

Understanding and Building Fault-Tolerant, Scalable and Low-Latency Event Stream Analytics Solutions With Correctness Guarantees

BACHELOR'S THESIS / T3300

for the study program
Computer Science

at the
Baden-Wuerttemberg Cooperative State University Stuttgart

by
Dominik Stiller

Submission Date	September 7, 2020
Project Period	12 Weeks
Company	DXC Technology
Corporate Supervisor	Dipl.Ing. Bernd Gloss
University Supervisor	Prof. Dr. Dirk Reichardt
Matriculation Number, Course	4369179, TINF17A

Declaration of Authorship

I hereby declare that the thesis submitted with the title *Scalable and Reliable Complex Event Processing on Event Streams* is my own unaided work. All direct or indirect sources used are acknowledged as references.

Neither this nor a similar work has been presented to an examination committee or published.

Sindelfingen September 7, 2020

Place Date Dominik Stiller

Confidentiality Clause

This thesis contains confidential data of *DXC Technology*. This work may only be made available to the first and second reviewers and authorized members of the board of examiners. Any publication and duplication of this thesis—even in part—is prohibited.

An inspection of this work by third parties requires the expressed permission of the author, the project supervisor, and *DXC Technology*.

Abstract

Real-time computer vision applications with deep learning-based inference require hardware-specific optimization to meet stringent performance requirements. Frameworks have been developed to generate the optimal low-level implementation for a certain target device based on a high-level input model using machine learning in a process called autotuning. However, current implementations suffer from inherent resource utilization inefficiency and bad scalability which prohibits large-scale use.

In this paper, we develop a load-aware scheduler which enables large-scale autotuning. The scheduler controls multiple, parallel autotuning jobs on shared resources such as CPUs and GPUs by interleaving computations, which minimizes resource idle time and job interference. The scheduler is a key component in our proposed Autotuning as a Service reference architecture to democratize autotuning. Our evaluation shows good results for the resulting inference performance and resource efficiency.

Contents

Acronyms	V
List of Figures	VI
List of Tables	VII
List of Source Codes	VIII
1 Introduction	1
1.1 Problem	1
1.2 Scope	1
2 Background	2
2.1 Batch Processing	2
2.2 Stream Processing	4
2.3 Stream Transport	10
2.4 Event Processing	10
3 Design Considerations	12
3.1 Stream Transport Platforms	12
3.2 Stream Processing Platforms	12
4 Solution Design and Implementation	15
4.1 Design	15
4.2 Deployment	16
5 Analytics Usecase	17
5.1 HSL API Data	17
5.2 Analytics	17
5.3 Data Flow Example	18
6 Evaluation	19
6.1 Methodology	19
6.2 Results	19
6.3 Discussion	20
7 Conclusion	21
Bibliography	22
Glossary	24

Acronyms

HDFS Hadoop Distributed File System

List of Figures

1	Relationship between event time and processing time	6
2	Lambda architecture	7
3	Kappa architecture	7
4	Relationship between bounded and unbounded datasets	8

List of Tables

List of Source Codes

1 Introduction

the earlier insight arrive, the higher the value historically, data arrived in batches, maybe once per day this gave rise to batch processing systems, now highly performance optimized

1.1 Problem

lately, move from batch data to streaming data, since more data arrive continuously wide application range

processing is treating streaming data like batch data stream-native processing can improve correctness and stream-specific features (session windows)

non trivial because of time and State time because events arrive out of order, state to enable complex tasks (pattern recognition) still want to have correctness and faulttolerance at low latency especially challenging at large scale some existing platforms to solve problems

we want to understand stream-native processing platforms get hands on experience with usecase

1.2 Scope

goal:

- give an overview over correct, fault-tolerant, low-latency and scalable processing of streaming data
- demonstrate concepts through the design and implementation of an exemplary stream processing solution

2 Background

Understanding the challenges that are inherent to the building blocks of stream analytics is key to building a good solution. Therefore, this chapter provides background on characteristics and processing of event streams.

2.1 Batch Processing

For the better part of history, data was processed in form of batch datasets. An early analog example of batch data processing is the United States census: when the census was initiated in 1790, horseback riders recorded citizen counts per area and then transported their records to a central location for aggregation. While this is an extreme example, the principle still holds for digital data like periodic database dumps or bulk log transfers found in many batch processing systems today, where the the whole dataset is processed at once after arrival [1, p. 28].

A batch processing system takes a large amount of input data and runs a *job* to process it. The produced result are often analytics, but arbitrary applications like search index building and machine learning feature extraction can be built with this method. Since batch jobs usually take a while to execute, they are not interactive but scheduled to run periodically. For example, web server logs can be imported once per day from the web server nodes and then user behavior analytics are available on the next morning. While latency is high, throughput, i.e. the amount of data processed per second, is a key performance metric since data volume is usually very large [2, p. 390].

As the volume of data grew, dataset became too large to be handled by a single node (we use the term *node* to refer to an individual server in a cluster). This sparked the development of distributed processing engines like Hadoop [3] (based on the MapReduce [4] programming model) and Spark [5]. These frameworks tackle two common challenges of large-scale batch processing [2, p. 429], [6, pp. 362–373]:

- Scalability: support for distributed processing across nodes requires orchestration and *partitioning*, i.e. the division of the dataset into subsets that can be processed in parallel, possibly on different nodes
- Fault-tolerance: guarantee of consistent and correct results even in case of job failures caused, for example, by hardware failure or scheduler-induced preemption

Having a framework to handle these issues makes focusing on the actual problem much easier.

Distributed batch processing engines assume that all functions applied to the data are stateless (no intermediate results are stored) and have no externally visible side effects (e.g., database updates) [2, p. 430]. While these assumptions result in a deliberately restricted programming model, they facilitate distributed execution. Since no state needs to be shared between nodes, partition-based scalability is simple. In case of faults, the job can be restarted using the same input data, and the final output will be the same as if no faults had occurred (assuming deterministic operations). This is possible because input data are stored in a distributed and fault-tolerant file system like Hadoop Distributed File System (HDFS) [3]. Therefore, the underlying file system facilitates processing across multiple nodes. Some processing engines store intermediate results to speed up re-computations after failures, but this often requires tracking of data ancestry or checkpointing [2, p. 430].

Batch processing has been successfully applied at massive scales, with Hadoop clusters at Yahoo of 35,000 nodes being used to store 600 PB of data and run 34 million jobs every month [7]. However, it is only suitable for applications where low latencies are not required. Batch engines fall short when real-time processing is required, since they only process data once all input data are available. In practice, most data arrive as a continuous stream but are divided into batches of a certain size for batch processing [2, p. 439]. An obvious solution might be to decrease the batch size and run the job at a higher frequency, a technique known as *micro-batching*. This can decrease the latency to less than a second, but ultra-low latency applications are still infeasible with micro-batch processing [8]. This is especially true when considering that data might arrive with a delay, which usually requires deferred processing or re-processing when late data arrive. Also, jobs that might span batch bounds, such as user session analysis in web applications, are inherently complex to implement [6, pp. 34–35].

Apart from the technical shortcomings, processing a continuous stream of data in batches seems wrong from a philosophical point of view. Batch processing frameworks are fundamentally ill-suited for this type of data. Why not build processing engines specifically designed with continuous streaming data in mind, that can overcome and embrace stream characteristics to enable new types of applications?

This far-reaching sentiment was first expressed by Google researchers Tyler Akidau, Robert Bradshaw, Craig Chambers, *et al.* in 2015:

We propose that a fundamental shift of approach is necessary to deal with these evolved requirements in modern data processing. We as a field must stop trying to groom unbounded datasets into finite pools of information that eventually become complete, and instead live and breathe under the assumption that we will never know if or when we have seen all of our data, only that new data will arrive [and] old data may be retracted [9, p. 1792].

2.2 Stream Processing

Stream-native processing, as opposed to batch processing on streams, comes with many challenges, but is ultimately the more powerful approach when dealing with streaming data. This section is an introduction to streams and stream processing, showing the fundamental characteristics and challenges.

2.2.1 Streaming Data Properties

The terms “stream processing” has been assigned a variety of meanings. Many associate low-latency, approximate, or speculative results with stream processing systems, especially in comparison to batch processing systems [6, pp. 23–24]. While many historic systems had these properties, they are not inherent and should therefore not be used for definitions. Well-designed stream processing systems are perfectly capable of producing correct results. Therefore we use the definition of Akidau, Chernyak, and Lax:

[A stream processing system is] a type of data processing engine that is designed with infinite datasets in mind [6, p. 24].

Accordingly, a stream is an *unbounded* dataset that is infinite in size. Unboundedness means that a stream does not terminate and new data will arrive continuously. Therefore the dataset will never be complete. Many data sources found in the real world produce data naturally as unbounded stream: sensors measurements, stock updates, user activities, credit card transactions, retail purchases, public transportation updates and business activities come from processes that are theoretically infinite (or at least very long-running), so we have to assume that they do not end. This is in contrast to *bounded* datasets found in batch processing, which are regarded as complete.¹

¹This assumption can be made because there usually is a delay between data collection and data processing. Correct results can only be produced if this assumption holds and no data is late.

The reason for the prevalence of batch processing despite the stream nature of most data stems from historical technical limitations of data collection [1, p. 29]. Batch collection was the norm, be it for early census calculations or digital bulk dumps. Now we see a shift to more continuous data processing thanks to automation and digitization in the data collection process, which reduces latency but also requires new processing techniques. For the census example, this could mean to record births and deaths to produce continuous calculation counts.

Time Domains

A stream consist of records that usually contains information about an *event*, i.e. something of interest that happens in the real world. These might, for example, be purchases, website views, temperature changes or the arrival of a bus at a stop. When processing an event stream, two time domains are involved [6, p. 29]:

- Event time: the time at which the event actually occurred in the real world
- Processing time: the time at which events are observed in the processing system

These two time domains often do not coincide. The processing time can never be before the event time. However, the delay between the occurrence and processing of an event can be arbitrarily large. Usually, there is some small base delay due to, for example, network latencies and resource limitations. Other events might occasionally arrive later than expected, for example, when a vehicle broadcasting its position enters a tunnel or people using their phone sit in an airplane. In case historic data are processed, there might even be years of delay between event and processing time. Note that processing time is the natural order in which events arrive and are processed, processing by event time order requires additional effort.

The relationship of the two time domains can be visualized by plotting the progress of processing time over event time as shown in figure 1. Events (denoted by the diamonds) occur at event time and arrive at the system at processing time. The delay between these two is also known as *event-time skew* or *processing-time lag* (both terms are two perspectives on the same issue) [6, p. 30–31]. The event-time skew for the green event is shown by the arrow. Events on the diagonal line would have no event-time skew. This would mean that data are processed instantly after occurring, which simplifies processing because events would arrive at the system in event time order. In reality, events are always above this line due to the base delay. However, the delay is not constant. While events occur every 30s as shown in the top margin, some are observed much faster than others as shown in the right margin. In case of the green, blue and orange events, this even changes their order. This makes the stream (partially) unordered with respect to event

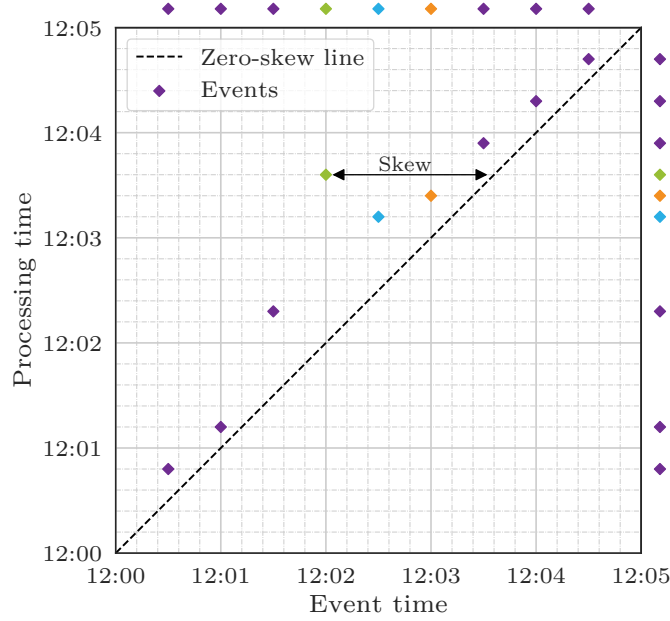


Figure 1: Relationship between event time and processing time: time skew varies a lot and leads to out-of-order arrival

time. Handling this skew and unordered is a key challenge stream processing frameworks have to solve [6, p. 28].

2.2.2 Stream Processing Architectures

The unbounded nature of streams, requiring continuous processing, cannot be handled by batch processing engines like Hadoop. While academic and commercial stream processing engines (SEEP, Naiad, Microsoft StreamInsight, IBM Streams) have existed before [10, p. 37], Apache Storm was the first one to find widespread adoption when it was released in 2011 [6, p. 375]. Like MapReduce, it solves many of the common challenges like fault-tolerance, networking and serialization and allows developers to focus on solving the actual problem [11].

While Storm excelled at providing low-latency results, it did so by sacrificing features like exactly-once processing required for guaranteed correctness. This sparked the development of the Lambda architecture [12], shown in figure 2. The batch layer produces correct results and handles fault-tolerance and scalability through the underlying processing engine, often Hadoop. Jobs are expressed in the MapReduce framework and store their results in a database optimized for batch writes and random reads for serving. The batch layer naturally lags behind real-time, therefore data is simultaneously processed in a real-time/speed layer, often implemented using Storm. The speed layer provides low-latency results but lacks in the correctness department due to approximative algorithms or possible

FIGURE OF LAMBDA ARCHITECTURE

Figure 2: Lambda architecture: the batch layer provides correct results, the speed layer provides low-latency results

FIGURE OF KAPPA ARCHITECTURE

Figure 3: Kappa architecture: a single processing engine provides correct, low-latency results

system faults. This is acceptable, however, since speed layer results are overwritten by correct batch layer results once available. Even if a speed layer job fails, batch layer results will be available at a later point. This requires the batch layer to store incoming data in an immutable and fault-tolerant way, also enabling recomputation in case processing code changes. By leveraging the two layers, the Lambda architecture provides low-latency, eventually-correct results [13, pp. 14–20, p. 27–28].

While the Lambda architecture has been used to build many successful systems, it is inherently complex. The processing logic needs to be implemented twice and in both cases specifically engineered towards the processing engine. Even if the logic is implemented in a higher-level API that can be compiled to MapReduce and stream processing jobs, the twofold operational effort remains [14].

The Lambda architecture was born out of necessity since no framework could guarantee both low latency and correctness. However, more and more modern stream processing frameworks are able to provide the batch layer’s correctness and the speed layer’s correctness in a single system, much simplifying development and operations. This is called the Kappa architecture, shown in figure 3. Instead of storing data on a distributed file system, the stream is often stored in a replayable stream transport platform like Apache Kafka. This enables fault tolerance and recomputation in case of processing logic changes [14].

Spark Streaming [15] was the first large-scale stream processing engine being suited for use in a Kappa architecture. While not a true streaming but rather micro-batch processing engine, the latency was low enough for most applications. Since micro-batching uses batch processing under the hood, consistency and correct results were guaranteed. However, Spark Streaming lacked support for event-time processing, therefore producing correct results only in case of in-order data or event-time-agnostic computations. Correctness is absolutely required for stream processing engines to achieve parity with batch processing engines. Tools for reasoning about time, and especially event time, are essential for dealing with unbounded streams [6, pp. 27–28]. Sophisticated time handling with high flexibility was explored in Google’s company-internal MillWheel [16] framework and Dataflow [9] processing models. Apache Flink [10] was the first open-source framework to incorporate

FIGURE OF BOUNDED AND UNBOUNDED DATASETS

Figure 4: Relationship between bounded and unbounded datasets: bounded datasets are sections of an unbounded dataset

the ideas into a high-throughput, low-latency stream processing engine that supports event-time processing and guarantees correctness.

Another contribution of Dataflow and Flink is the realization that batch and stream processing can be unified. Bounded batch datasets are effectively a section of an unbounded stream dataset, as shown in figure 4, and jobs can be specified using the same API and be executed on the same engine. However, bounded datasets are amenable to additional optimizations towards throughput at the cost of latency by increasing bundling sizes and computing processing stages successively instead of continuously [10, p. 35], [6, pp. 198–199]. Such a unified processing engine decreases development and operations cost since code and infrastructure can be shared, and allows to balance latency and throughput based on the application.

2.2.3 Processing Patterns

stream processing patterns [6, p. 35]:

Time-Agnostic very simple because no reasoning about time, only logic-based on single record, like filtering or inner join

Approximation complicated algorithms like streaming k-means, some with provable error bounds

Windowing chopping up stream into bounded datasets, but not necessarily with fixed bounds like in batch but allow arbitrary windows (like session)

windowing is what we call stream analytics relates multiple elements using time as ordering relation requires keeping state

2.2.4 Timely Processing

windowing as division of unbounded stream in bounded segments can be arbitrary, but usually time or count will focus on time, since count is effectively processing time fixed, sliding, session, fixed is special case of sliding window custom window using window assigner

watermark might be too fast or too short triggers define result materialization in processing time repeated or based on watermarks as measure of completeness can also have early and late

Repeated update triggers are great for use cases in which we simply want periodic updates to our results over time and are fine with those updates converging toward correctness with no clear indication of when correctness is achieved. [6, p. 63]

watermarks as heuristic, show different options difference between watermark and allowed lateness bounded out of orderness, ascending...

result refinement mode when having multiple triggers (fire and purge)

processing time is natural, event time requires special techniques analysis of data based on when they are observed as opposed to when they occur is usually not sufficient for correctness correct and latency are balancing with cost (more resources for fast and correct results) need to define measure of completeness to maximize correctness different methods for handling late data: retract old results, separate output, dismiss tradeoff completeness vs latency

based on processing time: simple and perfect measure of completeness, applicable in many cases where observation time is desired

based on event time: required when event time is desired, requires more buffering than processing time, usually no perfect measure of completeness, therefore based on heuristic

2.2.5 Stateful Processing

[6, Chapter 7] Many applications, especially as complex ones as analytics require state e.g. partial matches or intermediate results of aggregations

for batch: assumed that job can be restarted completely when it fails for streaming: assume that data might not be replayable from beginning, correctness and efficiency require persistent state

exactly once guarantees for correctness, requires offset and replayable source (at least data since last checkpoint) requires explicit state which is known to cluster and can be checkpointed exactly once especially important when side effects are non-idempotent [6, Chapter 5]

2.2.6 Stream–Table Duality

streams are data in motion table are data at rest can be converted [6, pp. 174–212]

2.3 Stream Transport

message queue, often ephemeral plain socket stream pubsub

2.3.1 Immutable logs

append-only immutable log with persistency

Reasons [1, p. 31]

- flexible consumers, also for debugging
- ordering
- Buffering and isolation, e.g. for backpressure handling and replay on node failure, important prerequisite for robustness and correctness

2.4 Event Processing

not based on data shape/cardinality like batch or stream rather data element type however, often streams of events

event types and definitions event type vs event instance

2.4.1 Event Driven Architecture

event happens in an instant complex events are multiple events in correlated according to a pattern (have a duration) composite event would be more fitting, but complex event is prevailing term

event driven types event notification event sourcing event-carried state transfer

geoevents

2.4.2 Pattern Recognition

Also called Complex Event Processing, but ambiguous pattern recognition performed on event streams seit sql:2016 auch iso standard not bound to stream processing, also e.g. microservices

selection of events to evaluate by window or consumption mode

3 Design Considerations

3.1 Stream Transport Platforms

rabbitmq

activemq

kafka

3.1.1 Apache Kafka

present Kafka

commercial distributions like confluent provide tiered retention

3.2 Stream Processing Platforms

storm

spark streaming

spark structured streaming

spark has good ML libraries

wso2

esper

comparison of frameworks: <https://youtu.be/PiEQR9AXgl4>

[17] shows performance

add feature list

mention apache beam as higher level unified API running on top of these platforms
implementation of dataflow model

3.2.1 Apache Flink

APIs

datastream, dataset, SQL

async queries

event time

unit testing

watermarking strategies

Cluster

workers and masters

task slots

high availability

Execution Model

tasks

operators

parallelism

co-location and operator chaining

shuffling after keyby

watermark propagation

backpressure sampling

code evolution, switch live to newer version, recomputation only possible if data are retained, but same with batch

State

state backends

broadcast state

Checkpointing

barriers

aligned and unaligned

Network Stack

backpressure handling

flow control

latency vs throughput

<https://flink.apache.org/2019/06/05/flink-network-stack.html>

Flink + Kafka

replay

partitioning

high availability

also managed versions on AWS, but set up ourselves to understand better

4 Solution Design and Implementation

aspects: correct, fault-tolerant, low-latency and scalable

4.1 Design

4.1.1 Architecture

separate clusters for ingest, streaming, processing and ui

decoupling of ingestion and processing with persistent event log in between has benefits

- handle backpressure without data loss
- decouple ingest and processing -> other processing possible
- replay in case of failure because not ephemeral

4.1.2 Event Schema

common schema, serialized as protobuf for strong typing but still allow flexible payload with any

show all definitions in appendix

for larger cases, should use central schema registry like supported by confluent

4.1.3 Ingestion

Extensible design with ingestors and processors

write to ingress topics

4.1.4 Flink Jobs

describe common functions (key selectors)

write to job topic

job design considerations:

- large sliding window with short period requires lots of memory
- High allowed lateness increases time until records can be garbage collected
- Accumulation functions only need to store a single value instead of all like in process function (aggregate early)
- only send relevant data to downstream tasks since data needs to be serialized, transferred and duplicates (for windows and CEP)
- state size influences checkpointing time
- watermarking and late data based on statistics (expected delay)
- checkpointing frequency
- retention period
- parallelism vs core/n_workers
- watermark frequency changes computation effort
- variable reuse if sequential (e.g. reuse output tuples instead of creating them for every record. what about downstream ops?)

4.2 Deployment

4.2.1 Infrastructure Considerations

capacity planning: <https://www.ververica.com/blog/how-to-size-your-apache-flink-cluster-general-guidelines>

immutable infrastructure (keep short)

infrastructure as code

5 Analytics Usecase

Looked for interesting use case which we can use to experiment with stream processing

Wanted to use real data

5.1 HSL API Data

Available data

statistics

5.2 Analytics

wanted to have analytics with challenges in different areas: pattern recognition, external queries

5.2.1 Geoaggregation

Division in cells

enables aggregation

provides way to reduce complexity with configurable resolution

late data handling

watermark bounded out of orderness time vs allowed lateness is a tradeoff between latency and recomputation effort

if only interested in latest window results: allowed lateness = window evaluation time

late side output if fine-grained handling required

but often if delayed: delayed much longer than allowed lateness, e.g. if bus is in tunnel instead of just small transmission delay

5.3 Data Flow Example

6 Evaluation

6.1 Methodology

6.1.1 Latency Tracking

processing latency reasons:

reasons for latency: <https://flink.apache.org/news/2019/02/25/monitoring-best-practices.html#monitoring-latency>

configuration can tradeoff latency vs throughput

not the same as stream latency caused by waiting for watermark

6.1.2 Volume Scaling

part of ingest

use recording and replay multiple times

each replay in separate process with two threads: s3 reader and kafka producer

payload adjustment

6.2 Results

describe cluster

calculate cluster costs per day

6.2.1 Latency

latency: latency is the delay between the creation of an event and the time at which results based on this event become visible (<https://flink.apache.org/news/2019/02/25/monitoring-best-practices.html#monitoring-latency>)

maybe test very simple stateful job to see scalability without CEP and windowing

pass through to minimum latency possible

maybe have late data

use confidence interval

also show Kafka backlog

6.2.2 Log Size

measure log size with json vs binary protobuf

6.3 Discussion

7 Conclusion

Bibliography

- [1] J. Kreps, *I Heart Logs*, First edition. Sebastopol CA: O'Reilly Media, 2014, ISBN: 978-1-491-90938-6.
- [2] M. Kleppmann, *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*, First edition. Boston: O'Reilly Media, 2017, ISBN: 9781449373320.
- [3] Apache, *Apache Spark*. [Online]. Available: <https://spark.apache.org/>.
- [4] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008, ISSN: 0001-0782. DOI: 10.1145/1327452.1327492.
- [5] Apache, *Apache Hadoop*. [Online]. Available: <https://hadoop.apache.org/>.
- [6] T. Akidau, S. Chernyak, and R. Lax, *Streaming systems: The what, where, when, and how of large-scale data processing*, First edition. Sebastopol CA: O'Reilly, 2018, ISBN: 1491983876.
- [7] Yahoo Developer Network, *Hadoop Turns 10*, 2016. [Online]. Available: <https://developer.yahoo.com/blogs/138739227316> (visited on 08/11/2020).
- [8] Hazelcast, *Micro-Batch Processing vs Stream Processing / Hazelcast*. [Online]. Available: <https://hazelcast.com/glossary/micro-batch-processing/> (visited on 08/11/2020).
- [9] 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, "The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing," *Proceedings of the VLDB Endowment*, vol. 8, pp. 1792–1803, 2015.
- [10] A. Katsifodimos, S. Ewen, and V. Markl, "Apache Flink: Stream and Batch Processing in a Single Engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

- [11] N. Marz, *History of Apache Storm and lessons learned - thoughts from the red planet - thoughts from the red planet*, 2014. [Online]. Available: <http://nathanmarz.com/blog/history-of-apache-storm-and-lessons-learned.html> (visited on 08/14/2020).
- [12] —, *How to beat the CAP theorem*, 2011. [Online]. Available: <http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html> (visited on 08/14/2020).
- [13] N. Marz and J. Warren, *Big Data: Principles and best practices of scalable real-time data systems*. Shelter Island NY: Manning, 2015, ISBN: 9781617290343.
- [14] J. Kreps, *Questioning the Lambda Architecture*, 2014. [Online]. Available: <https://www.oreilly.com/radar/questioning-the-lambda-architecture/> (visited on 08/14/2020).
- [15] Apache, *Spark Streaming*. [Online]. Available: <https://spark.apache.org/streaming/> (visited on 08/14/2020).
- [16] Tyler Akidau, Alex Balikov, Kaya Bekiroglu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle, “Mill-Wheel: Fault-Tolerant Stream Processing at Internet Scale,” in *Very Large Data Bases*, 2013, pp. 734–746.
- [17] E. Shahverdi, A. Awad, and S. Sakr, “Big Stream Processing Systems: An Experimental Evaluation,” in *2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW)*, IEEE, 2019, pp. 53–60, ISBN: 978-1-7281-0890-2. DOI: 10.1109/ICDEW.2019.00-35.

Glossary

node

an individual server in a cluster

Protobuf Definitions