Benchmark Cloud-Optimized Complex Event Processing Solutions

Gerti Bushati
Technische Universität Berlin
Berlin, Germany
bushati@campus.tu-berlin.de

Mateusz Musiał
Technische Universität Berlin
Berlin, Germany
m.musial@tu-berlin.de

Anna Chester Serra
Technische Universität Berlin
Berlin, Germany
a.chester-serra@campus.tu-berlin.de

1 INTRODUCTION

Data is playing a central role in the world's economy and many businesses are trying to profit from data. Analyzing large amounts of data is crucial for many organizations to get meaningful insights, therefore deploying powerful analytical stream processing (ASP) and complex event processing (CEP) systems. Some popular ASP systems are Apache Spark [7], Storm [8], Kafka [4], and Flink [2], whereas popular CEP systems are Apache Flink CEP[3], Esper[12], Siddhi CEP[20]. Use cases of such systems include surveillance scenarios, for example monitoring traffic congestion, or fraud detection like detecting the usage of the same bank account on two different cities. Other examples include IoT scenarios, Network monitoring etc. In all these examples massive amounts of data are generated and a real-time analytics is needed. While the ASP approaches focus on providing real-time calculations on a distributed environment with low-latency, high-throughput, fault-tolerance and horizontal-scalability, the CEP systems focus on pattern recognition, matching, creation of complex events as well as triggering actions upon pattern recognition. They mostly run on a single machine without the possibility to scale out, being unable to provide the same guarantees as the ASP systems. To counter these problems, the database research community developed different approaches. In the first one, scenarios where ASP and CEP systems were fussed together emerged. Examples include the fusion of Apache Storm with Esper CEP, which led to the creation of Esper on Storm [1, 23], or the Stratio Decision [21] which fuses Apache Spark with Siddhi CEP. The seconds approach to counter the scalability problems of CEP systems is to provide build-in support for CEP operations on an ASP system. This approach was used by Flink which provides an official CEP library [3] or Kafka Streams CEP library which is an attempt to create build-in CEP functionality on top of Kafka Streams[5]. Finally, the third approach is to create a general operator mapping to translate the well-defined CEP operators into ASP operators [22]. Having different variations of the CEP systems, makes it difficult for organizations to choose the right one for their use cases. In addition,due to the absence of a widely accepted framework and terminology, there is not a standardized benchmark to evaluate these systems [14]. The goal of this project is to investigate and find key criteria and metrics to benchmark these CEP solutions. In summary we make the following contributions:

- We analyze the repositories of Esper on Storm and Stratio Decision
- We define the throughput and latency metrics to compare the performance between the systems
- We create an operator coverage table which shows the operators included in the different approaches
- We integrate a data generator which controls the distribution of matches accross workloads
- We extend the CEP2ASP approach with more patterns and integrate a latency logger to measure the event time latency in the pattern
- We implement Kafka Stream CEP and integrate some of the patterns to measure the difference with Flink CEP and CEP2ASP

The remainder of this report is organized as follows: In Chapter 2, we give background information needed to understand the rest of the report. We give a short introduction to ASP and scalable CEP, as well as the original goals of the project and needed adjustments. In Chapter 3, we gives details about our contributions to the project. We show the integration of the evaluation metrics, the integration and translation of the patterns for Kafka Stream CEP, the analysis of operators between the systems together with the event selection policies, the design of additional patterns and the data generator. In Chapter 4, we explain the experiment design and analyse the outcomes. Finally we conclude our findings in Chapter 5.

2 BACKGROUND

2.1 Analytical Stream Processing

Analytical stream processing (ASP) is a technology that focuses on analyzing real-time streams of events as they occur. Events refer to significant occurrences or observations that are captured and transmitted as data. They may come from different sources such as sensors, applications, logs, and many more. Analyzing such events in real-time or near real-time allows organizations to take immediate, well informed decisions and respond rapidly to events. ASP is used in different domain such as fraud detection, Iot, monitoring etc. Several tools and frameworks such as Apache Flink [2], Storm [8], Spark [7] and Kafka [4] have proven to be very effective for ASP by providing a distributed environment with low-latency, high-throughput, fault tolerance, and horizontal scalability [15].

2.2 Scalable Complex Event Processing

Complex event processing (CEP) is similar in nature with Analytical stream processing (ASP). Although they overlap in many features, they have different goals. While ASP focuses more on the

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/annachester/BenchmarkingCEP.
https://github.com/Gerti10/BDPRO_KafkaCEP.

calculations and processing of data in real time, CEP focuses on pattern recognition, matching, aggregation and creation of new events. Pattern matching consists not only on filtering, but of extracting properties from the event objects that can be combined in higher levels producing a new event stream output. In contrast with ASP, CEP frameworks offer also languages to define and query patterns of events. Moreover, the goal is to trigger actions whenever a pattern of interest is detected [9, 11]. Several CEP engines such as Esper [12] and Siddhi CEP [20] have proven to be effective on definition of patterns, however, they lack some important features from ASP engines such as horizontal scalability making them not suitable for large scale production environments. To overcome such problems the database research community tried to combine both technologies, i.e., to add CEP engines on the distributed environments of ASP. Such attempts resulted in creation of Esper on Storm [1, 13, 23] and Stratio Decision [21]. Esper on Strom is using Apache storm to efficiently distribute the events on particular worker nodes, which than use the Esper CEP to find the interesting patterns. Similarly Stratio Decision uses Apache Spark to distribute the events and places Siddhi CEP engine on the worker nodes.

2.3 Original Goals and Problems Encountered

The initial goals of the project was to investigate scalable complex event processing (CEP) systems, languages and define key-criteria and suitable metrics to benchmark these systems. The systems to benchmark were FlinkCEP [3], KafkaStreamsCEP [5], Stratio Decision [21], and Esper on Storm [13]. Note that from all systems only FlinkCEP is officially developed and maintained by Apache Software Foundation[6]. The other systems are research prototypes and lack on documentation or features. Even though the idea of integrating CEP engines in ASP environments was very promising, the attempts were not further researched and the prototypes created, were not maintained. Therefore, we encountered problems attempting to integrate and analyze them. The dependencies of Esper on Strom were not refreshed. Due to evolution of Apache Storm, the libraries are not compatible anymore and we had to drop the system from our benchmark. We had similar experiences also with Stratio Decision. Even though the code is available we did not have access to the documentation. Note that both GitHub repositories are achieved by their owners. Even though we tried to contact many project participants we did not get any answer and therefore we decided to drop Stratio Decision from our benchmark task.

2.4 Project Goals Adjustments

After dropping two systems from our benchmark task, we added a different approach to solve the scalability problem on the CEP engines. The approach is CEP2ASP and it maps the CEP operators to ASP operators showing how to solve the same CEP tasks using only the ASP engines. Early results indicate that the ASP approach can be faster than the CEP.

3 CONTRIBUTIONS

3.1 Integration of evaluation metrics

To ensure a robust system benchmarking process, the incorporation of appropriate evaluation metrics holds significant importance. Our investigation into identifying the most suitable metrics for assessing CEP resulted in the identification of two critical ones: Maximum sustainable throughput and latency. Regarding the throughput measurement, we leveraged the implementation from the original experiments for the CEP2ASP paper [22], making necessary refinements to suit our objectives. The latency measurement was developed and integrated from scratch. Additionally, an interesting paper we found, that discusses evaluation possibilities for this use case is "BiCEP - Benchmarking Complex Event Processing Systems" [10]. However, integrating further metrics was out of scope for this project. The details of the chosen evaluation metrics are outlined in this section.

Throughput. Regarding the throughput measurements, we focus on two critical aspects: determining the system's **maximum sustainable throughput** and assessing the **time required to attain a predefined target throughput**. By subjecting the system to different data ingestion rates, we aim to identify the upper limit of event traffic that it can consistently handle without exhibiting prolonged backpressure – resulting in increasing event-time latency. This sustainable throughput concept, as outlined in [16, p.10], encapsulates the system's robustness under diverse operational conditions.

Additionally, the goal is to understand how quickly the system can reach a specific desired throughput. This aspect is essential for grasping how well the system adjusts to and copes with different workloads. We measure the throughput as elements and megabytes processed per second. Overall, this offers a view of the system's performance, encompassing both its upper limits and its adaptability to a specific data load.

Latency. The second significant metric of interest is the latency. The goal is to evaluate the time it takes for the system to recognize and complete patterns based on the occurrence of events. We're only interested in events that contribute to a pattern match since all other events pass through the system super fast. Particularly important is the duration necessary for the last event to finalize the pattern's detection. The starting point is marked by the entry of an event generated by our data generator into the system. The endpoint marks the successful detection of a specific pattern. This detection process of a pattern relies on multiple event tuples, resulting in varying time intervals for pattern matching due to the time differences between these events. For the purpose of benchmarking the system's performance, we focus on the latency associated with the last event that concludes the pattern. The latency measurement captures the time difference between the moment the last event in a pattern is created at the data source and when the entire pattern, including that last event, finishes at the destination (sink). The implementation integrates a LatencyLogger which processes various timestamps tracked during the pattern detection process. The timestamps in question the following:

(1) ingestion-time (t_i): moment the data-point is ingested into the system, i.e. timestamp of creating java object.
(2) event-detection-time (t_e): moment the event completing a pattern is recognized as such.
(3) pattern-detection-time (t_p): moment the pattern match is terminated and saved. This moment is only very slightly after the event-detection-time and could be neglected.

After tracking these timestamps throughout the process, the latency l is calculated in the Latency Logger as follows: $l = t_p - t_i$. Additionally, we track the event-detection-latency l_e, which doesn't include this last time-gap between event-detection-time and pattern-detection-time, which can be neglected since it only includes the final collection of the detected pattern. It is calculated as follows: $l_e = t_e - t_i$ Note that none of these timestamps is to be confused with the actual event-time of the events, i.e. the timestamp in the CSV data file. This is not used for latency logging, only for windowing (20min) as described above. First, we tracked the latency for every matched pattern individually, but as soon as we scaled up our experiments, this resulted in enormous log files with too much information. Therefore, we changed to only one latency log per second. The implementation tracks the number of patterns matched in each second and the sum of latencies within that second. These values yield the average latency per second through a simple division. The latency logging per second, also makes it analogous to the throughput logging (also per second), which enabled the integration of both measurements into one data structure for simpler further processing and plotting.

3.2 Successful integration of KafkaStreamsCEP

In order to compare the results of our experiments we implemented the our patterns using the Kafka Stream CEP library [5]. The architecture is shown in Figure 1.

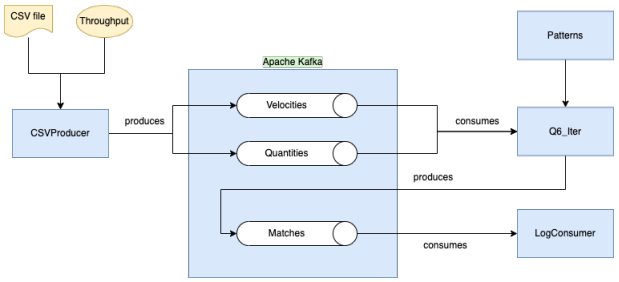

Figure 1: Kafka CEP Architecture

In difference with Flink which can natively read data from files, directories, sockets etc., and write to files, Kafka Streams can only read and write data from Kafka topics. Consequently we implemented a Topic Producer to create the input (velocities, quantities) and output (Matches) topics programmatically. Afterwards, we implemented a CSV Producer which reads data from a input CSV input file, serializes them into JSON format and publishes the events into velocity and quantity topics. Here, we also limit the data ingestion rate to be able to obtain the sustainable throughput. Having data published in the input topics we implemented different sequence and iteration patterns. Note that the library does not enable to correlate different types of events [17], therefore not supporting the integration of sequence patterns. However, it allows to read the events from different input topics. To this extent, we implemented a workaround by creating a general event type which contains the attributes of both velocity and quantity events, then published into their respective topics by adding a default value for the additional attributes. For each pattern we implemented the topology and created the output JSON structure which also contains the event time latency for the pattern. To extract the latency's and reduce the size of the measurements we implemented a log consumer which subscribes to the matches topic, extracts the latency measurements and aggregates them by summing the latency and counting the patterns found for each second. Finally the log consumer writes the collected data to a file for further analysis. The implementation of the project together with a guide on how to run it is available on GitHub . Note that the repository also contains the CEP library source files because the last version, which is also the stable version, is not in the Maven central repository. Also we tried to build the library and add it to our project dependencies, but it generated exceptions we were unable to solve.

3.3 Operator coverage analysis

The work of CEP2ASP [22], involves implementing basic patterns in FlinkCEP (referred to as Patterns) and successfully translating them into FlinkASP (referred to as Queries). Their contribution goes beyond operators such as selection, projection and windowing, which already have semantically equivalent counterparts in ASP (filter, map and time-based sliding windows). They introduce mappings for CEP operators, that differ from traditional ASP operators. Some of the operators in question are:

- **Sequence** (noted SEQ): The sequence operator is a binary temporal operator that expects the occurrence of two (or more) events in a specific order. This sequence should occur within a defined time window w.
- **Conjunction** (noted AND): The conjunction operator is a binary operation that checks the concurrent occurrence of two events, within the same time window w. This operator identifies instances where both events coincide.
- **Disjunction** (noted OR): The disjunction operator is a binary operation that checks the occurrence of either event e_1 or event e_2 within the time window w. At least one of the events takes place.
- **Iteration** (noted ITER): The iteration operator is a unary operation that allows for a specific number of event occurrences ($m > 0$) of a specific event type within a sequence, bounded by the exact occurrence of m events.

Their contribution includes a few other patterns (e.g. negation), but to remain in scope with our evaluation, we focus on the four aforementioned operators. It's worth noting that Conjunction and Disjunction lack native support in FlinkCEP, yet their implementation achieves analogous outcomes through join operations. Given our intention to benchmark against KafkaStreamsCEP, we need to gain insight into the supported patterns within that framework. To address this, we provide an analysis of operator coverage across the three systems. Our findings indicate that, of the mentioned patterns, KafkaStreamsCEP supports only the Sequence and Iteration patterns. Moreover, both FlinkCEP and KafkaStreamsCEP can match Sequence patterns with three distinct selection policies:

https://github.com/Gerti10/BDPRO_KafkaCEP

- **Strict Contiguity** (noted *SP*2): Expects all matching events to sequentially follow one another without intervening non-matching events.
- **Relaxed Contiguity** (noted *SP*1): Disregards non-matching events that occur between the matching ones.
- **Non-Deterministic Relaxed Contiguity** (noted *SP*0): Extends contiguity relaxation, permitting additional matches that overlook matching intermediate events.

These selection policies and their uses are explained in more detail in the corresponding documentations. For FlinkCEP it can be found in the section "Combining Patterns" [3] and for KafkaStreamsCEP in the section "Event Selection Strategies" [5].

The following table illustrates how these selection policies for Sequence patterns are implemented in both systems:

Table 1: Sequence selection policies

policy	FlinkCEP	KafkaCEP
SP0	.followedByAny()	.withSkipTilAnyMatch()
SP1	.followedBy()	.withSkipTilNextMatch()
SP2	.next()	.withStrictContiguity()

However, CEP2ASP solely supports the first selection policy—Non-Deterministic Relaxed Contiguity. To put all this information together, the subsequent operator coverage table provides an overview of which operators are supported by each of the three systems in question:

Table 2: Operator coverage table

	CEP2ASP	FlinkCEP	KafkaCEP
SEQ-SP0	✓	✓	✓
SEQ-SP1		✓	✓
SEQ-SP2		✓	✓
AND	✓	✓	
OR	✓	✓	
ITER	✓	✓	✓

In summary, only the Sequence pattern with the "non-deterministic relaxed contiguity" selection policy and the Iteration pattern are universally supported across all three frameworks. These two patterns are therefore used in our comparison of all systems. Sequence patterns across all three selection policies are used for contrasting FlinkCEP and KafkaStreamsCEP. Finally, in alignment with the original CEP2ASP experiments, we deploy Sequence (SP0), Conjunction, Disjunction, and Iteration patterns for the comparison of CEP2ASP and FlinkCEP, as in original experiments of CEP2ASP.

3.4 Design of additional Sequence patterns

For further investigation of the difference between FlinkCEP and KafkaStreamCEP we contribute an additional sequence pattern, since alongside the iteration pattern it is the main operator covered by both systems. The first original sequence pattern looks for a velocity event v above given threshold $v > velFilter$, along with a quantity event q exceeding another threshold $q > quaFilter$. When both events occur in order, the pattern matches. The second sequence pattern checks for two velocity events above a given threshold, followed by two quantity events below a given threshold. Additionally, the velocity event values should increase and the quantity event values should decrease. The events in the stream are such that they alternate between velocity and quantity events. All patterns can be looked up in the *newPatternCatalogue.md*. Both Sequence patterns were implemented for the three different selection policies that were mentioned before. This was done for both FlinkCEP and KafkaStremsCEP. The next section will focus on the details of the pattern implementations for KafkaStreamsCEP.

3.5 Translation of patterns to KafkaStreamsCEP

After successfully deploying KafkaStreamsCEP, the two sequence patterns and the iteration pattern were translated into this system. In the following, we outline how these patterns were implemented for KafkaStreams and how the implementation differs from the equivalent FlinkCEP implementation. In Kafka Stream CEP a complex pattern is created out of multiple stages. Each stage defines simple patterns which specity the conditions to select the events from the stream. The complex patterns are created by combining multiple stages together[5]. Figure 2 shows the implementation of an iteration pattern.

```
public static final Pattern<String, GeneralEvent> Q7ITER = new QueryBuilder<String, GeneralEvent>()
        .select( name: "stage-1") StageBuilder<String, GeneralEvent>
        .where((event, states) -> event.value().velocity > 175 ) PatternBuilder<String, GeneralEvent>
        .<~>fold( state: "timestamp", (k, v, curr) -> v.timestamp)
        .then() Pattern<String, GeneralEvent>
        .select( name: "stage-2", Selected.withSkipTilAnyMatch()) StageBuilder<String, GeneralEvent>
        .times(4) PredicateBuilder<String, GeneralEvent>
        .where((event, states) -> event.value().velocity > 175) PatternBuilder<String, GeneralEvent>
        .and((event, states) -> event.value().timestamp - (long)states.get("timestamp") < 20)
        .within( time: 1, TimeUnit.HOURS)
        .build();
```

Figure 2: Iteration Pattern

In the first stage the pattern chooses an event with velocity over the threshold and saves the timestamp of the event. Then, in the second stage the pattern chooses four events using the same condition and with timestamp difference, less that 20 milliseconds. For each stage we can define the selection policy to look for the next event and the number of the events that should match the iteration using the same condition defined in the stage. The within clause specifies the time window for the event using the timestamp associated with the event when published to the Kafka topic. However, in our case it is not useful, since we are interested in the event time and our generated data contain synthetic timestamps which do not match current time. Therefore we add the time constraints using the where clause.

3.6 Data generator for custom selectivities

Original data. The data Ariane used for her CEP2ASP system was QnV-Data, a real-world, publicly available dataset about sensor readings from traffic in Hessen, Germany [18]. Using real data gives us no control over the distribution of event matches across the workload (matches could be clumped in one area while there could be close to none matches in the rest of the workload). Since clumps and distribution anomalies in general could have an impact

on the performance of different CEP solutions the project mentor requested to create a dataset with uniform distribution of matches and have it tested. This required us to have a data generator.

Data pipeline recap. We start by having a CSV file with any number of rows and columns, some of the columns will be relevant, all of the rows will be relevant, and importantly one of the columns must be the timestamp for event windowing purposes. In our CEP solutions, every query/pattern corresponds to a Java class which then imports the relevant CEP packages. Such pattern class ingests the CSV, looks through the relevant columns, and finds matches. The matches are then output to an output file and the work is done.

Goals of the data generator. The overarching goal is to have big (>2GBs) CSVs with a uniform distribution of matches across them and with the desired selectivity (number of matches/number of rows). To achieve that our preferred strategy is to first divide our CSV into chunks of set size by modifying the timestamp attribute of each row and have a set number of matches in each chunk, making each chunk have the same selectivity as the whole workload. The logic of the timestamp chunking is as follows: every CEP pattern needs a window size argument (in minutes) which corresponds to this: for every event that is a part of a match, we only consider the follow-up events that complete the match as long as their timestamp (in milliseconds) is within the specified window. In our experiments, we uniformly use the window size of 20 minutes and by making the difference between the last member (chronologically) of the previous chunk and the first member of the next chunk >20 minutes (1200000 ms) we make sure the matches do not happen between the chunks.

Execution. We decided to use pandas as our tool for modifying the data due to its large community and since it handles large numbers of rows better than numpy. Pandas have dedicated methods that scale decently for dataframes that are multiple GB large. Possible other solutions that would scale better include using Flink or another stream processing engine as a tool for creating CSVs that are 10s to 100s gbs large or, since our workload is mostly uniform and repetitive, any tool that could efficiently copy and merge smaller CSVs or maybe even single dataframe chunks, preferably using multithreading.
We generate our data by creating an empty dataframe of desired size and using a dedicated pandas' "apply" function, which itself takes a function as an argument and then applies it to every row. In that argument function, we have access to the row processed and its row id allowing us to pinpoint its position in a chunk and doing the necessary modifications. With this solution the generation time for a 3gb CSV on the DIMA cluster was acceptable, allowing us to focus on the experiment part. The further improvements described above might make it even more comfortable if there is a need for it.

4 EXPERIMENTS

4.1 Design

Our design for the experimental evaluation includes the following experiments:

(1) Varying data ingestion rates
(2) selectivity experiments
(3) Parallel execution on multiple nodes

1. general experiments with varying data ingestion rates. The first experiment aims to detect how the systems behave when subjecting them to varying data ingestion rates. The goal is to detect the maximum sustainable throughput a system can reach (as described above) and how long it takes it to reach this sustainable throughput. For this experiment we keep the selectivity fix to 10%. For both FlinkCEP and CEP2ASP, this experiment is performed for the following patterns: AND, OR, ITER2, ITER2, SEQ1-SP0. Additionally, the experiments run exclusively for FlinkCEP for both SEQ patterns with different selection policies: SEQ1-SP0, SEQ1-SP1, SEQ1-SP2, SEQ2-SP0, SEQ2-SP1, SEQ2-SP2.

2. Selectivity experiments with custom data generator. The second experiment subjects the systems to varying selectivities while keeping the data ingestion rate fixed. Since our project shifted focus from further exploring the CEP engines we were looking for meaningful ways of benchmarking what we had and one way was to expand on Ariane's selectivity experiments, not just for SEQ1 but also for ITER2, AND, OR patterns/queries. On top of that we would be using our newly generated datasets which made sure that the matches were evenly distributed across the workload and not clumped in one area (except the skewed control variant). We achieved that by having a different dataset for each selectivity/query. This allows us to control the selectivity of each result and thus the query/pattern and selectivity (0.1%,1%,10%,10%-skewed) are our latent variables while the throughput is our observed variable.

3. Scalability experiments. The third experiment aims to explore the scalability of the systems. Due to the problems with the access to the cluster, they couldn't be performed yet. However, this section outlines the experimental design for this experiment. The plan was to try different combinations of settings in the $flink\_conf.yaml$ file. Given a desired parallelism for execution, the setup for distributing workers across nodes requires these three parameters to stand in the following relation: $parallelism.default = taskmanager.numberOf TaskSlots * taskmanager.numberOfTaskManagers$. For example, we can allocate 8 task slots (nodes) and have 8 task managers (workers) per task slot, which results in a total parallelism of 64. Details on suitable configurations for Flink parallel execution we found in this documentation. For our experiments, we aimed to try the following configurations for parallel execution:

	8	16	TaskSlots
2	16	32	
4	32	64	
8	64	128	
Workers			Parallelism

Figure 3: Parameter settings for flink_conf.yaml

Using the number of available CPUs as the default parallelism was reported to be a solid option. Since the dima cluster has 64

SEQ1, AND, OR all use 2 events for matching, ITER2 requires 5 events to complete a match
https://nightlies.apache.org/flink/flink-docs-release-1.2/setup/config.html#configuring-taskmanager-processing-slots

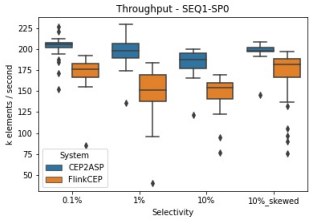

Figure 4: SEQ1 selectivity

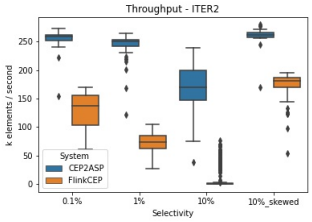

Figure 5: ITER2 selectivity

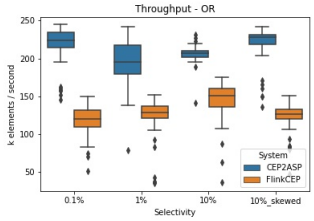

Figure 6: OR selectivity

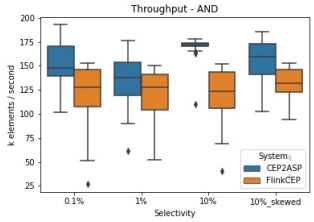

Figure 7: AND selectivity

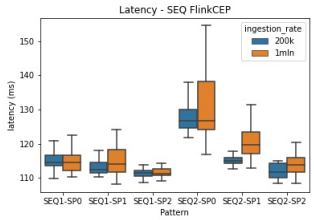

Figure 8: Latency SEQ1 and SEQ2

cores, we estimate that the example setting mentioned above would yield satisfactory results.

4.2 Execution

Original plan for cluster. Our experiments were designed to run on the cluster, we would generate our data sources for each query (3GB CSV for each) on the DIMA cluster in our directory and run our experiments using the high RAM resources available on the cluster. This was not possible due to the permission issues basically forcing us to have the project mentor run the experiments that we prepared for us. Due to the summer break a decision was made to run the experiments locally on a windows laptop using WSL2 to have some results for this project report. This is not however the end of the road since the project will be continued in Mateusz's thesis and we expect to see our results there.

Downscaling for local machine. The downscaling hindered in particular the scalability experiment since we were using localhost Flink while the experiment required different setups of TaskSlots, TaskManagers, and parallelism. We did not have a cluster infrastructure on a laptop hence we decided to drop the scalability experiment for now (it is however ready to be executed once the permission issues on the cluster are no more).
Since we were using a laptop with a limited amount of RAM (16GBs of which WSL2 was only using around 2GBs), we decided to also downscale our data source for each query, from 3GBs to 300MBs. Each query was on average not taking longer than 2 minutes to complete and while the startup/shutdown overhead for such runtimes is significant, we assume it to be similar enough between the runs to be able to compare them with one another.

4.3 Analysis

Selectivity experiments. Before talking about the results it needs to be repeated that these are the results from a localhost on a laptop, they might be useful in giving a general idea about the strengths and weaknesses of certain patterns on a framework, however, they are of limited use until replicated on the cluster (work in progress). As we can see on figures 3-6 the results are in line with the results in Ariane's paper as the throughputs of CEP2ASP are universally better than those of FlinkCEP. The improvements are much more profound in the OR and ITER2 while more similar in SEQ1 and AND. In Ariane's results increasing the selectivity had an enormous impact on the throughput in case of SEQ1, it is surprising to not see that replicated here, moreover, in general higher selectivity does not seem to impact the throughput much in all patterns/queries except ITER2 FlinkCEP. In that case increasing the selectivity results in a significant drop in the throughput to the point where at 10% selectivity it is unable to complete the job. Surprisingly enough, the same selectivity for FlinkCEP ITER2 but in a skewed workload, meaning all of the matches are concentrated in one half of the workload, the job is completed and throughput remains at acceptable levels. For other patterns/queries the skewed workload does not seem to impact the throughput much, only ITER2 CEP2ASP shows some improvement for 10%skewed over regular 10%.

General experiments. All plots regarding the general experiments can be found in the plots folder. Here we only display two summary plots that compare the performance of FlinkCEP for data ingestion

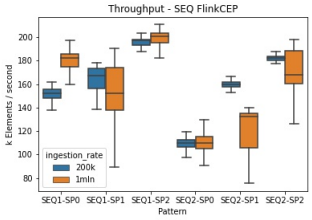

Figure 9: Throughput SEQ1 and SEQ2

rates 200k and 1 million tuples per second, regarding SEQ1 and SEQ2 for all three selection policies. We can observe that for high ingestion rate the latency becomes more unstable (wider ranges). Also, the throughput doesn't rise by increasing the ingestion rate as to be expected (for some it is even detrimental), which means that the maximum throughput for our local machine was already reached for ingestion rate 200k. As to be expected, higher throughput correlates with lower latency. We also observe that the system performs better the "stricter" the selection policy is. It struggles most for SEQ2 with selection policy non-deterministic relaxed contiguity. This seems like an interesting behaviour that should be further investigated on the cluster.

4.4 Experiments on Kafka Stream CEP

As already mentioned, we integrated both sequence patterns with different selection policies and iteration patterns in Kafka Stream CEP. The goal was to compare the results to ensure that both systems semantics is the same, find the sustainable throughput by limiting the data ingestion rate when publishing the events on the input topics, as well as measure the event time latency to compare with Flink CEP and CEP2ASP.

Result Comparison. We tested the patterns with small samples to ensure the same output for all three systems. We found that for both iteration patterns which are natively supported by the library the output is the same. For the sequence patterns the library does not give the smae output as Flink. The cause is that Kafka will read the event from the first topic and match it with events from the second topic therefore advancing the topic-offset for each message. Even though another event from the fist topic should be matched with event from second topic Kafka does not look back and will search only for unread messages to match.

Throughput and Latency. We run the experiments on a small scale (up to 200k records) to check the measurement of sustainable throughput and latency. Note that the topics are not partitioned for parallelism. When running the producer without limiting the throughput the event time latency increases steadily. The reason is that the messages are consumed in a much slower rate from the CEP than produced. By limiting the data ingestion rate we found that the sustainable throughput for the library was around 5000 records/second making it incomparable with Flink.
When running the experiments with a larger input data (300 MB) the CEP Library throws exceptions. We investigated the cause of exception and found that the CEP library creates three internal topics to save the state of the patterns, matchings and keep aggregations. When running the experiments the message-length in the state topic increases creating messages larger that the server will accept. We tried to increase the size of the message that the server will accept in kafka configuration, however without success. In conclusion the library is an acceptable solution to show that CEP operators can be integrated on top of Kafka Streams, but it is not suitable for production environments with large amounts of event.

4.5 Tools

For the analysis of the experimental evaluation, we provide a python script that plots overviews of results in a structured manner. The plots display and compare the experiment results, enabling deep insights into the performances of the systems. The first part of the plotting corresponds to the selectivity experiment and the second part to the general setup with varying rates and fixed selectivity. In general, we provide detailed plots that show the throughput or latency per second for each pattern, but also various summaries of these per-second-plots using boxplots. This enables the comparison of multiple variables at once. For creating the boxplots we use *seaborn* [19] library. It allows for easy setting which features should be displayed through which visual channels. For example: $sns.boxplot(data = pattern\_df, x = "Pattern", y = "Latency", hue = "System", orient = "v", order = patterns)$ This compares the latency (y-axis, box-plot) of both systems (hue) for all selectivities (x-axis) - concise in one plot. This principle of comparing variables is used in various manners for both experiments. A more detailed overview of all the provided plots can be found in the README.md file withing the "plotting" folder of the project. Additionally, we provide the python script used for data preprocessing. It reads the log files produced by the program, and creates pandas dataframes with suitable structure for plotting.

5 CONCLUSION

In this project, we investigated possibilities to benchmark scalable Complex Event Processing solutions. We explained the need for a standardized benchmark and defined latency and throughput to measure the performance of the CEP systems. We integrated Kafka Stream CEP to compare it with Flink CEP and CEP2ASP approach as well as a new data generator where we control the selectivity of the events. Due to problems encountered with Kafka Stream CEP library we were unable to measure the sustainable throughput and latency, therefore could not compare it with Flink. Moreover, due to problems encountered with the permissions on DIMA cluster we run the experiments locally. We found that the throughput of CEP2ASP approach is better than of Flink CEP. Moreover, we found that the selectivity seems to not effect the throughput of the systems. We note that the experiment behaviour may completely change when running on the cluster especially, since the pattern recognition in Flink CEP is performed on a single topology node limiting the ability of the system to scale the computation across different tasks.

Plotting.ipynb
data_preprocessing.ipynb

REFERENCES

[1] Ali SA Al-Haboobi, Abbas FH Alharan, and Radhwan HA Alsagheer. 2016. Experimenting with Storm/Esper Integration and Programmatic Generation of Storm Topologies. *International Journal of Computer Science and Information Security* 14, 8 (2016), 169.

[2] Apache Flink. 2011. https://flink.apache.org. Accessed: 2023-06-07.

[3] Apache Flink CEP. 2019. https://nightlies.apache.org/flink/flink-docs-master/docs/libs/cep/. Accessed: 2023-06-07.

[4] Apache Kafka. 2011. https://kafka.apache.org. Accessed: 2023-06-07.

[5] Apache Kafka CEP. 2016. https://github.com/fhussonnois/kafkastreams-cep. Accessed: 2023-06-07.

[6] Apache Software Foundation. 2016. https://www.apache.org. Accessed: 2023-06-07.

[7] Apache Spark. 2014. https://spark.apache.org. Accessed: 2023-06-07.

[8] Apache Storm. 2011. https://storm.apache.org. Accessed: 2023-06-07.

[9] Tim Bass. 2007. Mythbusters: Event Stream Processing versus Complex Event Processing. In *Proceedings of the 2007 Inaugural International Conference on Distributed Event-Based Systems* (Toronto, Ontario, Canada) *(DEBS '07)*. Association for Computing Machinery, New York, NY, USA, 1. https://doi.org/10.1145/1266894.1266896

[10] Pedro Bizarro. 2007. BiCEP - Benchmarking Complex Event Processing Systems. (01 2007).

[11] Gianpaolo Cugola and Alessandro Margara. 2012. Processing Flows of Information: From Data Stream to Complex Event Processing. *ACM Comput. Surv.* 44, 3, Article 15 (jun 2012), 62 pages. https://doi.org/10.1145/2187671.2187677

[12] Esper CEP. 2023. https://www.espertech.com. Accessed: 2023-05-27.

[13] Esper On Storm. 2012. https://github.com/tomdz/storm-esper. Accessed: 2023-06-07.

[14] Nikos Giatrakos, Alexander Artikis, Antonios Deligiannakis, and Minos Garofalakis. 2017. Complex event recognition in the big data era. *Proceedings of the VLDB Endowment* 10 (08 2017), 1996–1999. https://doi.org/10.14778/3137765.3137829

[15] Haruna Isah, Tariq Abughofa, Sazia Mahfuz, Dharmitha Ajerla, Farhana Zulkernine, and Shahzad Khan. 2019. A Survey of Distributed Data Stream Processing Frameworks. *IEEE Access* 7 (2019), 154300–154316. https://doi.org/10.1109/ACCESS.2019.2946884

[16] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. 2018. Benchmarking Distributed Stream Data Processing Systems. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 1507–1518. https://doi.org/10.1109/ICDE.2018.00169

[17] Samuele Langhi, Riccardo Tommasini, and Emanuele Della Valle. 2020. Extending Kafka Streams for Complex Event Recognition. 2190–2197. https://doi.org/10.1109/BigData50022.2020.9378217

[18] mCloud 2015. *MDM: Geschwindigkeitsdaten Hessen.* Retrieved Nov, 2021 from https://www.mcloud.de/web/guest/suche/-/results/filter/latest/provider%3AHessen+Mobil+-+Stra%C3%9Fen-+und+Verkehrsmanagement/0/detail/_mcloudde_mdmgeschwindigkeitsdatenhessen

[19] Michael Waskom. 2012. https://seaborn.pydata.org. Accessed: 2023-08-20.

[20] Siddhi CEP. 2023. https://siddhi.io. Accessed: 2023-06-07.

[21] Stratio Decision. 2016. https://github.com/Stratio/Decision. Accessed: 2023-06-07.

[22] Ariane Ziehn, Philipp M. Grulich, Steffen Zeuch, and Volker Markl. 2020. Bridging the Gap: Complex Event Processing on Stream Processing Engines. *Proceedings of the VLDB Endowment* 14, 1 (2020).

[23] Nikolaos Zygouras, Nikos Zacheilas, Vana Kalogeraki, Dermot Kinane, and Dimitrios Gunopulos. 2015. Insights on a scalable and dynamic traffic management system.. In *EDBT*. 653–664.