of field names between the TSAR query and the Thrift definition is handled automatically (simply a matter of mapping between "snake case" and "camel case" naming conventions).

The aggregation itself is specified declaratively. The `onKeys` block declares the tuples over which we are aggregating, which must be subsets of fields in the event Thrift struct (Figure 3). In this case, we compute three types of aggregations. The `produce` block specifies what to compute: in this case, the total count of events in each aggregation bucket and the number of unique users in that bucket. Currently, the supported metrics include count, unique count, sum, mean, variance, standard deviation, max, and min.

The final destination of the results are specified in the `sinkTo` block. Options include:

- Manhattan, Twitter's custom key–value store.[3] Two key features that are relevant to TSAR include tight integration with Hadoop to enable batch data imports and support for high-volume time series counters as an optimized use case.
- Nighthawk, Twitter's sharded Redis cache.
- Other RDBMSes such as MySQL and Vertica.

In our example, the output will be stored in both Manhattan and Nighthawk. TSAR is capable of writing to multiple sinks.

The final component of a TSAR job is the configuration data. Here, the developer specifies the granularity of aggregation in arbitrarily multiples of minutes, hours, and days (including multiple granularities simultaneously). The configuration also specifies bindings for abstractions in the query: for example, HDFS path of the raw log data in the case of batch processing, coordinates to the correct Kafka topic in the case of real-time processing. Similarly, the developer must also specify the exact coordinates of the data sink, e.g., which datacenter, which cluster, etc.

4.2 Job Execution

A TSAR job comprises the query (Figure 2), the Thrift struct (Figure 3), and associated job configuration data. From there, the TSAR infrastructure manages all other aspects of execution.

When aggregations run in batch, typically over large volumes of log data stored on HDFS, the TSAR job translates into several individual batch jobs, since the developer usually applies a number of data transformations to clean the raw data and to encode business logic, specified imperatively in the `fromProducer` block of the TSAR query. These batch jobs are automatically defined, staged, and deployed without additional intervention. TSAR constructs a primary batch job that processes raw source data to materialize intermediate data and several smaller secondary batch jobs that ingest the intermediate data to compute the specified aggregates. These jobs implement simple optimizations that are well known in the

[3]https://blog.twitter.com/engineering/en_us/a/2014/manhattan-our-real-time-multi-tenant-distributed-database-for-twitter-scale.html

literature on cube materialization to reduce the number of passes required over the full dataset (i.e., rollup of individual dimensions into higher-level aggregates).

TSAR relies on Twitter's Manhattan key–value store to provide fast, scalable, and robust access for high-load dashboard applications, and is a frequent destination for the `sinkTo` block in a query. The output of batch jobs is first written to HDFS, and then bulk imported into Manhattan—this process is orchestrated by TSAR without requiring developer intervention. TSAR groups aggregate data into temporal buckets that are placed on cluster nodes to optimize performance around the most common use case: a query for all data points falling within an interval that is approximately 100 times the aggregation granularity of the time series.

Output to relational databases is another important use case for TSAR. For example, data stored in Vertica provide vehicles for self-serviceable interactive exploration by data scientists. In some cases, MySQL provides a suitable backend for rapid development of internal dashboard applications. To populate relational databases, TSAR uses Crane, a tool developed by the Analytics Infrastructure team at Twitter designed specifically to move data between datastores. TSAR exports data to an RDBMS via a batch job comprising two steps: the first step converts the developer-defined Thrift struct into a format suitable for bulk loading into the RDBMS, and then in the second step the data are ingested into the RDBMS atomically. Once again, data movement is orchestrated by TSAR and does not require developer intervention.

Referring back to design principles in Section 3, the key point here is that TSAR provides a unified schema architecture (starting with a developer-defined Thrift struct such as the one in Figure 3) and transparently handles the mapping to different datastores such that aggregate data can be seamlessly replicated and accessed. For schema evolution, new attributes can be specified simply by adding new optional fields to the Thrift struct, although with a few caveats: existing attributes cannot be deleted or renamed, and new attributes must be appended to the end of the schema to maintain backwards compatibility. New aggregation templates and new metrics can be added at any time. Upon redeployment of the TSAR job, the framework will begin computing new values, but the developer must explicitly trigger backfilling on historic data if desired.

Online execution of a TSAR job is quite similar to batch execution, except that Summingbird bypasses HDFS; instead, Heron [11] consumes Kafka queues to compute the real-time aggregations. To support robust, high-throughput computations, Summingbird pre-partitions the aggregation key space (at compile time) across nodes in the Heron topology and pre-aggregates large cardinality keys in the "map" phase (in essence, acting as combiners). Typically, real-time results are written to Nighthawk, which is Twitter's caching service. Results from online execution of a TSAR job are not persisted to HDFS.

4.3 Data Consumption

Event time series computed by TSAR can be consumed in a number of ways. The results of batch computations, typically over long periods of time, are persisted on HDFS and can feed additional downstream analytics jobs—for example, event counts can serve as input training data to machine-learned models. Another frequent use case is interactive data exploration by data scientists over time series data that have been imported into an RDBMS.

Another common use case for TSAR is to power high-request, customer-facing dashboards with strict SLAs, for example, serving advertisers. These typically allow users to examine both historic and live data, which requires the integration of results from batch and online aggregations. For this, TSAR automatically builds a Thrift query service for each TSAR job. This one-service-per-job design allows independent capacity allocation and minimizes incident spillover effects. The basic interface takes two structured parameters, a query key specifying the event to be queried and the time range. By default, the service automatically queries both the online and batch results (since TSAR knows exactly where they are stored). The results from both sources are merged, and in cases with overlap in coverage, the online results are discarded. Part of the TSAR configuration includes parameters to control overlap between batch and online results, allowing the developer to control a tradeoff between robustness and resource consumption.

5 RELATED WORK

The "process every event twice" design of TSAR is worth discussing, since it appears antithetical to the design of more recent platforms for integrating batch and stream processing such as Spark (with Spark Streaming [21] and Spark Structured Streaming), Apache Beam [3] (the open-source API to Google Cloud Dataflow [2]), or Kafka Streams. One important consideration in maintaining distinct batch and online execution frameworks is the existing investment Twitter had already made in building infrastructure around Hadoop and in developing Storm and later Heron. Furthermore, the TSAR architecture dates back to 2013, before modern approaches were available or were sufficiently mature for our processing volume. Given a clean-slate redesign, these alternative approaches might have been viable, but organizations are frequently restricted by path dependencies imposed by previous design choices.

From a technical perspective, though, a "process twice" design allows us to sidestep the many complexities in a streaming-only design. For example, Apache Beam needs to keep track of event time and processing time and has quite elaborate built-in mechanisms for managing incremental computations (e.g., specifying whether previous results are retracted or only deltas are propagated). This yields complex expressions such as the following:[4]

```
.triggering(
  AtWatermark()
    .withEarlyFirings(AtPeriod(Duration.standardMinutes(1)))
     .withLateFirings(AtCount(1)))
.accumulatingAndRetractingFiredPanes()
```

And of course, the underlying execution engine must support the associated semantics of such expressions. To accomplish this, it is inevitable that the execution engine must track mutable state. A similar notion in Kafka Streams, the `KTable`, can be interpreted as a stream of updates. Mutability is hard [8].

These issues do not exist in TSAR because the batch jobs can be orchestrated such that late-arriving events are not an issue, based on the service-level agreements of the underlying data sources.

[4]https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-102

For example, if the SLA of the upstream process guarantees that all messages will be delivered within ten minutes after the event time, then we can simply offset the batch processing start time and ensure that all output will be correct. Inputs and results are both immutable. In the language of Apache Beam, results are *only* computed at the watermark, and thus all the execution machinery related to early and late firings becomes completely unnecessary. While it is true that all events are processed twice, the online and batch execution frameworks can be optimized for specific data processing patterns, as opposed to a single execution engine with a more complex design.

In general, a batch execution engine will be faster and more efficient than a streaming execution engine of comparable implementation quality and effort, by the virtue that batch processing is amenable to more optimizations. This performance difference will be magnified as the amount of data increases, which is important in large backfill jobs where a new query needs to be run on petabytes of retrospective data. Furthermore, given the "backup" provided by the batch processing system when processing everything twice, the online system can be more cavalier in error recovery, load shedding, applying backpressure, etc., yielding simpler designs. While it is certainly true that modern enterprise data architectures are moving in the direction of Kafka and "everything is a stream" processing, and such systems have matured significantly over the last few years, it is not a foregone conclusion that "one processing engine to rule them all" is superior in all respects.

Notwithstanding this architectural discussion, additional lessons from TSAR come from elsewhere. Even if we were to replace the execution engine in TSAR with an alternative framework, the only thing that would change is the box marked Summingbird in Figure 1. Critically, much of the service infrastructure and tooling remains necessary: job scheduling and coordination, error handling, replication to multiple datastores to support different application scenarios, etc. This is perhaps a lesson that the academic community is beginning to internalize: in a production environment, the elegance of a computational model meets the harsh reality of needing to interface with a multitude of heterogeneous, unreliable, and often legacy components in end-to-end pipelines. Frequently, it is the "plumbing" that consumes the efforts of infrastructure engineers. Perhaps the biggest contribution of TSAR is the tooling that automates a significant amount of tedium (and articulation of what those issues are) to provide an easy-to-use end-to-end service.

There is a large body of work on efficiently computing aggregations both in the batch setting (i.e., cube materialization [1, 7, 15, 16]) as well as the online setting [4, 9, 10, 18, 20]. This work, however, is largely orthogonal to TSAR since it focuses on a much higher level of abstraction, leaving the execution to Summingbird, and ultimately Hadoop MapReduce or Heron. Since aggregations in TSAR are declaratively specified, all relevant optimizations can be implemented in these underlying frameworks.

6 CONCLUSIONS

Building robust and scalable data pipelines is hard. Summingbird addresses the physical execution of online and batch MapReduce computations under a common programming model, but that is only one (albeit important) aspect of an end-to-end solution. For a large class of aggregation queries, TSAR takes the next step of automating the tedium involved in orchestrating hybrid data pipelines, making developers' lives a bit easier.

ACKNOWLEDGMENTS

We would like to thank people who have contributed to TSAR: Pankaj Gupta, Timur Abishev, Ben Pence, Aaron Siegel, Reza Lotun, Ryan King, Dave Loftesness, Dmitriy Ryaboy, Andrew Nguyen, Jeff Sarnat, John Chee, Eric Conlon, Allen Chen, Gabriel Gonzalez, Ali Alzabarah, Zongheng Yang, Zhilan Zweiger, Kevin Lingerfelt, Justin Chen, Ian Chan, Jialie Hu, Max Hansmire, David Marwick, Oscar Boykin, Sam Ritchie, Ian O'Connell.

REFERENCES

[1] Sameet Agarwal, Rakesh Agrawal, Prasad Deshpande, Ashish Gupta, Jeffrey F. Naughton, Raghu Ramakrishnan, and Sunita Sarawagi. 1996. On the Computation of Multidimensional Aggregates. In *VLDB*. 506–521.

[2] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. 2013. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. *PVLDB* 6, 11 (2013), 1033–1044.

[3] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. 2015. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *PVLDB* 8, 12 (2015), 1792–1803.

[4] Arvind Arasu and Jennifer Widom. 2004. Resource Sharing in Continuous Sliding-Window Aggregates. In *VLDB*. 336–347.

[5] Oscar Boykin, Sam Ritchie, Ian O'Connell, and Jimmy Lin. 2014. Summingbird: A Framework for Integrating Batch and Online MapReduce Computations. *PVLDB* 7, 13 (2014), 1481–1484.

[6] Pankaj Gupta, Ashish Goel, Jimmy Lin, Aneesh Sharma, Dong Wang, and Reza Zadeh. 2013. WTF: The Who to Follow Service at Twitter. In *WWW*. 505–514.

[7] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. 1996. Implementing Data Cubes Efficiently. In *SIGMOD*. 205–216.

[8] Pat Helland. 2015. Immutability Changes Everything. In *CIDR*.

[9] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. 1997. Online Aggregation. In *SIGMOD*. 171–182.

[10] Sailesh Krishnamurthy, Chung Wu, and Michael Franklin. 2006. On-the-Fly Sharing for Streamed Aggregation. In *SIGMOD*. 623–634.

[11] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. 2015. Twitter Heron: Stream Processing at Scale. In *SIGMOD*. 239–250.

[12] George Lee, Jimmy Lin, Chuang Liu, Andrew Lorek, and Dmitriy Ryaboy. 2012. The Unified Logging Infrastructure for Data Analytics at Twitter. *PVLDB* 5, 12 (2012), 1771–1780.

[13] Jimmy Lin. 2017. The Lambda and the Kappa. *IEEE Internet Computing* 21, 5 (2017), 60–66.

[14] Jimmy Lin and Dmitriy Ryaboy. 2012. Scaling Big Data Mining Infrastructure: The Twitter Experience. *SIGKDD Explorations* 14, 2 (2012), 6–19.

[15] Arnab Nandi, Cong Yu, Phil Bohannon, and Raghu Ramakrishnan. 2011. Distributed Cube Materialization on Holistic Measures. In *ICDE*. 183–194.

[16] Kenneth A. Ross and Divesh Srivastava. 1997. Fast Computation of Sparse Datacubes. In *VLDB*. 116–125.

[17] Aneesh Sharma, Jerry Jiang, Praveen Bommannavar, Brian Larson, and Jimmy Lin. 2016. GraphJet: Real-Time Content Recommendations at Twitter. *PVLDB* 9, 13 (2016), 1281–1292.

[18] Kanat Tangwongsan, Martin Hirzel, Scott Schneider, and Kun-Lung Wu. 2015. General Incremental Sliding-Window Aggregation. *PVLDB* 8, 7 (2015), 702–713.

[19] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. 2014. Storm @Twitter. In *SIGMOD*. 147–156.

[20] Jun Yang and Jennifer Widom. 2003. Incremental Computation and Maintenance of Temporal Aggregates. *The VLDB Journal* 12, 3 (2003), 262–283.

[21] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *SOSP*. 423–438.