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

CEP Complex Event Processing

HDFS Hadoop Distributed File System

List of Figures

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*. This sparked the development of distributed processing engines[1] like Hadoop[2] (based on the MapReduce [3] programming model) and Spark[3]. These frameworks tackle two common challenges of large-scale batch processing [2, p. 429], [4, pp. 362–373]:

[1] We use the terms *processing engine*, *processing framework* and *processing platform* interchangeably since their semantic difference is not relevant for our purpose.

[2] `https://hadoop.apache.org/`

[3] `https://spark.apache.org/`

- 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). 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 [5]. 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 be 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 **Hazelcast.2019**. This is especially true when considering that data might arrive with a delay, which usually requires deferred processing or re-proccessing when late data arrive. Also, jobs that might span batch bounds, such as user session analysis in web applications, are inherently complex to implement [4, pp. 34–35].

Apart from the technical shortcomings, processing a continous stream of data in batches seems wrong from a philosohphical 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 [6, 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. It can enable many applications like real-time analysis of Internet of Things sensor data, continuous credit card fraud and anomaly detection, live business process and quality monitoring, among many others. 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 [4, 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 [4, 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.[4]

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.

Streams can also be regarded as *data in motion*. Scanning through the stream, it is possible to observe the evolution of data over time and build a view of the data at a single point in time. Such view are also called tables, which are *data at rest*. Relational databases have traditionally dealt with tables. Capturing the changes of a table in turn yields a stream. Therefore, streams and tables are really just two representations of the same data, a philosophy that many stream processing systems build upon [4, pp. 174–212]. This concept is also known as stream–table duality [7].

Time Domains

A stream consist of *records* that usually contain information about events. Events might, for example, be purchases, website views, temperature changes or the arrival of a bus at a stop. A stream emanates from a *producer* and can be received by multiple *consumers*. When processing an event stream, two time domains are involved [4, p. 29]:

- *Event time*: the time at which the event actually occured in the real world
- *Processing time*: the time at which events are observed at a given processing stage

These two time domains often do not coincide. The processing time can never be before the event time. However, the delay between the occurance 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.

[4] This 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.

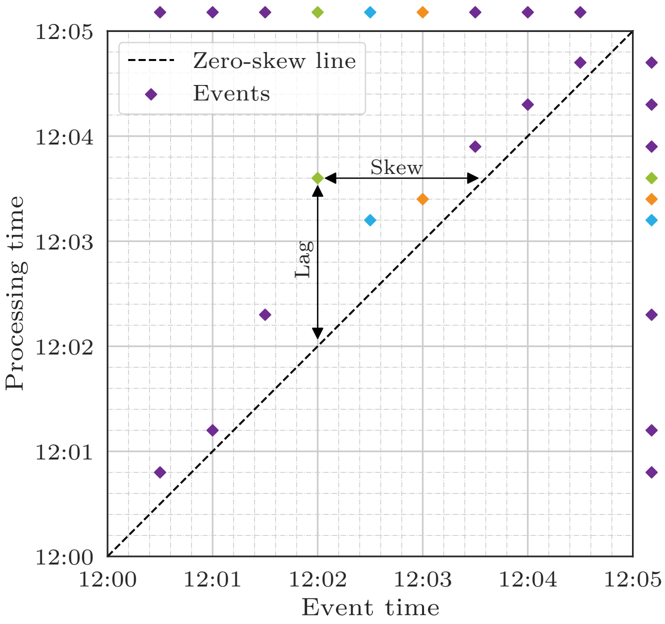

Figure 1: Relationship between event time and processing time: time skew varies a lot and leads to out-of-order arrival

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 in event time and arrive at the system in 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) [4, 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 30 s 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 time. Handling this skew and unorder is a key challenge stream processing frameworks have to solve [8, p. 3].

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 like SEEP, Naiad, Microsoft StreamInsight and IBM Streams have existed before [9, p. 37], Storm[5] was the first one to find widespread adoption when it was released in

[5] `https://storm.apache.org/`

FIGURE OF LAMBDA ARCHITECUTURE

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

2011 [4, 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 [10].

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 [11], 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 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, b 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 [12, pp. 14–20, pp. 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 [13].

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 a storing data on a distributed file system, the stream is often stored in a stream transport platform like Kafka[6] that allows *replay*, i.e. rewinding to an earlier point in time and start reading from there instead of the latest record. This enables fault tolerance and recomputation in case of processing logic changes [13].

[6] https://kafka.apache.org/

FIGURE OF KAPPA ARCHITECUTURE

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

FIGURE OF BOUNDED AND UNBOUNDED DATASETS

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

Spark Streaming[7] 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 processing in event-time order, 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 [4, pp. 27–28]. Sophisticated time handling with high flexibility was explored in Google's company-internal MillWheel [14] framework and Dataflow [6] processing models. Flink[8] was the first open-source framework to incorporate 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 [9, p. 35], [4, 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 use case.

Stream processing jobs often consist of multiple stages that are connected into a directed *dataflow graph*[9] [1, p. 30]. The graph starts at one or more stream sources and ends at one or more stream sinks. *Operators* in between can filter, aggregate, join and split streams or transform them otherwise. Operators are essentially the graph's vertices, with streams between sources, operators and sinks forming the edges. A vertex further to the start of the graph (i.e. the sources) is referred to as *upstream*, and a vertex further to the end of the

[7] `https://spark.apache.org/streaming/`

[8] `https://flink.apache.org/`

[9] Note that dataflow graphs are a general concept that are also found in batch processing, and are different from the Dataflow stream processing model.

FIGURE OF DATAFLOW GRAPH

Figure 5: Example of a dataflow graph: three streams are joined, transformed and emitted as a single stream

graph (i.e. the sinks) is referred to as *downstream*. An example graph is shown in figure 5. The graph can contain also contain loops to enable iterative algorithms like incremental machine learning and graph processing [9, p. 33]. Dataflow graphs provide a very flexible programming model for building stream processing jobs, but also allow the processing framework to apply optimizations like operator fusion and fission [15]. Stream processing jobs can also be written in a higher-level abstraction like streaming SQL and pattern matching specifications [16], which are eventually compiled down to dataflow graphs [9, p.30].

2.2.3 Processing Patterns

Streams require processing patterns that support their unbounded nature. There are three major categories: [4, p. 35]:

Time-agnostic When the processing logic is purely data-driven, ordering by time is irrelevant. This makes processing very simple because out-of-order records need not be accounted for, therefore this pattern is supported by even the most basic streaming systems. This includes record-by-record processing like filtering based on a record attribute but also inner joins, where a joined record is produced once the respective records from all input streams have been observed.[10]

Approximation Approximation algorithms like sketches for frequency distribution or distinct-value queries [17] are optimized to handle large quantities of data by trading exact correctness for computational feasability, albeit usually within some error bounds. However, they are often complicated which makes it difficult to invent new ones. Furthermore, they usually work only in processing time, limiting their applicability.

Windowing To handle the unboundedness and lack of completeness of streams, they can be chopped into bounded datasets known as *windows*, which can be processed independently. For example, a stream can be divided into contiguous sections of 1 min, and results like aggregations are computed per section. More complex, even arbitrary, windows are also possible. Note that this pattern includes many more time-based processing types that are not immediately obvious. For example, pattern

[10] If many uncompleted joins are to be expected, a timeout-based garbage collection becomes necessary to limit memory requirements, introducing a time component.

recognition effectively builds a window for each stream record ending with the final record of the pattern, resulting in variable-length windows [18, p. 350]. Additionally, regular windows can be used to limit the records regarded for pattern matching and expire partial patterns to keep state size in check [18, p. 354]. Another example are outer joins, where a joined record can also be produced when the respective records have only been observed from some of the input streams. Outer joins on streams require a timeout after which a partial join should be produced, which effectively determines the window length, where the window contains all records that were regarded for a record's join.

The focus of the rest of this thesis will be on the windowing pattern, since it has unique challenges compared to time-agnostic and approximative processing. Specifically, correct windowing of out-of-order streams requires event-time awareness, and relating data within those window requires keeping consistent and fault-tolerant state. We will refer to this type of stream processing as *stream analytics*, since time and state are required for producing sophisticated and valuable insights. Note that the term "stream analytics" is not well-defined in the literature and the boundary to other techniques like Complex Event Processing (CEP), Event Stream Processing, Distributed Stream Computing and Information Flow Processing is blurry [19], [2, p. 466]. For our purpose, we consider any kind of sophisticated stream processing as stream analytics. We will now regard the challenges of time and state from a stream processing job developer's perspective. For an elaborate overview of specific implementations of out-of-order-data management and fault-tolerant state management in early and modern stream processing frameworks, refer to [8, chapters 3–5].

2.2.4 Windowing

Windowing is a key technique for enabling processing of unbounded datasets which inherently lack completeness. Each window of a stream is a finite chunk that is a complete dataset in itself. In this section, we will look at window types and how latency and correctness can be balanced for the use case at hand.

Windows can either be non-keyed (windows apply to the stream as a whole) or keyed (the stream is divided into subsets by key, e.g., per user, to which windows are applied individually). Three commonly found window types are shown in figure 6 [6, p. 1794]:

Fixed/tumbling Fixed windows are defined by a fixed-length temporal window size. For example, a fixed window of 10 min divides the stream into subsets of data from 12:00 to 12:10, then 12:10 to 12:20, continuing that way until the processing is stopped. Windows may either be aligned or unaligned across keys, depending on if the windows

(a) Fixed (b) Sliding (c) Session

Figure 6: Common window types

of different keys start at the same time or are staggered by an offset, which spreads window completion load more evenly across time.

Sliding/hopping Sliding windows are defined by a fixed window size and a fixed period. For example a sliding window of 10 min starting every 1 min divides the stream into subsets from 12:00 to 12:10, 12:01 to 12:11, so every record ends up in 10 windows. The window size is often an integer multiple of the period, and sliding windows can also be aligned or unaligned. Note that fixed windows are a a special case of sliding windows where size equals period.

Session Session windows are defined by a timeout gap to capture periods of activity. For example, user activity analysis on a website during one sitting is a common use case for session windows. Session windows are defined per key and the length depends on the data involved, therefore they are inherently unaligned. Because the window length cannot be defined in advance, they are one area where the stream processing excels compared to batch processing. Since sessions may span multiple bounded batch datasets, the dataset must be treated as unbounded. Otherwise, complex stitching is required [4, p. 35].

Apart from these time-based window types, there are also tuple-based windows that contain a fixed number of records. However, they are essentially a form of time-based windows with incrementing logical timestamps [20, p. 47] and will therefore not be regarded further here.

All windows can be defined in both time domains. When windowing by processing time, incoming data are buffered for the specified period and then processed, as shown in figure 7a. This is straigtforward because windows are complete as soon as the window time has passed, therefore there are no late data to handle. It works well for many monitoring scenarios, where insights about data as they are observed is desired. However, most use cases require processing of data in event-time order, but there might be an arbitrarily long delay between an event occuring and the event being processed, which changes the order of events. This can lead to incorrect results if not handled appropriately [4, p. 41]. For example, when recognizing patterns, out-of-order data can result in matches that do not actually exist, and other matches might be missed. In billing applications, where correctness is paramount, quarterly reports might contain incorrect numbers if records end up in the wrong windows. Therefore, windowing by processing time is not sufficient in many cases.

(a) Processing time (b) Event time

Figure 7: Windowing in different time domains

Windowing in the event-time domain, as shown in figure 7b requires ordering the out-of-order data to assign them to the correct window. This requires extra effort because event time is not the natural time domain of processing. On the one hand, data need to be buffered longer until the window is closed, therefore windows of the same size are open much longer in event time than in processing time. This demands more resources, but optimizations can be made to, for example, store aggregates incrementally. What is more challenging, however, is judging the completeness of a window. If the event-time skew can be arbitrarily long, it is non-trivial to judge when all data for a specific event-time window have been observed. This simplest approach is to delay processing for a fixed amount of time. For example, if data is usually not delayed for more than 30 s, we can reasonably assumed that all data for this window has arrived when we close the window 35 s after a record with the window end timestamp has been observed. However, this is essentially a tradeoff between latency and completeness (and by extension, correctness), since waiting longer necessarily increases latency but also increases the probability that no data is missed. This black-and-white tradeoff is far from satisfactory for many use cases. Therefore, the Dataflow [6] model introduced fine-grained control over window semantics to balance correctness, latency and cost.

Windowing in Dataflow

In the Dataflow, windowing is strictly event-time-based. However, processing-time windows are possible when assigning the arrival time as the event time. We will now look at the four aspects that enable a clear and flexible definition of windows. For a more detailed description, refer to [4, chapter 2].

Transformations Transformations define what results are produced from the records in a window. This includes aggregations like summing and counting, training machine learning models or detecting anomalies. Depending on the specific transformation, individual records can either be accumulated and processed all at once when window results are *materialized* (i.e. emitted and sent downstream for storage or further processing), or records can be aggregated eagerly to spread computation load more evenly and minimize state size. There are various choices for which event timestamp to assign the result, but often the end of the window is used [4, p. 101].

Windowing Windowing determines which records are grouped together based on some strategy. This includes fixed, sliding and session windows, but custom strategies are supported as well. Custom strategies (but also the built-in ones) consist of window assignment, which assigns records to one or more windows, and optional window merging, which allows merging of windows for window evolution as more data arrive. For example, window merging is required for session windows when a record arrives that connects two sessions which were before separated by the timeout gap [4, pp. 136–146].

Triggers Where windowing determines the location of windows in event time, triggers specify when transformation results are materialized in processing time. This allows windows to be evaluated more than once, where each specific result of the window's transformation is referred to as *pane*. There are two general types of triggers [4, p. 60]:

- Repeated update triggers: these trigger window evaluation periodically, either after a specific count of records or at some processing-time frequency, such as every minute. The choice of period is primarily a tradeoff between latency and computation requirements.
- Completeness triggers: these trigger window evaluation when they believe that all data for the window has been observed, and therefore the window is complete.

Repeated update triggers show evolving results over time that converge towards correctness, but they do not indicate when correctness is achieved [4, p. 63]. Therefore, completeness triggers may be more appropriate for use cases where correctness is important.

Output Mode The output mode describes how different panes, i.e. subsequent evaluation results of a window, are related and refine previous results. Therefore, the choice is only relevant if windows are triggered multiple times. We will use the naming proposed in [4, p. 94] instead of the original naming from the Dataflow paper for clarity. There are three types of output modes, with an example of two panes shown in table 1:

- Delta: upon triggering, the result is materialized and any stored state is discarded. Therefore, successive panes are independent of each other. For example, when summing input records, only the sum of all panes will yield the total sum for the window.
- Value: upon triggering, the result is materialized but stored state is retained. Therefore, successive panes build on each other's results. For example, when summing input records, each pane contains the total sum for the window so far.

	Delta	Value	Value and Retracting
Pane 1: inputs=[3]	3	3	3
Pane 2: inputs=[6, 1]	7	10	10, -3
Value of final pane	7	10	10
Sum of all panes	10	13	10

Table 1: Example of windowing output modes

- Value and retracting: upon triggering, the result is materialized and any stored state is discarded. Additionally, previous panes are explicitly retracted. For example, when summing input records, each pane contains two parts: the total sum for the window so far, and a retraction for the old sum.

The choice of output mode usually depends on the input expected by downstream consumers. Aggregating consumers might expect deltas, while databases that are updated with new data require values.

These four composable pieces provide flexible tools to balance correctness and latency by adjusting trigger frequencies and output modes, but also cost by affecting compute and memore requirements. Completeness triggers play an important role for correctness, but can be hard to implement, especially when event-time skew is highly variant. *Watermarks* are an approach to indicating input completeness in the even-time domain [4, pp. 64–66]. The watermark denotes the point in event time up to which the system believes all inputs with lower event timestamps have been observed. In other words, the watermark is an assertion that no more data with event timestamps earlier than the watermark will arrive. Completeness triggers can trigger window materialization once the watermark passes the window end in the belief that no more records will be assigned to that window. Note that watermarks must be monotonically increasing [4, p. 88].

Watermarks can be a strict guarantee or an educated guess of completeness. Perfect watermarks are possible when the system has full knowledge of all input data, for example, when assigning arrival timestamps as event timestamps. In some cases, the data source itself might produce watermarks. *Late data*, i.e. data with an event timestamp earlier than the watermark that arrive past the watermark, will never occur. In most practical applications, only heuristic watermarks that approximate a perfect watermark based on the available information are possible. Heuristic watermarks can be generated by incorporating knowledge of the sources [4, p. 66], but also in form of percentile watermarks [4, pp. 106–108] based on the event-time skew distribution, if known. This would, for example, enable watermarking after 99% of all data are believed to have been observed, decreasing latency by ignoring stragglers. Another common strategy is by specifying a fixed bound for event-time skew, limiting the expected out-of-orderness. For example, the watermark could always lag 10 s behind the latest known timestamp if we know that the event-time skew will never exceed 10 s.

While watermarks are very useful to judge window completeness, they have two shortcomings [4, pp. 68–69]. Watermarks may sometimes be too slow, which increases latency. This might either be the case because the data really have a high delay, or because the watermark generation overestimates the delay. On the other hand, heuristic watermarks might be too fast due to their approximate nature, in which case late data might arrive after the watermark. Therefore, watermark-based completeness triggers alone cannot provide both low-latency and correctness.

This motivates the use of multiple triggers per window. Early repeated update triggers compensate for watermarks being too slow by periodically providing early results which are incomplete. A single on-time trigger based on the watermark materializes results which the system believes to be correct. In case the heuristic watermark was too fast, late repeated update triggers refine results when late data arrive. Often, the late trigger fires for every late data record, which can drastically increase the number of computations in case of watermarks that are too fast by a large margin. Note that the output mode needs to be set appropriately when windows might be triggered more than once. This ensures that downstream consumers process multiple panes per window correctly.

Window state needs to be retained after the watermark when late triggers are enabled. Due to practical resource limitations, a maximum *allowed lateness* in processing time must be specified. After a window is completed by a watermark, state is expired after the maximum allowed lateness. Any record that arrives later will be discarded. Since the value of data diminishes with time, trading off resource cost for data value is usually sensible

2.2.5 State Consistency and Persistence

Any stream analytics solution that does not process streams record-by-record but correlates multiple records is stateful. For windowing, state consists of intermediate aggregation results. For pattern recognition, state consists of partial matches. For online machine learning training, the state consists of the current model parameters. Since stream processing jobs are effectively intended to run forever, interruptions like node failures, infrastructure maintenance or code changes are inevitable. To ensure correct results, the state needs to be persisted in a fault-tolerant way. This is especially important, since unbounded datasets usually cannot be replayed in their entirety, either because they are not retained forever, or because it is computationally infeasible [4, pp. 216–218]. Simply storing state externally in a database can become a bottleneck [21, pp. 1718–1719]. Compare this with

(a) Example scenario

(b) At-least-once

(c) At-most-once

(d) Exactly-once

Figure 8: Example of different processing semantics after failure recovery

batch processing, where it is often assumed that the dataset can be reprocessed in its entirety until the job succeeded.

Therefore, correct and efficient fault tolerance in stream processing requires persistent state that can be checkpointed. *Checkpointing* is the process of persisting in-memory state to a durable storage medium. This state needs to be exposed to the stream processing framework for management. To expose state, frameworks usually provide a flexible API with support for a variety of data structures [4, p. 228], often with efficient implementations of lists and maps [21, p. 1721]. Apart from checkpointing for fault tolerance, this allows state redistribution during cluster scaling and alleviates the developer of needing to implement efficient persistence [21, pp. 1718–1719].

After fault recovery, the processing framework needs to guarantee that any materialized results are identical to the results if no fault occured. This is required for *consistency* [8, p. 15]. The key to consistent and correct results is *exactly-once processing*. This means that every stream record is guaranteed to be processed exactly once even in case of failures. For example, assume that a job counts the number of records in a stream, as shown in figure 8. After having processed record 5, the job fails and needs to be restarted on another node. If the restarted job starts earlier than record 5, the total count will be higher than the actual count. This is referred to as *at-least-once processing*, since each record is guaranteed to be processed at least once. On the other hand, of the restarted job starts after record 5, the total count will be lower than the actual count. This is called *at-most-once processing*. Only if the job is guaranteed to be restarted from record 5 for exactly-once processing, the total count will be correct.

Exactly-once processing needs to be end-to-end, which means that checkpointing not only needs to consider the processing framework but also sources and sinks [4, p. 153]. A persistent and immutable source is required to be able to replay the stream from the last checkpointed position [21, p. 1722]. If the stream is ephemeral, only at-most-once processing might be possible since the stream cannot be restarted from the correct position. The sink needs to be either idempotent (like most databases) or support transactions (like using two-phase commits) [21, pp. 1725–1726]. To enable end-to-end exactly-once processing, the persistent state consists of the actual application state, but also needs to include the position in the stream on which that application state is based. Based on these information, a recovered job can provide consistent and correct results. However, exactly-once processing is not possible due to the nature of some sources and sink, and consistent and correct results cannot be guaranteed in those cases.

Most stream processing frameworks do not offer true exactly-once processing due to performance reasons, but rather *effectively-once processing*. This means that each record will only affect the results once, but might actually be processed multiple times in case of a job restart due to failure. This raises consistency issues, since non-deterministic operations might produce different results for the same inputs. For example, a database lookup for stream enrichment might return different data of a table has been updated in the meantime. Frameworks cope with non-determinism by checkpointing results from such transformations [4, pp. 155–156] or simply assuming deterministic transformations [21, p. 1722].

2.3 Stream Transport

Having regarded streams as a concept, we now show how streams are physically manifested, stored and transported around. A *stream transport* system is responsible for moving streams between producers, processing systems and consumers. In its simplest form, a direct TCP connection can transport records from upstream producers to downstream consumers. For low-latency applications, UDP multicasting can be used. Brokerless messaging libraries like ZeroMQ[11] implement publish/subscribe messaging on top of these network protocols. Even HTTP and RPC requests can be used to push records from producers to consumers. However, such direct messaging methods fall short in terms of delivery guarantees and fault tolerance [2, pp. 441–441].

Message brokers like ActiveMQ[12], Pulsar[13] or RabbitMQ[14] provide a way to decouple producers and consumers using common protocols such as AMQP or JMS. The broker can store records durably to cope with offline or slow consumers by persisting them on disk until delivery is acknowledged. Many are also distributed for fault tolerance in case of broker node crashes. The decoupling makes record transport asynchronous, since producers do not know when a message is delivered to consumers [2, p. 443]. Broker-centric instead of direct messaging furthermore allows multiple consumers to receive a stream without needing to change the producer, enabling organizational scalability and flexibility [1, pp. 20–22] as well as debugging and monitoring by peeking into the stream [1,

[11] `https://zeromq.org/`

[12] `http://activemq.apache.org/`

[13] `https://pulsar.apache.org/`

[14] `https://www.rabbitmq.com/`

(a) Load balancing (b) Fan-out (c) Mixed

Figure 9: Message distribution patterns with multiple consumers

pp. 31–32]. There are two patterns when multiple consumers receive the same stream, as illustrated in figure 9 [2, pp. 444–445]. Load balancing means that each record is delivered to only one consumer, so processing load is spread over all consumers. Fan-out means that each record is delivered to all consumers, which allows for independent consumers to receive the full stream. Combinations of these two patterns using consumer groups are possible as well.

However, when using message brokers as transport platform in streaming systems, they have a number of shortcomings. First, they might deliver records more than once in case of lost acknowledgements, which might break exactly-once semantics or require deduplication in the processing framework. Secondly, most brokers do not make guarantees about record order, which requires special consideration in the processing framework. Thirdly, message brokers are ephemeral by design, which means that messages are discarded in the broker once delivered [2, pp. 445–446]. This prevents stream replay for fault recovery or reprocessing after processing code changes.

2.3.1 Immutable Logs

Message brokers that use a log as data structure do not have the shortcomings of regular message brokers. A *log* is an append-only sequence where each record is assigned a monotonically increasing number called *offset* [1, pp. 1–3]. Records are stored in the order they are appended and cannot be updated or deleted, therefore logs are immutable and are totally ordered. Log data structures are commonly found in databases for in form of write-ahead logs or replication logs, but also in many distributed consensus systems [22, pp. 54–66]. When used for streaming, producers append new records to the end of the log, and consumers receive records by reading the log sequentially. Once a consumer reaches the end of the log, it waits for new records to be appended. Fan-out for multiple consumers is trivial since records are retained.

The total ordering solves the at-least-once delivery and out-of-orderness problems of traditional message brokers, since the offset enables consumers to know exactly which records they have or have not read yet. This offset can be part of state checkpointing as described when talking about state consistency in subsection 2.2.5. A configurable, possibly infinite, retention period instead of ephemeral records is feasible since performance does not degrade with increasing data size [23, p. 3]. This enables stream replay which is required for exactly-once processing in case of faults. In fact, fault tolerance in many

FIGURE OF PARTITIONS

Figure 10: Log partitioning: each partition is an independent log with internal total order

modern stream processing systems was only made possible by fundamentally relying on this ability of log-based sources [4, pp. 390–391]. The ability to replay old data is also useful for reprocessing after bugfixes and for development and regression testing. While this capability is very common in batch processing, durable logs offer a robust and reliable streaming alternative [4, p. 390].

Durable log transportation platforms also provide an isolation layer between producers and consumers [2, pp. 450–450], [1, p. 32]. The log acts as large buffer that prevents consumers from being overwhelmed by a high volume of data by allowing consumers to read records at their own pace. Once the produced volume decreases, the consumer can catch up without needing to worry about data loss. This moves backpressure handling from the consumer to the transport layer.

A stream can be divided into *partitions* that contain the same record types but are otherwise separate logs that can be read and written to independently from other partitions [1, pp. 24–26]. This is shown in figure 10. Partitions enable efficient distributed logs, since each partition can be hosted on a different node for horizontal scalability without synchronization overhead. Additionally, partitions can be replicated to other nodes for fault tolerance. While each partition is totally ordered within itself, there is no global order between partitions. In practice, this is only a minor limitation since processing jobs often handle partitions independently (e.g., in keyed windows). In that case, it makes sense to use the same key for determining the log partition, since a random partition is selected by default. Partitions also enable load balancing, since each partition can be assigned to a single consumer. In that respect, partitions are the smallest unit of parallelism [23, p. 4].

2.4 Event Processing

So far, we have regarded streams of arbitrary records that have a timestamp. Usually, each record represents an event in the real-world. Many authors [8], [24] go as far as treating stream records and events synonymously. Events are usually generated continously and therefore lend themselves well to be treated as unbounded stream, leveraging the processing and transport methods mentioned before. Thus, event processing can also be implemented as operator in a general-purpose stream processing framework instead of using a standalone event processing engine. However, the notion of events has been used long before stream processing frameworks gained popularity, for example in active database systems arising around 1990 [25, p. 5]. Therefore, we will regard events and event processing detached from stream processing in this section.

An *event* is an occurence, something that has happened, within a particular system [26, p. 4]. That system is often the real world, but may also be artificial like a simulation. Change events refer to significant changes of an environment, while status events refer to the observation of some value, even if it remained the same [27, p. 1]. Events are always time-related and are therefore assigned a timestamp. Events can also have attributes that provide more detailed information like the value of a sensor readings. The attributes are defined in the *event type* which determines the event's semantic intent, and individual instances of an event type are called *event occurences* [26, pp. 62–63]. For example, events of type "environment measurement" might always have the attributes "temperature" and "humidity", and instances of the event happening at different times have different values for these attributes. Events originating in the real world might have a geographical location as attribute, making the event a *geo-event*. *Complex events*, also called composite events, emerge as result of relating multiple events in event processing, for example through windowed aggregation or pattern matching. In this context, regular events are also referred to as *primitive event*.

Processing of events comes in two flavors [26, pp. 10–11]. *Event-based programming*, sometimes called *event-driven architecture*, uses events for interaction between components of a system. This asynchronous model provides better decoupling than synchronous request–response communication, since event producers have no expectation of event consumers' actions [26, pp. 33–34]. This event-based communication, can be as simple as a notification, in which case the attribute size is rather small, often just an identifier which consumers can use to lookup further information. In other patterns like event-carried state transfer or event sourcing, that information is itself stored in an event attribute [28]. Event-driven architectures are described in-depth in [26].

In the other flavor, events are the subject of filtering, transformations and pattern recognition instead of a means of communication [26, pp. 121–127]. This type of event processing is very similar, if not identical, to stream processing, and many concepts like event-time ordering and windowing apply to both in the same way. Therefore, treating stream records specifically as events instead of arbitrary objects that happen to be events is rather a semantical than technical difference. This thesis will focus on the second processing type, and we will approach event stream processing from the streaming side rather than the event side.

Pattern recognition over event streams emanates from the event domain, therefore it makes sense to regard stream records as events. *Pattern recognition* is the process of matching a stream of events against some pattern, and detecting instances of the pattern in the stream. This technique can be used to detect anomalies, monitor patients or stock tickers, or do predictive analytics. Pattern recognition is sometimes referred to as Complex Event Processing, but we will avoid this term due to its ambiguity in the literature. Processing of events in the order that they happened in is often crucial for correct pattern recognition. While standalone pattern recognition engines have existed for a long time, libraries built on top of stream processing frameworks can leverage their event-time ordering guarantees and allow pattern recognition to be easily integrated into stream processing jobs.

The *event pattern* is the template specifying one or more combinations of events to match the stream against. There is a wide range of pattern types that can be combined, but the most common include [26, pp. 219–236]:

- Logical expressions: "events of type A, B and C occured"
- Comparisons: "an event of type A with attribute x > 10 occured"
- Temporally ordered sequences: "first, an event of type A, then multiple events of type B, and finally an event of type C occured"
- Trends: "multiple events of type A occured with attribute x increasing monotonically"
- Spatial distance: "events of type A and B occured within 5 km of each other"
- Spatial direction: "events of type A and B occured with A moving towards B"

The pattern recognition process is governed by policies that, for example, determine which events are selected for matching, whether events can participate in multiple matches and whether matches can overlap [26, pp. 237–242]. The combination of patterns and policies provides rich tools to express a variety of situations and can therefore be applied widely.

3 Streaming Platforms

After having regarded the theoretical foundations of streaming, we now look at existing implementations of stream transport and processing platforms that are available to the public (some companies have custom platforms for internal use only). We will present them under the aspects of correctness, fault tolerance, latency and scalability.

3.1 Stream Transport Platforms

Stream transport platforms are responsible for moving streams between producers and consumers, but may also store them for replay. Theoretically, any message queue or publish–subscribe message broker like ActiveMQ, Pulsar or RabbitMQ but also hosted services like Google Cloud Pub/Sub[1] could be used for this task. However, these do not provide the end-to-end exactly-once[2] and ordering guarantees as well as stream retention required for correct and fault-tolerant stream processing. Log-based messaging systems, on the other hand, store stream records in order and can be used as exactly-once sources by including the record offset in checkpointed state. Kafka and Amazon Kinesis Data Streams[3] are two publicly available log-based stream transport platforms[4].

Kinesis Data Streams is a fully managed and reliable service hosted in the AWS cloud [31] and allows easy integration with other AWS offerings. The number of partitions per log, retention period and record encryption are the only configuration options. Up to 1 MB or 1000 records can be produced to each partition per second, and up to 2 MB can be consumed from each partition per second. Records can be retained for up to 7 days, which is sufficient to allow replay for fault tolerance in stream processing, but generally not enough for long-term retention to support data reprocessing in retrospect. While Kinesis

[1] `https://cloud.google.com/pubsub`

[2] RabbitMQ has limited support for being used as exactly-once source but not as exactly-once sink since transactions are not supported [29]. Pulsar can be used as exactly-once source using the same mechanism and as exactly-once sinks once transactions are implemented in Pulsar 2.7.0, therefore Pulsar could become a viable Kafka alternative for some applications in the future [30].

[3] `https://aws.amazon.com/kinesis/data-streams/`

[4] The actual implementation of Kinesis Data Streams is unknown since it is a managed service, but because only appending is supported we regard it as log for all intents and purposes.

Data Stream's operational costs are about four times lower than Kafka's, it lacks flexibility and throttles producers quickly [32].

3.1.1 Apache Kafka

Apache Kafka is a widely used [33] open-source stream transport platform first developed at LinkedIn. Kafka was the first system to leverage logs for high-throughput, low-latency messaging [23] and has been very influential on the design of modern stream processing platform by enabling fault-tolerance through replay [4, pp. 390–391]. Records can be retained based on a time-based or size-based policy, with support for infinite retention which makes Kafka not only suitable for stream transport but also for persistent storage, effectively making it a file system for streams. This enables streams which show the evolution of data as primary data source, instead of traditional tables which represent data at a specific point in time[5]. For example, the New York Times stores all historic and present articles and assets like images and tags in a single Kafka partition for total order, and every update to content does not overwrite old data but is appended to the log [35]. For serving, the log is replayed from the beginning and an index is built for efficient lookup of content. However, Kafka is usually used as continuous event streaming platform. For in-depth information on development with and operation of Kafka as well as internals, refer to [36].

Nodes in a Kafka cluster are called *brokers* and serve as distributed storage of *topics*. Topics are Kafka's notion of streams that contain a certain event type, and each topic consists of one or more partitions. The logical position of a record in a partition is called the offset. Records are guaranteed to be stored in the order they are appended to the log within a single partition, but not across partition of a topic. The retention period can be configured per-topic, with records exceeding the retention period being removed from the beginning of the log. Logs can also be compacted, which means that only the most recent record for a key is retained. This can make sense to keep log size small if applications are only interested in the most recent record for a key. Partitions can be replicated across the cluster for fault tolerance. Each partition has a broker acting as partition leader, which clients connect to to consume existing records or append new records to a partition log, as shown in figure 11. Follower replicas do not serve client requests but replicate records from the leader, and can be promoted to become the new partition leader in case the current leader crashes. Only in-sync replicas, i.e. follower replicas that have caught up with the current leader, are eligible for leader election. Cluster metadata like broker membership,

[5] This approach leverages the stream–table duality to "turn the database inside out" [34], [2, pp. 459–462], [4, pp. 174–212]

FIGURE OF KAFKA CLUSTER WITH PARTITIONS, LEADERS AND REPLICATION

Figure 11: Structure of a Kafka cluster: Partitions are spread across brokers acting as leader and follower replicas

partition locations and topic configurations are stored in Zookeeper[6], a distributed and highly reliable configuration service required to run Kafka. However, this dependency will be removed in future releases to simplify deployment.

Kafka supports flexible consumption patterns using *consumer groups* previously shown in 9. Each partition of a topic is assigned to one consumer in a consumer group. Therefore, load balancing can be achieved when every consumer belongs to the same consumer group. On the other hand, if all consumers are part of different consumer groups, every consumer receives all records for fan-out. Mixed patterns are also possible by having multiple consumer groups with multiple consumers. When using the load balancing pattern, Kafka effectively acts like a traditional message queue, whereas fan-out is more similar to publish–subscribe semantics. Offsets between consumers of a consumer group are coordinated via special topics, which becomes necessary when consumers within a group change and partitions need to be rebalanced between consumers. Consumers commit their offsets regularly to avoid duplicate consumption. An important design decision is the use of consumer polling instead of broker pushing of records. This allows consumers to control the pace of consumption and not be overwhelmed, but also enables replay of older records. Kafka brokers use a zero-copy method when serving consumers, which means that Kafka sends records directly from disk to network without intermediate buffers, which is enabled by using the wire format for storage. Multiple brokers called bootstrap servers can be specified in the consumer for the initial connection to the cluster, after which the consumer can ask for the desired partition leader.

Producers also use bootstrap servers to connect to the partition leader for appending records to the log. Producers can control how many replicas must have received the record before the write can be considered successful. A producer may not wait for a reply from the partition leader at all to achieve very high throughput. A producer may wait for a success reply from the leader to be able to retry in case of leader failure. At the highest level of write safety but also the highest latency, a producer may wait for all in-sync replicas to have received the message. Producers can also batch multiple records in one write request to improve throughput at the cost of latency. The partition a new record will be assigned to is determined by a partitioner. The default partitioner assigns partitions based on a key like a user identifier or at random if none is given, but custom partitioning strategies are supported. Depending on the configuration, records sent from

[6] `https://zookeeper.apache.org/`

a single producer may or may not be appended to the partition in the same order. Events must be serialized before being stored as binary payload in a record. Kafka provides basic serializers, but often custom serializers based on JSON or a generic serialization library like Avro[7], Google Protocol Buffers[8] (also known as *protobuf*) or Thrift[9] is used. Kafka also supports transactions for atomic writes of multiple records across multiple partitions, which enables producers to implement exactly-once stream sinks. The prevent consumption of records of uncommitted transactions, consumers must support the isolation level directive.

Due to Kafka's popularity, a rich ecosystem of commercial offerings and libraries has emerged. Confluent, a company founded by one of the inventors of Kafka, provides a commercial Kafka platform with features like schema registries (centralized store for serialization schemas to be shared between consumers and producers) and tiered storage (offloading of old retained data from broker nodes to cheaper and scalable store like Amazon S3, also accelerating failure recovery and cluster rebalancing). Cloud platform providers like Amazon and Google also have fully managed offerings for Kafka, combining the flexibility and configurability of Kafka with the ease of operation of Kinesis Data Streams. Other tools like Kafka Connect allow the ingestion of streams from other systems to integrate Kafka into existing technology landscapes, and even more tools exist to mirror whole clusters, monitor performance and availability or run queries on streams.

3.2 Stream Processing Platforms

Stream processing platforms are also part of the stream transport platform ecosystem, consuming and producing streams while doing processing in between. They can also act as bridge between different transport platforms. The platform facilitates the development of processing jobs by taking care of aspects like event-time ordering, fault tolerance of state and scalability, and providing APIs to perform transformations on streams like windowing or pattern recognition. There is a number of frameworks with different approaches and paradigms that have been tried over time, refer to [8] for a comprehensive overview of the evolution and features of stream processing systems. We will focus on modern frameworks that can guarantee correct and consistent results through exactly-once processing. Therefore, we will not regard Samza[10] (which only supports at-least-once processing), Spark Streaming (which has no support for event-time ordering and uses micro-batching) and Storm (although rudimentary stateful processing is supported with the Trident extension).

[7] https://avro.apache.org/

[8] https://developers.google.com/protocol-buffers

[9] https://thrift.apache.org/

[10] http://samza.apache.org/

However, they might still be suitable for certain applications. For example, Spark Streaming can leverage the extensive machine learning features of the Spark ecosystem. Kafka Streams[11], Spark Structured Streaming[12], Hazelcast Jet[13] and Flink are open-source stream processing frameworks which are suitable for stream analytics. All frameworks support high throughput and low latency, but exact numbers depend on the workload and configuration and will therefore not be compared here. Some performance results are shown in [37]. Note that all frameworks are under active development and new features are developed constantly, therefore existing limitations might be overcome.

Kafka Streams is a lightweight processing framework designed to run alongside the transport cluster, simplifying deployment because no separate cluster is needed but limiting it to Kafka as stream transport platform. Low-latency exactly-once processing with event-time ordering is supported. State can be stored in special compacted changelog topics for fault tolerance and queried from external applications. Kafka Streams also transparently handles rebalancing of partitions when adding and removing processing instances through consumer groups. Kafka Streams applications perform stateless or stateful transformations on stream and table abstractions as well as joining and windowing. While not having the flexibility of the Dataflow mode, windowing in Kafka streams supports all three common window types and early/late triggers. In general, Kafka Streams is suitable for many processing types but is limited to Kafka sources and sinks, and the simple deployment lacks flexibility.

Spark Structured Streaming is a stream processing API built on top of the Spark batch processing framework. Jobs can be submitted to regular Spark clusters and run there in a scalable and fault-tolerant fashion. Kafka and Kinesis Data Streams can be used as stream sources and sinks. Jobs can be run in two modes. Exactly-once processing can be guaranteeed in micro-batching mode, while only at-least-once processing is possible in continuous processing mode. However, latencies can only be as low as 100 ms in micro-batching mode whereas latencies of 1 ms can be achieved with continuous processing. Spark Structured Streaming supports all common stream operations, but uses the latest event timestamp as watermark, which can lead to incorrectly discarded data in case of varying event-time skew [4, p. 386]. Spark Structured Streaming does not use a dataflow graph processing model but uses an incremental version of the static queries on an unbounded table from the batch API [38, p. 601]. Compared to other frameworks, Spark Structured Streaming has not many unique advantages. However, stream processing jobs can be run on an existing Spark cluster and existing developer know-how from Spark batch jobs can be applied to Spark Structured Streaming.

[11] `https://kafka.apache.org/documentation/streams/`

[12] `https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html`

[13] `https://jet-start.sh/`

Hazelcast Jet is the the newest of the four frameworks, having been released in 2018 with focus on high performance and easy maintenance [39]. Hazelcast Jet is built on top of the Hazelcast In-Memory Data Grid to support distributed in-memory data structures and lookups for stream enrichment. Jobs can be run on a a cluster for scalability and fault tolerance. The framework comes with many connectors for sources and sinks, and supports exactly-once processing with Kafka. While it has no SQL-like API, common stream operations with event-time ordering are supported, including early triggers. It also integration with change data capture software for relational databases, and supports batch processing with a similar API to stream processing. Coroutines with shared threads instead of individual threads are used for executing tasks, maximizing CPU utilization and reducing context switching overhead. Hazelcast Jet is a promising stream processing engine, but still immature due to its young age.

While Beam[14] is not a stream processing engine, it is still worth mentioning in this context. Beam is an abstraction layer that allows to write stream and batch processing jobs which can be executed on a number of stream execution frameworks, including Flink, Hazelcast Jet, Spark and Samza but also Google's fully managed Cloud Dataflow[15] platform. Beam is one implementation of the Dataflow paper [6] and therefore provides an API for all windowing concepts presented earlier. While Beam was designed as portability layer for jobs, the exact features like exactly-once processing and window output modes are dependent on the underlying stream execution framework.

3.2.1 Apache Flink

flink is one of most capable open source platform kinesis data analytics is hosted Ververica Platform is commercial distribution

APIs

datastream, dataset, SQL async queries event time unit testing watermarking strategies transfer of dataflow model to flink triggers/evictors beam runner

Cluster

workers and masters task slots high availability

[14] https://beam.apache.org/

[15] https://cloud.google.com/dataflow/

Execution Model

tasks operators parallelism co-location and operator chaining shuffling after keyby watermark propagation backpressure sampling shuffling

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

State

state backends broadcast state queryable

Checkpointing

barriers aligned and unaligned, difference if IO or compute is bottleneck checkpoint for fault tolerance savepoint for reconfiguration and code changes

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

general design advice: [2], [1]

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

Serialization formats [2, chapter 4]

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 ingess 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?)
- need to balance correctness, latency and cost through watermark boundedness, allowed lateness and computing resources

4.2 Deployment

4.2.1 Infrastructure Considerations

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

https://ci.apache.org/projects/flink/flink-docs-release-1.11/ops/state/large_state_tuning.html#capacit
planning

when stream can be partitioned for parallelization, many small instances can work well, but also might require more shuffling

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

event time because IoT data often have high time skew

use completeness trigger and repeated update trigger for windows

value output mode

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

for general performance of flink, refer to ...

here, only evaluate our own solution

6.1.1 Latency Tracking

processing latency reasons:

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

configuration can tradeoff latency vs throughput

not the same as stream latency caused by waiting for watermark

data creation must not become bottleneck [40, p. 73]

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

average vehicle position message is 150 bytes as binary, 450 bytes with utf8 json

6.2 Results

describe cluster

flink 1.11

Kafka 2.5.0

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

evaluate event time latency for live events

use difference between event timestamp and egress timestamp

note that relies on proper time sync

also show Kafka backlog

6.2.2 Log Size

measure log size with json vs binary protobuf

6.3 Discussion

with checkpointing, fault tolerance is given, but free task slots must be available for failover

maybe s3 is bottleneck, otherwise store in hdfs

could not make emergency stop job work with rocksdb

somehow, checkpointing makes it work

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] 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`.

[4] 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.

[5] Yahoo Developer Network, *Hadoop Turns 10*, 2016. [Online]. Available: `https://developer.yahoo.com/blogs/138739227316` (visited on 08/11/2020).

[6] 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.

[7] M. J. Sax, G. Wang, M. Weidlich, and J.-C. Freytag, "Streams and Tables," in *Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics*, D. Chatziantoniou, Ed., ser. ACM Digital Library, New York, NY: ACM, 2018, pp. 1–10, ISBN: 9781450366076. DOI: `10.1145/3242153.3242155`.

[8] M. Fragkoulis, P. Carbone, V. Kalavri, and A. Katsifodimos, *A Survey on the Evolution of Stream Processing Systems*, 2020. [Online]. Available: `https://arxiv.org/pdf/2008.00842.pdf` (visited on 08/16/2020).

[9] 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.

[10] N. Marz, *History of Apache Storm and lessons learned*, 2014. [Online]. Available: http://nathanmarz.com/blog/history-of-apache-storm-and-lessons-learned.html (visited on 08/14/2020).

[11] ——, *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).

[12] N. Marz and J. Warren, *Big Data: Principles and best practices of scalable real-time data systems.* Shelter Island NY: Manning, 2015, ISBN: 9781617290343.

[13] J. Kreps, *Questioning the Lambda Architecture*, 2014. [Online]. Available: https://www.oreilly.com/radar/questioning-the-lambda-architecture/ (visited on 08/14/2020).

[14] Tyler Akidau, Alex Balikov, Kaya Bekiroglu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle, "MillWheel: Fault-Tolerant Stream Processing at Internet Scale," in *Very Large Data Bases*, 2013, pp. 734–746. DOI: 10.14778/2536222.2536229.

[15] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm, "A catalog of stream processing optimizations," *ACM Computing Surveys*, vol. 46, no. 4, pp. 1–34, 2014, ISSN: 0360-0300. DOI: 10.1145/2528412.

[16] M. Hirzel, G. Baudart, A. Bonifati, E. Della Valle, S. Sakr, and A. Akrivi Vlachou, "Stream Processing Languages in the Big Data Era," *ACM SIGMOD Record*, vol. 47, no. 2, pp. 29–40, 2018, ISSN: 0163-5808. DOI: 10.1145/3299887.3299892.

[17] G. Cormode, "Sketch Techniques for Massive Data," in *Synposes for Massive Data: Samples, Histograms, Wavelets and Sketches*, ser. Foundations and Trends in Databases, G. Cormode, M. Garofalakis, P. Haas, and C. Jermaine, Eds., NOW publishers, 2011. [Online]. Available: http://archive.dimacs.rutgers.edu/~graham/pubs/papers/sk.pdf.

[18] R. Adaikkalavan and S. Chakravarthy, "Seamless Event and Data Stream Processing: Reconciling Windows and Consumption Modes," in *Database Systems for Advanced Applications*, ser. Lecture Notes in Computer Science, J. X. Yu, M. H. Kim, and R. Unland, Eds., vol. 6587, Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 341–356, ISBN: 978-3-642-20148-6. DOI: 10.1007/978-3-642-20149-3{\textunderscore}26.

[19] M. Dayarathna and S. Perera, "Recent Advancements in Event Processing," *ACM Computing Surveys*, vol. 51, no. 2, pp. 1–36, 2018, ISSN: 0360-0300. DOI: 10.1145/3170432.

[20] S. H. Ahmed and S. Rani, "A hybrid approach, Smart Street use case and future aspects for Internet of Things in smart cities," *Future Generation Computer Systems*, vol. 79, pp. 941–951, 2018, ISSN: 0167739X. DOI: 10.1016/j.future.2017.08.054.

[21] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas, "State management in Apache Flink," *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1718–1729, 2017, ISSN: 21508097. DOI: `10.14778/3137765.3137777`.

[22] M. Kleppmann, *Making Sense of Stream Processing*, 1st edition. O'Reilly Media, Inc, 2016, ISBN: 9781491937280. [Online]. Available: `https://www.oreilly.com/data/free/files/stream-processing.pdf`.

[23] J. Kreps, N. Narkhede, J. Rao, *et al.*, "Kafka: A distributed messaging system for log processing," in *Proceedings of the NetDB*, vol. 11, 2011, pp. 1–7.

[24] M. Kleppmann, A. R. Beresford, and B. Svingen, "Online Event Processing: Achieving Consistency Where Distributed Transactions Have Failed," *Communications of the ACM*, vol. 62, no. 5, pp. 43–49, 2019, ISSN: 0001-0782. DOI: `10.1145/3312527`.

[25] G. Cugola and A. Margara, "Processing flows of information," *ACM Computing Surveys*, vol. 44, no. 3, pp. 1–62, 2012, ISSN: 0360-0300. DOI: `10.1145/2187671.2187677`.

[26] O. Etzion and P. Niblett, *Event processing in action.* Greenwich 74° w. long.: Manning, 2011, ISBN: 9781935182214. [Online]. Available: `http://www-di.inf.puc-rio.br/~endler/courses/RT-Analytics/transp/Books/Event%20Processing%20in%20Action.pdf` (visited on 08/17/2020).

[27] A. Hinze, K. Sachs, and A. Buchmann, "Event-based applications and enabling technologies," in *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, A. Gokhale and D. C. Schmidt, Eds., New York, N.Y.: ACM Press, 2009, p. 1, ISBN: 9781605586656. DOI: `10.1145/1619258.1619260`.

[28] M. Fowler, *What do you mean by "Event-Driven"?* 2017. [Online]. Available: `https://martinfowler.com/articles/201701-event-driven.html` (visited on 08/17/2020).

[29] Apache, *Apache Flink 1.11 Documentation: RabbitMQ Connector.* [Online]. Available: `https://ci.apache.org/projects/flink/flink-docs-release-1.11/dev/connectors/rabbitmq.html#rabbitmq-source` (visited on 08/18/2020).

[30] StreamNative, *Pulsar vs Kafka: From Use Cases and Community to Features and Performance*, StreamNative, Ed., 2020.

[31] Amazon Web Services, *What Is Amazon Kinesis Data Streams?* 2020. [Online]. Available: `https://docs.aws.amazon.com/streams/latest/dev/introduction.html` (visited on 08/18/2020).

[32] D. Nguyen, A. Luckow, E. Duffy, K. Kennedy, and A. Apon, "Evaluation of Highly Available Cloud Streaming Systems for Performance and Price," in *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, IEEE, 2018, pp. 360–363, ISBN: 978-1-5386-5815-4. DOI: `10.1109/CCGRID.2018.00056`.

[33] Apache, *Powered By Apache Kafka.* [Online]. Available: `https://kafka.apache.org/powered-by` (visited on 08/18/2020).

[34] M. Kleppmann, *Turning the database inside-out with Apache Samza*, 2015. [Online]. Available: `https://martin.kleppmann.com/2015/03/04/turning-the-database-inside-out.html` (visited on 08/18/2020).

[35] B. Svingen, *Publishing with Apache Kafka at The New York Times*, 2017. [Online]. Available: `https://open.nytimes.com/publishing-with-apache-kafka-at-the-new-york-times-7f0e3b7d2077` (visited on 08/18/2020).

[36] N. Narkhede, G. Shapira, and T. Palino, *Kafka: The Definitive Guide: Real-time data and stream processing at scale*, First edition. Sebastopol CA: O'Reilly Media, 2017, ISBN: 9781491936160.

[37] 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`.

[38] M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia, "Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark," in *Proceedings of the 2018 International Conference on Management of Data*, G. Das, Ed., ser. ACM Digital Library, New York, NY: ACM, 2018, pp. 601–613, ISBN: 9781450347037. DOI: `10.1145/3183713.3190664`.

[39] Hazelcast, "Hazelcast Jet Datasheet," [Online]. Available: `https://www2.hazelcast.com/l/30822/2017-02-05/bpdslz/30822/106931/Hazelcast_Jet_Datasheet.pdf`.

[40] R. Lu, G. Wu, B. Xie, and J. Hu, "Stream Bench: Towards Benchmarking Modern Distributed Stream Computing Frameworks," in *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, IEEE, 2014, pp. 69–78, ISBN: 978-1-4799-7881-6. DOI: `10.1109/UCC.2014.15`.

Glossary

allowed lateness

the maximum delay in processing time after the watermark that a late record may have without being discarded

at-least-once processing

the guarantee that every record of a dataset will be processed at least once, but possibly more often

at-most-once processing

the guarantee that every record of a dataset will be processed at most once, but possibly not at all

batch dataset

a bounded dataset that is complete at the time of processing

checkpointing

a technique for persisting in-memory state to a durable storage medium for fault tolerance

complex event

an event resulting from the relation of multiple primitive events

consistency

a property of data processing meaning that a job will produce the correct results, even in case of failures

data at rest

a table that shows data at a specific point in time

data in motion

a stream that shows the evolution of data over time

dataflow graph

a model for the directed data flow between sources, operators and sinks in a job with streams moving along the edges from upstream to downstream

dataset partition

a subset of a dataset, usually determined by a key

effectively-once processing

the guarantee that every record of a dataset will affect the result as if it were processed exactly once, even though it might actually be processed more often for failure recovery

event occurence

a single event instance of a specific event type

event pattern

a template that specifies one or more combinations of events to for pattern recognition

event time

the time domain in which events occur

event type

a specification of attributes for event occurences that share the same semantic intent and structure

event-based programming

also known as *event-driven architecture*, an interaction model between system components that uses events for communication

event-time skew

a measure of the skew of event time perception in the processing system, identical with processing-time lag

exactly-once processing

the guarantee that every record of a dataset will only be processed once, even in case of failures

geo-event

a primitive event with an associated geographical location

job

a specific processing task

late data

data with an event timestamp earlier than the watermark that arrive past the watermark

log

an immutable and durable append-only data structure with total order of records

log partition

a method for parallelizing logs where each partition itself is a log, multiple partitions of a log share no global order

materialized result

the result of a window evaluation that is emitted as stream record and sent to downstream consumers

micro-batching

a technique which decreases batch processing latency by decreasing batch size

node

an individual server in a cluster

offset

the position of a record in a log, monotonically increasing due to total order

operator

a transformation of a stream in the dataflow graph of a job

pattern recognition

a method for detecting subsets of events that match an event pattern specification

primitive event

an occurrence within a particular system, often just called *event*, can be a significant change or observation of some value

processing time

the time domain in which events are observed at a given processing stage, never before event time

processing-time lag

a measure of the delay between processing time and event time, identical to event-time skew

record

an element of a bounded or unbounded dataset

replay

the process of consuming a stream from an earlier point in time instead of starting with the latest record

stream

an unbounded dataset that will never be completed

stream analytics

any type of sophisticated stream processing that requires correct time handling and intermediate state

stream consumer

a component of a streaming system that receives an unbounded dataset

stream producer

a component of a streaming system that creates an unbounded dataset

stream transport

the component in a streaming system responsible for storing and transporting streams between producers and consumers

watermark

the timestamp of the point in event time up to which the system believes all inputs with lower event timestamps have been observed, a perfect or heuristic measure of completeness

window

a bounded subset of another bounded or unbounded dataset, often grouping records within a certain time period

window pane

the result of a single window evaluation of the same window with multiple triggers

Protobuf Definitions