EasyFlinkCEP: Big Event Data Analytics for Everyone

Nikos Giatrakos
Athena Research Center
Technical University of Crete
ngiatrakos@athenarc.gr

Eleni Kougioumtzi
Athens University of Economics and Business
elkoumi15@gmail.com

Antonios Kontaxakis
Athena Research Center
Technical University of Crete
akontaxakis@athenarc.gr

Antonios Deligiannakis
Athena Research Center
Technical University of Crete
adeli@athenarc.gr

Yannis Kotidis
Athens University of Economics and Business
kotidis@aueb.gr

ABSTRACT

FlinkCEP is the Complex Event Processing (CEP) API of the Flink Big Data platform. The high expressive power of the language of FlinkCEP comes at the cost of cumbersome parameterization of the queried patterns, acting as a barrier for FlinkCEP’s adoption. Moreover, properly configuring a FlinkCEP program to run over a computer cluster requires advanced skills on modern hardware administration which non-expert programmers do not possess. In this work (i) we build a novel, logical CEP operator that receives CEP pattern queries in the form of extended regular expressions and seamlessly re-writes them to FlinkCEP programs, (ii) we build a CEP Optimizer that automatically decides good job configurations for these FlinkCEP programs. We also present an experimental evaluation which demonstrates the significant benefits of our approach.

CCS CONCEPTS

• Information systems → Query optimization; • Computer systems organization → Distributed architectures.

KEYWORDS

Big Data; FlinkCEP; Complex Event Processing; Optimizer

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM Reference Format:


1 INTRODUCTION

Complex Event Processing (CEP) enables analysts to express business rules as patterns and directly query incoming streams of data to detect these patterns [11, 12, 14]. For instance, in the maritime domain, CEP is utilized to detect illegal activities at sea based on vessel position streams [4, 5]. Or, in the financial sector, sequences of increasing, decreasing trend events on stock trade time series express price swing patterns [17, 18].

In the CEP context, rapid streams of data are processed online to timely report detected Complex business Events (CEs). CEs consist of simpler events in the form of singleton, looping, sequencing or other patterns with the use of quantifiers. CEP queries additionally include (a) contiguity conditions (a.k.a. selection strategies [12]) to determine whether we are allowed to ignore events of the incoming streams that are irrelevant to the monitored pattern or not, (b) event consumption policies, to denote how many matches an event may be assigned to, and (c) windowing operations [12, 14, 19]. To handle the volume and velocity of Big Streaming Data, Big Data platforms such as Apache Flink [9] have emerged. They focus on scaling-out the computation to a number of machines in a computer cluster/cloud, working in parallel on portions of the streams, to speed up continuous analytic outcomes. Flink includes a native API, namely FlinkCEP [10], for CEP analytics. FlinkCEP provides a CEP language of high expressive power [7, 15] and also allows for parallel processing to ensure rapid delivery of CEP analytics. Nonetheless, there are certain barriers in FlinkCEP’s adoption.

First, FlinkCEP requires business analysts, who know how to define business rules (i.e., the CEs to be monitored) but are usually non-expert programmers, to write code in Scala or Java. Second, pattern expression and parameterization involves cumbersome notations which makes the whole code writing process error-prone [12]. Third, submitting a FlinkCEP program to get executed as a job on a Flink cluster requires advanced skills on properly configuring and optimizing the usage of the available hardware resources which, more often than not, business event analysts do not possess.

To overcome the barrier posed by coding directly in FlinkCEP, we build a novel, logical CEP operator that receives as input CEP pattern queries in the form of extended regular expressions [1, 2, 6, 15, 22, 25] and seamlessly re-writes them to FlinkCEP programs. To offload domain experts from computer cluster administration decisions, we build a CEP Optimizer that automatically decides good configurations for executing the transformed FlinkCEP code on a computer cluster. We implement a prototype, namely EasyFlinkCEP, in the open-source platform of the INFORE project 1 and present experiments demonstrating the benefits of our approach.

1https://info-re-project.eu/; https://bitbucket.org/info_re_project/
https://github.com/eleniKougiou/Flink-cep-automation
Simple events workflows from day(s) to minutes. To understand why, compare with the same CE (iii) SKIP_PAST_LAST_EVENT: discard partial (P4), with Listing 1 which provides part of the FlinkCEP code in this is done by simply specifying (P1) a regular expression (Regex) the EasyFlinkCEP Operator [29] is a logical operator allowing Java querying for an equivalent pattern.

Optimizers automate such cluster configuration decisions. has to specify the computer cluster resources, i.e., parallelism, of EasyFlinkCEP's simple Regex created code the user specifications for (P2)-(P4).

Listing 1: Java code in FlinkCEP for detecting \( ab^+ \) patterns

```java
1  // -------- Create pattern "ab+" --------
3  @Override
4  public boolean filter (Event value){
5  return value.getName().startsWith( "a" );
6  }
7  }) .followedBy( "next" ).where( newSimpleCondition<>() { //followedBy for Relaxed Contiguity
8  }) .where( newSimpleCondition<>() { //followedBy for Relaxed Contiguity
9  }) .where( newSimpleCondition<>() { //followedBy for Relaxed Contiguity
10  }) .where( newSimpleCondition<>() { //followedBy for Relaxed Contiguity
11  })
```

3 THE EasyFlinkCEP OPTIMIZER

Challenges. FlinkCEP operators internally function on automata [7, 12, 16]. Briefly, these automata have simple events that triggers transitions from one state to another. Whenever an automaton reaches a final state, we say that a full match has been completed and a CE is outputted. But for an automaton that is in a non-final state, FlinkCEP has to keep and process all the partial matches that are valid based on the chosen contiguity, consumption and window specifications, since these may produce full matches for future streaming tuples. Therefore, FlinkCEP may have to process simultaneously multiple partial matches with each streaming tuple, which obviously affects Flink CE performance in terms of throughput, i.e. number of tuples being processed per time unit. Some relaxed contiguity strategies may produce a number of partial matches that grows exponentially to the number of input events [3, 31]. On the other hand, these partial matches may be reduced based on the specified window and consumption policy [12, 21]. Event consumption policies may also act as a bottleneck for a CEP operator, for example, halting its parallel execution to check if a given event has been consumed by other parallel instances of a pattern [12, 20].

Hence, it is hard to come up with an analytic formula to describe the expected performance of a CEP query and, therefore, the EasyFlinkCEP Optimizer treats a Regex as a black-box function attempting to learn its behavior—performance in terms of throughput for various parallelism degrees, contiguity, consumption and window specs. To do that, it encompasses the Benchmarking and the Cost Modeler Submodules (Figure 1).

The EasyFlinkCEP Benchmarking Submodule. When a previously unseen Regex, interpreted into a FlinkCEP program, is submitted, the Benchmarking Submodule of the EasyFlinkCEP Optimizer (Figure 1) performs a number of micro-benchmarking iterations, before actually deploying the CEP query in production. These micro-benchmarks focus only on the interpreted Regex and each runs a series of smaller scale FlinkCEP jobs, of limited duration each, testing the Regex under different sets of (Regex, contiguity, consumption, window, parallelism) specifications. Note that the submitted FlinkCEP program has fixed such specifications (excluding parallelism which is to be prescribed by the optimizer), while the Benchmarking Submodule runs jobs with different (contiguity, consumption, window) parameters. The aim is to learn Regex's expected throughput under different such specs and build a performance model that can be used any future time the same Regex is submitted, even if the exact specs are altered.

Micro-benchmarks may be executed over actual samples of data from the involved streams or on synthetic data streams produced...
The EasyFlinkCEP Cost Modeler Submodule.

Before \( D_{init} \) blue area shows uncertainty covering the whole range of throughput values.

(a) Before \( D_{init} \), blue area shows uncertainty covering the whole range of throughput values.

(b) Fitting GPR around micro-benchmarks in \( D_{init} \) (red lines). Blue line shows expected throughput.

(c) Acquisition function graph, \( \alpha(x) \) chooses next micro-benchmark \( x \) of high uncertainty.

(d) EasyFlinkCEP Cost Model: a GPR after a number of BO iterations - micro-benchmarks.

Figure 2: EasyFlinkCEP BO-based Optimizer Operation.

by data generators. The EasyFlinkCEP Optimizer collects statistics about the achieved throughput of each micro-benchmark using JMX [30]. The benchmarking process reserves cluster resources from other Flink jobs (only for a previously unseen Regex, though) and, thus, we want to keep these micro-benchmarks to a minimum.

The EasyFlinkCEP Cost Modeler Submodule. To sum up, we have a black-box function we wish to learn to estimate the throughput of the submitted (and any similar) FlinkCEP program under different parallelism configurations. We further want to accomplish that, using a limited number of benchmarks. This setup perfectly fits to a Bayesian Optimization (BO) approach [27, 28].

The first time we see a new Regex we know nothing about its expected throughput (Figure 2a). Therefore, the Benchmarking Submodule chooses a few \( x = (\text{contiguity, consumption, window, parallelism}) \) instances from the set of possible specification combinations and submits micro-benchmarks to the Flink cluster (right-hand side of Figure 1). Throughput statistics are monitored and are compiled in an initial benchmarking set \( D_{init} \) fed to the Cost Modeler upon the completion of the corresponding jobs. The outcome of the micro-benchmarks is a series of \( (\text{contiguity, consumption, window, parallelism, throughput}) \) tuples which will be used as training data. The BO approach used by the Cost Modeler builds an initial performance model fitting a Gaussian Process Regressor (GPR) around \( D_{init} \), as shown in Figure 2b. Then, the Cost Modeler uses an acquisition function \( a(x) \) in order to determine the (contiguity, consumption, window, parallelism) instance that should be used in the next micro-benchmark. The acquisition function determines the next micro-benchmark in order to improve the GPR in terms of describing the true throughput. Acquisition functions bargain exploration and exploitation. Exploitation means determining the next \( x \) point to be in areas where GPR predicts high/low values for throughput and exploitation means choosing \( x \) where the prediction uncertainty (e.g. variance) is high. Both correspond to high \( a(x) \) values and the goal is to optimize (maximize) \( a(x) \) to determine the next benchmark (Figure 2c). Commonly used Acquisition Functions are Expected Improvement (EI), Lower Confidence Bound (LCB), Maximum Probability of Improvement (MPI) or Entropy Search. After a number of such micro-benchmark submission and GPR update iterations, EasyFlinkCEP’s Cost Modeler ends up with an accurate GPR model that will be used in the future by the EasyFlinkCEP Optimizer every time the same Regex with any (contiguity, consumption, window) specifications is submitted (Figure 2d). Trained GPRs models are developed using the scikit-optimize [23] library.

When an interpreted FlinkCEP program with a specific (Regex, contiguity, consumption, window) is submitted, the EasyFlinkCEP Optimizer asks the trained GPR for a throughput estimation on \( x \) values composed of its (contiguity, consumption, window) specifications and various parallelism levels. For each queried parallelism level we have a different \( x \). So the GPR replies back with the throughput values on the blue line in Figure 2d crossed at these \( x \). Among these responses, the EasyFlinkCEP Optimizer will choose the level of parallelism that achieves the highest increase in throughput with the minimum number of provisioned resources, i.e., the lowest parallelism, and will configure the respective FlinkCEP job to run accordingly. Applications of this scheme are provided in Section 4.

EasyFlinkCEP & Best Practices. We list a set of best practices, applied in Section 4, extracted from EasyFlinkCEP’s adoption in the scenarios of Section 1.

1. \( D_{init} \) should be constructed by sampling with probability \(-10\%\) from all possible (contiguity, consumption, window, parallelism) combinations of the specification space.
2. Choosing LCB as an acquisition function results in GPRs that can better capture the trends in the benchmarked operator’s throughput, compared to EI or MPI, with a lower number of BO iterations.
3. The number of BO iterations required by LCB to establish accurate GPRs correspond to \( 1/3 \) of all possible (contiguity, consumption, window, parallelism) specifications.
4. Contiguity, consumption, and parallelism in the \( x \) axis of Figure 2 are categorical features and also window values can be easily discretized based on the number of tuples in the window or its time duration. Therefore, we suggest the GPR model to get parameterized with a weighted Hamming Distance Kernel using equal weights for each specification category.
5. The developed cost model can incrementally get updated for specific (Regex, contiguity, consumption, window, parallelism) with statistics from currently deployed CEP queries, avoiding the need for further micro-benchmarks.

4 EXPERIMENTS

In this section we provide experimental results on the EasyFlinkCEP Operator, the performance of the produced FlinkCEP programs and the ability of the EasyFlinkCEP Optimizer to devise good FlinkCEP job execution configurations. All experiments were run in a VM with 16 cores and 16GB of main memory running Ubuntu 18.04 using the latest version of Flink. The input and output streams were instantiated as Kafka topics using a broker running in the same machine. We used 8 stock streams with 2M simple events each.
(16M in total). Each stream was populated using events denoted as lower case latin characters. In this particular scenario, we monitor patterns involving stock trade price trends with the following notation: a: decreasing trend, b: increasing trend, c: steady trend and d: undetermined trend. We use a Regex of the form ab(1, 3)cd and a tumbler (i.e., disjoint) window of 128 events per stream after discussions with the domain experts. The default consumption policy was NO_SKIP, but we vary that parameter as well. To increase the complexity in terms of partial matches (Section 3), we further inject 125K CEs at random positions within the input streams.

In Figure 3a we plot the achieved throughput (processed input events/sec) of the interpreted FlinkCEP program as we vary the degree of parallelism in the EasyFlinkCEP operator and the desired contiguity condition. We notice that all instances of the operator scale by increasing the parallelism except for the execution of the strict contiguity with parallelism = 8, as in that case, the speed of the operator is capped by the rate of reading data from Kafka. As expected, selecting the relaxed or the non-deterministic contiguity conditions results in significantly smaller throughput due to increased number of partial pattern matches monitored before detecting CEs and the larger number of CEs produced (see Section 3).

In Figure 3b we repeat the experiment with parallelism = 1 and vary the contiguity condition and consumption policy. Using a consumption policy that skips events after a match benefits mainly the relaxed and non-deterministic contiguities, as it helps limit the number of partial matches monitored and CEs produced.

Our next set of experiments concentrates on the number of required micro-benchmarks and the effectiveness of the EasyFlinkCEP Optimizer in devising good job execution configurations. Figure 3c plots the prediction accuracy of the trained BO model. In the horizontal axis we include the number of executed micro-benchmarks which correspond to BO iterations. In the first vertical axis (left-hand side of Figure 3c) we measure the L1 error, then normalize the vertical axis to improve readability and include separate lines for the median and the average such error. In the second vertical axis (right-hand side of Figure 3c) we measure the $R^2$ score between BO predictions and actual throughput values. The concept is to use $L_1$ to deduce the prediction error in terms of absolute throughput values, i.e., $|\text{predictedThroughput} - \text{actualThroughput}|$, but also $R^2$ score to check the intuition that if the EasyFlinkCEP Optimizer (Cost Modeler in particular) can at least capture the trends of throughput, instead of the absolute values, it will still devise good job configurations. As Figure 3c demonstrates, the $L_1$ error drops below 5% after 20 BO iterations, while $R^2$ score exceeds 80% after, roughly, 10 micro-benchmarks – BO iterations.

To test our intuition, in Figure 4, we use a BO model trained with just 1/3 of the possible micro-benchmarks (red vertical line in Figure 3c where $R^2$ score > 80%) and we compare the optimal FlinkCEP configurations versus those prescribed by the EasyFlinkCEP Optimizer. The horizontal axis holds grouped triplets of all possible specifications regarding parallelism, consumption and contiguity for our Regex. The vertical axis plots the increase (percentage) in throughput upon increasing parallelism. We use two lines for the predicted throughput increase (predDiff) and the true one (trueDiff). The optimal job configuration is the one that achieves the highest increase in throughput with the lowest parallelism. So for instance, for the first triplet, the best FlinkCEP configuration under (Strict Contiguity, NO_SKIP) is to set parallelism to 2, since this parallelism provides the highest increase in throughput using the lowest possible resources (2 VMs or task slots etc). This is captured by EasyFlinkCEP since the (not necessarily the same) maximum for the corresponding predDiff, trueDiff lines is reached for the same value of parallelism. The same holds for the last two (8th and 9th) triplets under Non-Deterministic Contiguity and SKIP_TO_LAST, respectively.

As the marks in Figure 4 show, 8/9 times the EasyFlinkCEP Optimizer devises good job configurations. In 6 out of 9 possible consumption, contiguity specifications, the EasyFlinkCEP Optimizer configured the FlinkCEP job in the optimal way (✓ signs), 2/9 times it deviated the second best configuration (○ signs), while only 1/9 times (× sign) it reached a bad decision. Therefore, just capturing the throughput trends in terms of $R^2$ is proved sufficient to automatically devise good FlinkCEP job configurations, using just 1/3 (12/36) possible micro-benchmarks.

5 CONCLUSIONS

We present EasyFlinkCEP, a CEP component operating on top of a state-of-the-art parallel CEP engine, namely FlinkCEP. EasyFlinkCEP offers (i) the EasyFlinkCEP Operator which abstracts the details of coding directly on FlinkCEP, and (ii) the EasyFlinkCEP Optimizer to optimize FlinkCEP jobs executed in a Flink cluster of choice. Thus, business event analysts can focus on rapidly defining CEP queries and then exploit the processing capacity of modern hardware without any cluster administration background knowledge.

![Figure 3: EasyFlinkCEP Performance Study.](image)

![Figure 4: Job Config. Suggestions.](image)