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

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 (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 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 [8]. 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 [6, 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 [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]

[1] 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.

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 in motion*. 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 stream processing systems build on [6, pp. 174–212].

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 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.

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

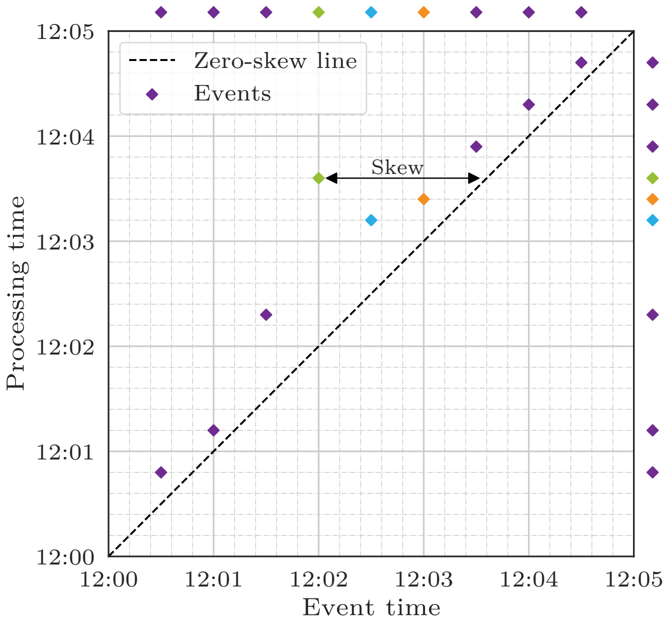

Figure 1: Relationship between event time and processing time: time skew varies a lot and leads to out-of-order arrival

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 [10, 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 (SEEP, Naiad, Microsoft StreamInsight, IBM Streams) have existed before [11, 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 [12].

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

FIGURE OF LAMBDA ARCHITECUTURE

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

FIGURE OF KAPPA ARCHITECUTURE

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

of the Lambda architecture [13], 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 [14, 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 [15].

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 replayable stream transport platform like Apache Kafka. This enables fault tolerance and recomputation in case of processing logic changes [15].

Spark Streaming [16] 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,

FIGURE OF BOUNDED AND UNBOUNDED DATASETS

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

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 [6, pp. 27–28]. Sophisticated time handling with high flexibility was explored in Google's company-internal MillWheel [17] framework and Dataflow [9] processing models. Apache Flink [11] 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 [11, 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 use case.

2.2.3 Processing Patterns

Streams require processing patterns that support their unbounded nature. There are three major categories: [6, 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.[2]

Approximation Approximation algorithms like sketches for frequency distribution or distinct-value queries [18] are optimized to handle large quantities of data by trad-

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

ing 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 recognition effectively builds a window for each stream record ending with the final record of the pattern, resulting in variable-length windows [19, 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 [19, 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. We will now regard these two challenges 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 [10, 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.

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

Figure 5: Common window types

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 5 [9, 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 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 [6, 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 6a. 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

(a) Processing time

(b) Event time

Figure 6: Windowing in different time domains

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 [6, 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.

Windowing in the event-time domain, as shown in figure 6b 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 [9] 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 [6, 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.

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 [6, 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 [6, 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 [6, 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 [6, 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,

	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

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.
- 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, latency and cost by adjusting trigger frequencies and output modes which also affects 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 [6, 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 [6, 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 times as event times. In some cases, the data source itself might produce watermarks. Late data, i.e. data with a 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 ordering within partitions or file growth rates [6, p. 66], but also as percentile watermarks [6, 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 [6, 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. 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 requires state. 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

(a)

(b) Processing time

(c) Event time

(d) Event time

Figure 7: Example of different processing semantics after failure recovery

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 [6, pp. 216–218]. Simply storing state externally in a database can become a bottleneck [21, pp. 1718–1719]. Compare this with 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. 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 [6, 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* [10, 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 7. 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 [6, 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 it 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 [6, pp. 155–156] or simply assuming deterministic transformations [21, p. 1722].

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

[22] 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

transfer of dataflow model to flink triggers/evictors

beam runner

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 of 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 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

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#monitor 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] 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).

[11] 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.

[12] N. Marz, *History of Apache Storm and lessons&nbsp;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).

[13] ——, *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).

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

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

[16] Apache, *Spark Streaming.* [Online]. Available: `https://spark.apache.org/streaming/` (visited on 08/14/2020).

[17] 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.

[18] 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`.

[19] 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`.

[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] 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