# Scalable and Reliable Complex Event Processing on Event Streams

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

## 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

problem: Scalability + Stateful Processing (-> required for detection of complex events)

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

## 2.1 Batch Processing

batch processing for a couple of years

processing of complete, bounded dataset, e.g. once per day

many jobs have same structure, therefore mapreduce/hadoop enables fault tolerance and scalability with common programming model

Flume extended simple model with support for optimized complex pipelines

easy to reason about completeness since it is bounded

when node fails, simply restart on other nodes

latency usually not an issue

falls short e.g. with session windows

high latency

## 2.2 Stream Processing

### 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 [1, p. 29]

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 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 [2, p. 28]

instead of chopping up natural streams as in batch processing, embrace characteristics (unbounded) and process stream continuously

> 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, old data may be retracted, and the only way to make this problem tractable is via principled abstractions that allow the practitioner the choice of appropriate tradeoffs along the axes of interest: correctness, latency, and cost. [3, p. 1792]

batch can be processed with streaming system

explanation of a true streaming use case [2, p. 386]

### 2.2.3 Processing Patterns

stream processing patterns [2, 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)

### 2.2.4 Timely Processing

windowing as division of unbounded stream in bounded segments

can be arbitrary, but usually time or count

will focus on time, since count is effectively processing time

fixed, sliding, session, fixed is special case of sliding window

custom window using window assigner

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. [2, 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

### 2.2.5 Stateful Processing

[2, 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 [2, Chapter 5]

## 2.3 Apache Flink

### 2.3.1 APIs

datastream, dataset, SQL

async queries

event time

unit testing

### 2.3.2 Cluster

workers and masters

task slots

high availability

### 2.3.3 Execution Model

tasks

operators

parallelism

co-location and operator chaining

shuffling after keyby

watermark propagation

### 2.3.4 State

state backends

broadcast state

### 2.3.5 Checkpointing

barriers

aligned and unaligned

### 2.3.6 Network Stack

backpressure handling

flow control

latency vs throughput

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

## 2.4 Stream Transport

message queue (Rabbitmq), often ephemeral

plain socket stream

pubsub

### 2.4.1 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

### 2.4.2 Apache Kafka

present Kafka

commercial distributions like confluent provide tiered retention

## 2.5 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

### 2.5.1 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

### 2.5.2 Complex Event Processing

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 Proof-of-Concept Solution

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

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

Wanted to use real data

## 3.1 Use Case

### 3.1.1 HSL API Data

Available data

statistics

### 3.1.2 Analytics

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

### 3.1.3 Geoaggregation

Division in cells

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

## 3.2 Design

### 3.2.1 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

### 3.2.2 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

### 3.2.3 Ingestion

Extensible design with ingestors and processors

### 3.2.4 Flink Jobs

describe common functions (key selectors)

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
- caching for asnyc functions (show numbers, 5 req/s instead of 800 req/s)

### 3.2.5 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

### 3.2.6 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

## 3.3 Data Flow Example

## 3.4 Deployment

### 3.4.1 Server Considerations

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

### 3.4.2 Automation

immutable infrastructure (relevant?)

# 4 Evaluation

## 4.1 Results

### 4.1.1 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

### 4.1.2 Log Size

measure log size with json vs binary protobuf

## 4.2 Discussion

## 5 Conclusion

# Bibliography

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

[2] 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.

[3] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle, "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.

# A Protobuf Definitions