Understanding and Building Fault-Tolerant, Scalable and Fast Stream Analytics Solutions With Correctness Guarantees

Bachelor's Thesis / T3300

for the study program
Computer Science

at the
Baden-Wuerttemberg Cooperative State University Stuttgart

by
Dominik Stiller

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

Declaration of Authorship

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

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

Sindelfingen	Septermber 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

List of Figures

List of Tables

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 analytic is key to building a good solution.

2.1 Batch Processing

Historically, data was processed in form of bounded batch datasets. 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 processed 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 processed. Throughput, i.e. the amount of data processed per second, is a key metric since data volume is usually very large [1, p. 390].

As the volume of data grew, it 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 [2] (based on the MapReduce [3] programming model) and Spark [4]. These frameworks tackle two common challenges of large-scale batch processing [1, p. 429], [5, pp. 362–373]:

- Scalability: support for distributed processing across nodes requires orchestration and *partitioning*, i.e. the division of the data set 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) [1, 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 HDFS [2]. Therefore, the file system facilitates processing across multiple nodes.

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 [6]. 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 need to be divided into batches of a certain size for processing [1, 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 a few seconds, but ultra-low latency applications are still infeasible with micro-batch processing. This is especially true when considering that data might arrive with a delay, which usually requires deferred processing or re-proccessing when the late data arrive. Also, jobs that might span batch bounds, such as user session analysis in web applications, are inherently complex to implement [5, 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 data streams in mind, that can overcome and embrace stream characteristics to enable new types of applications?

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 continous data streams. This section is an introduction to streams and stream processing, showing the fundamental characteristics and challenges.

2.2.1 Streaming Data Properties

properties: unbounded, unordered, varying event time skew stream vs table

streams are natural for many data even many batch processing applications only use batch because of a lack of digitalization [7, p. 29] batch is easy to reason about completeness since it is bounded

concepts of event time and processing time analysis of data based on when they are observed as opposed to when they occur is usually not sufficient need to handle event time separately

event time and processing time often do not coincide example: delayed data (vehicle in tunnel, phones in airplane), fast-forward through historic data

2.2.2 Challenges

define 4 attributes

2.2.3 Stream Processing Architectures

Storm was first widespread stream system but tradedoff correctness for latency use lambda for correctness at first spark streaming as first large-scale stream processing engine with correctness guarantee for kappa architecture, nut only processing time

correctness needed for stream to get parity with batch, time gets you beyond batch [5, p. 28]

instead of chopping up natural streams as in batch processing, embrace characteristics (unbounded) and process stream continuously achieves much lower latency throughput may suffer, but future developments might help

> 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. [8, p. 1792]

batch can be processed with streaming system explanation of a true streaming use case [5, p. 386]

2.2.4 Processing Patterns

stream processing patterns [5, p. 35]:

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

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

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

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

2.2.5 Timely Processing

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

triggers define result materialization in processing time repeated or based on watermarks as measure of completeness

> Repeated update triggers are great for use cases in which we simply want periodic updates to our results over time and are fine with those updates converging toward correctness with no clear indication of when correctness is achieved. [5, p. 63]

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

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

processing time is natural, event time requires special techniques need to define measure of completeness to maximize correctness different methods for handling late data: retract old results, separate output, dismiss tradeoff completeness vs latency

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

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

2.2.6 Stateful Processing

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

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

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

2.3 Stream Transport

message queue, often ephemeral plain socket stream pubsub

2.3.1 Immutable logs

append-only immutable log with persistency

Reasons [7, 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

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

[9] shows performance

add feature list

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

implementation of dataflow model

3.2.1 Apache Flink

APIs

datastream, dataset, SQL

async queries

event time

unit testing

watermarking strategies

Cluster

workers and masters

task slots

high availability

Execution Model

tasks

operators

parallelism

co-location and operator chaining

shuffling after keyby

watermark propagation

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

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

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] M. Kleppmann, *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*, First edition. Boston: O'Reilly Media, 2017, ISBN: 9781449373320. [Online]. Available: `http://proquest.tech.safaribooksonline.de/9781491903063`.

[2] Apache, *Apache Spark*. [Online]. Available: `https://spark.apache.org/`.

[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] Apache, *Apache Hadoop*. [Online]. Available: `https://hadoop.apache.org/`.

[5] 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.

[6] Yahoo Developer Network, *Hadoop Turns 10*, 2016-02-05. [Online]. Available: `https://developer.yahoo.com/blogs/138739227316` (visited on 08/11/2020).

[7] J. Kreps, *I Heart logs*, First edition. Sebastopol CA: O'Reilly Media, 2014, ISBN: 978-1-491-90938-6.

[8] 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.

[9] 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

A Protobuf Definitions