Complex Event Processing with Proper Load Shedding

Quoc-Huy Trinh
Aalto University
Espoo, Finland
huy.trinh@aalto.fi

Anh Dao
Aalto University
Espoo, Finland
anh.d.dao@aalto.fi

Pablo Rubio
Aalto University
Espoo, Finland
pablo.rubiogarcia@aalto.fi

Abstract

Complex Event Processing (CEP) systems face significant challenges when processing high-velocity event streams, particularly when patterns involve Kleene closure operators that can cause exponential growth in partial matches. In this project, we implement and evaluate a state-based load shedding strategy for the OpenCEP framework to handle bursty workloads while maintaining low latency and acceptable recall. Through experiments, we tested our framework on the 2018 citibike dataset with a number of records about 10000 events. Our approach focuses on the CitiBike dataset, detecting hot path patterns where bicycles accumulate at specific stations. We implement intelligent partial match pruning based on query-specific utility functions that consider chain length, proximity to target stations, and remaining time in the detection window. Through comprehensive evaluation, we demonstrate that our load shedding strategy can maintain recall rates of 90-100% while keeping latency limits at the reasonable rate, effectively balancing system responsiveness with pattern detection quality. The implementation of this report can be viewed at https://github.com/huyquoctrinh/Scalablesys-Assignment1.

ACM Reference Format:
Quoc-Huy Trinh, Anh Dao, and Pablo Rubio. 2025. Complex Event Processing with Proper Load Shedding. In . ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Complex Event Processing (CEP) has emerged as a critical technology for real-time pattern detection in massive data streams across domains such as financial trading, IoT monitoring, and urban transportation systems. CEP systems continuously evaluate complex patterns over incoming events, assembling primitive events into higher-level complex events when predefined patterns are matched. Recently, several methods have been proposed to handle the CEP system, such as automata [1] or leveraging tree-based data structures [3] for the planning step and evaluator.

Although the previous works show the potential results while they can handle the extensive streaming data, they still have problems when dealing with bursty workload, when the extensive amount of the data arrives, along with the multiple data sources. The system maintaining responsiveness under bursty workloads presents a fundamental challenge: as event arrival rates fluctuate, the number of partial matches—incomplete pattern instances being tracked—can grow exponentially, particularly for patterns containing Kleene closure operators. To deal with this challenge, previous works such as Kafka, Heros propose the simple load shedding method to shed the events randomly, which can help to reduce the workload of the infrastructure during the bursty stream arrival. Following this, Gedik et al. [2], Tatbul et al. [4], and Wei et al. [5] propose the customization of the load shedding, which yields impressive results in the efficiency in both throughput and latency of the overall system.

Inspired by previous works, to tackle challenges of CEP systems, in this project, we address the problem by proposing the integration of the Adaptive load-shedding method along with state-management for CEP systems through integrating the adaptive load-shedding strategies along with the state management for the partial matches as the pruning strategies in the OpenCEP framework. Unlike traditional input-based load shedding that drops events before they enter the evaluation engine, our approach operates at the state management level, selectively pruning low-utility partial matches when the system approaches capacity constraints. This distinction is crucial for patterns with Kleene closure operators, where a single input event can spawn multiple partial matches within the evaluation tree.

In summary, our contributions include threefold:

- We implement the state management for the partial matches.
- We integrate adaptive semantic load shedding to handle low-latency streaming in bursty load conditions.
- We conduct several benchmarks to evaluate the effectiveness of the load shedding mechanism in the overall CEP system.

The rest of the report includes these sections: Section 2 presents the overall architecture of our system and our method, Section 3 illustrates our experimental setup and results. Finally, Section 4 concludes our main contributions anh results.

2 System Architecture

In this section, we will present in detail the components and the workflow of each module in our system. In general, there are six main modules in our overall framework. The first one is the I/O stream, which is used to stream the data, the second one is the statistic collector, to get the statistical information from the data. The third one is the optimizer to create the evaluation plan via the pattern query input. Following is the Evaluator, which will evaluate the partial matches and output to the output stream. Beside that, we also implement the State Management to manage the partial matches, and a load shedder to shed the data following the criteria, which help to balance the recall and the latency of the CEP system.

Conference'17, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2.1 OpenCEP

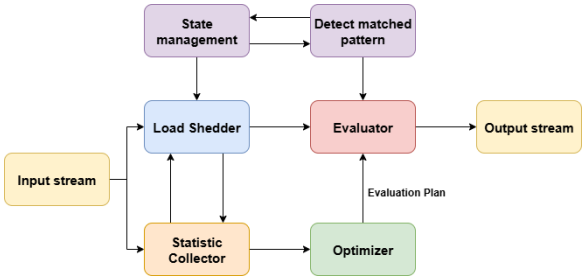

Figure 1: Overview of the OpenCEP framework

2.2 Streaming I/O

In order to simulate the streaming, in the OpenCEP, we implement the CSV streaming pipeline based on inheriting the I/O stream from the OpenCEP. Initially, the csv files are loaded, then small chunks from the CSV buffer are streamed to a queue. Afterward, the data from the queue is popped and handled by other modules in the CEP system before being pushed to the output stream to write to the text file buffer.

From our observation, we observe that the input stream can be slow due to the *append* method of the queue. Additionally, the complexity of the hot-path detection query can lead to an out-of-memory issue. For this reason, in our work, we also propose the integration of the deque as the main queue for the input stream. Thus, by leveraging the append and pop method with a time complexity of $O(1)$, we can fasten our streaming process, also we can manage the memory efficiently.

2.3 State Management

The purpose of the state management is to manage the partial matches in the system. However, in the bursty streaming environment, it can causes two main problems: the first one is the memory consumption, which will require the infrastructure more memory to store the partial matches, and the second problems is the time complexity in retrieving the partial matches from it. For this reason, in this block, we leverage the idea of the hash table for the partial matches storage via using the dictionary implementation. For the dictionary, the time complexity for retrieving is $O(1)$, while the space complexity will be approximately $O(n)$, which can enhance the efficiency of this module.

2.4 Adaptive Load Shedding strategies

We employ an *Adaptive load shedding* strategy to keep end-to-end detection latency under a user-specified budget while preserving the most useful work. Each incoming event is first transformed via normalized, and enriched with value featurers, including (*importance*, *priority*, *time_criticality*, *station_importance*, *bike_chain_potential*). When system pressure is detected (tail latency above budget and/or CPU/memory beyond thresholds), the admission gate sheds a fraction of the incoming batch, preferentially discarding low-value items. In detail, a linear score S_{event}, which is calculated by Equation 1, is used to calculate the importance of each partial match. Then, the list of partial matches is sorted (in $O(nlog(n))$ time complexity). Afterward, the lowest-scoring portion is dropped. Downstream, the query itself further prunes work via early target-station filtering, Kleene-adjacency constraints (same bike and end-to-start chaining), a maximum Kleene length, and lightweight deduplication/top-K of near-duplicate prefixes.

$$S_{\text{event}} = w_1 \cdot \text{importance} + w_2 \cdot \text{priority} + w_3 \cdot \mathbb{1}_{\text{rush\_hour}} + w_4 \cdot \text{station\_importance} + w_5 \cdot \text{chain\_potential}. \quad (1)$$

If the runtime supports state eviction, partial matches are also prioritized and, under memory pressure, evicted from the bottom of a value ordering that favors near-complete or promising chains. A representative state score S_{state} is calculated via the Equation 2.

$$S_{\text{state}} = a_1 \cdot \min\left(\frac{L}{L_{\max}}, 1\right) + a_2 \cdot \text{TTL} + a_3 \cdot \overline{S_{\text{event}}} + a_4 \cdot \mathbb{1}_{\text{near\_targets}} - a_5 \cdot \text{age}. \quad (2)$$

where L is the Kleene length, TTL the normalized time-to-live inside the 1h window, $\overline{S_{\text{event}}}$ an aggregate of the constituent events, and age is the state age. Eviction removes the lowest-S_{state} partials first, shrinking memory without sacrificing high-yield chains.
Shedding intensity is covered by a feedback controller. Let $e_t = \text{p90\_latency}_t - \text{budget}$ be the latency error at time t. The drop fraction p_t^{drop} is updated Equation 3

$$p_t^{\text{drop}} = \text{clip}\left(\beta\, p_{t-1}^{\text{drop}} + \alpha \cdot \max(0, e_t/\text{budget}),\ 0,\ p_{\max}\right). \quad (3)$$

with CPU/memory pressure modulating aggressiveness. A small fairness floor (uniform keep) avoids starving quiet keys, while ranking preserves rush-hour, target-proximal, and longer chains. The result is a graceful recall–latency trade-off: as the budget tightens, latency is stabilized by selectively removing the least valuable work first.

2.5 Statistic Collector

The statistics collector in our pipeline is a lightweight, passive probe that logs every detected match and computes run-level metrics without affecting execution. On each match, the add item interface adds records with $O(1)$ time complpexity for each sample (now $- b$.ts), increments a match counter, and writes the tuple ('RETURN', a_1.start, a_i.end, b.end) for offline evaluation. After the run, `run_once` derives wall time, engine time, throughput (EPS), and latency percentiles (p50/p90/p99) by sorting the collected latencies once (time $O(M \log M)$, space $O(M)$ for M matches); if exposed by the engine, it also records load-shedding diagnostics (events dropped, drop rate, average throughput). In addition, the evaluator parses the printed interface in $O(L)$ complexity and computes recall/precision via set intersections in $O(|\text{GT}| + |R|)$. These metrics populate reports/CSVs, quantify latency–throughput–recall trade-offs under different shedding hyperparameters, and explain behavior (e.g., rising drop rates), while keeping per-match overhead negligible.

2.6 Evaluator

Regarding the evaluator, we keep as the same original version of the OpenCEP. In the implementation, the Multi-Pattern Tree Based

mechanism of the OpenCEP framework is used to evaluate the partial matches and output it to the output stream.

2.7 Optimizer

The optimizer converts high-level CEP patterns into efficient execution plans. In practice you use it implicitly: submitted Patterns are compiled into TreePlans that the multi-pattern tree shares across queries for speed. Technologically, it is a rule-based planner (Python) drawing on standard CEP/DB techniques such as predicate pushdown, operator reordering, and window/negation handling, emitting a tree/DAG plan consumable by the evaluator. It integrates with the OpenCEP stack and is load-shedding aware, exposing hooks so the runtime can prioritize or evict state while preserving semantics.

3 Implementation

3.1 Experimental Setup

To evaluate the efficiency of our system, we conduct the experiments in the PC with I7 146650HX, which includes 24 vCPUs, and 32 GB RAM. For the codebase, we employ the implementation from the OpenCEP (in python) as our baseline for developing the state management, load shedding, and conducting the benchmark.

3.2 Query Test case

The pattern we evaluate is specified in the SASE query language, which is illustrated in the following Listing:

Listing 1: Query for the Hotpath Detection.

```
PATTERN SEQ(BikeTrip+ a[], BikeTrip b)
WHERE a[i+1].bike = a[i].bike
AND   b.end in S
AND   a[last].bike = b.bike
AND   a[i+1].start = a[i].end
WITHIN 1h
```

This pattern searches for chains of trips using the same bicycle (the Kleene closure `a[]+`), ending at target stations set S, included {519, 497, 402, 359, 435, 445, 3255, 490, 477, 514, 491, 426, 520, 3443, 281, 2006, 459, 368, 492, 523, 3165, 247, 358, 3163}. Due to the limitation in hot-path contained in the citibike dataset, we conduct the tested on several stations instead of three stations. The Kleene closure operator can match one or more consecutive trips, creating potentially unbounded numbers of partial matches as events arrive.

The goal of our query is to detect *hotpaths* as sequences of CitiBike trips by the same bike within a 1h window where the final trip b ends at a *target station set* T. We evaluate four patterns: a Kleene closure `SEQ(BikeTrip+ a[], BikeTrip b)` plus fixed 1/2/3-hop chains (`SEQ(a,b)`, `SEQ(a,c,b)`, `SEQ(a,c,d,b)`). The target filter on b (`b.end` $\in T$) is compiled as an OR-chain for pushdown.

3.3 Evaluation Metrics

To assess the performance of our system, we leverage the following metrics:

- Throughput: to measure the number of events our system can handle in one second, the unit for this measurement is EPS (events per second).
- Latency: to measure how long it takes for data to travel from a source to a destination and back. To do that, we calculate the latency at percentile at 99% (P99).
- CPU Utilization: We measure the amount of CPU utilization to show the resource consumption of this system.
- Recall: To measure the correctness of the hotpath detected.

3.4 Comparison Baselines

In our comparison, we conduct severeal experiments in three different scenarios. The first scenario is without load shedding and state-management inclusion. The second scenario is the benchmark with the

(1) **Changing the number of events**: To assess the difference in the throughput, latency, and recall under the different conditions of data stream ingests
(2) **Load Shedding Strategy**: We tried to make a comparison of the baseline when applying the Adaptive Load Shedding strategy, and in case it has no integration.

Unlike systems that prioritize absolute completeness, our design acknowledges the fundamental trade-off between latency and recall: under high load conditions, dropping some partial matches allows the system to maintain bounded latency while still detecting the most important patterns.

3.5 Dataset

We use the Citi Bike trip logs for New York City, October 2018. The cleaned CSV contains 1,878,433 trips, 10,850 unique bikes, and 777 distinct stations. Each record includes `tripduration`, start/stop timestamps, `bike_id`, `start_station_id`, `end_station_id`, coordinates, and user type. Our CSV formatter parses timestamps into engine time `ts`, normalizes identifiers (`bike_id`, `start_station_id`, `end_station_id`), and exposes only the attributes consumed by the pattern; the stream is processed in file order while the 1 h window uses the parsed timestamps. The month exhibits a clear diurnal profile: starts between 07:00–09:59 account for 19.97% of trips and 17:00–19:59 for 25.66% (45.63% combined). Demand is spatially skewed, with the top decile of stations concentrating 30.79% of starts and 31.22% of ends. Users are predominantly Subscribers (89.76%), and trip durations are short (median 599 s, mean 900 s).

For the hot-paths query we parameterize the target end-station set T from real identifiers present in the month; unless noted otherwise, $T = \{358, 3165, 247\}$, which receive 8,094, 5,252, and 2,743 inbound trips, respectively. Using the full month as input lets us cover peak and off-peak regimes, quantify the selectivity of `b.end_station_id` $\in T$, and observe the behavior of the Kleene closure under realistic temporal clustering and station imbalance.

3.6 Experimental results

Table 1 demonstrates our benchmark results on full of the citibile dataset. From the experimental results, it is shown that with the extensive records, our load shedding method can efficiency handle the low latency with high recall. This result shows that our method can effectively remove the partial matches that are not important, thus the correct matches are chosen, which makes a good recall evaluation. From this results, it also demonstrates the robustness and the efficiency of our method in the bursty workload. In addition,

Table 1: Overall Performance Comparison for the system in different numbers of the streamming events

Version	Number of Events	Throughput (EPS) ↑	Latency (ms) ↓	Recall ↑	CPU (%) ↓
Baseline	1000	248.6	3.86	100%	12.9%
Adaptive Load Shedding (threshold = 60ms)	1000	231.5	4.40	100%	17.6%
Baseline	10000	14.8	67.41	100%	31%
Adaptive Load Shedding (threshold = 60ms)	10000	16.6	60.04	96.4%	33%
Baseline	0.1M	8.6	124.02	100%	80%
Adaptive Load Shedding	0.1M	9.1	119.54	92.6%	74%

the extensive CPU utilization in the benchmark results also shows that our method needs to be improved for better integration in the lower-resource infrastructure.

In addition, from the benchmark table, it shows that, with the state management and adaptive load shedding integration, it is efficient for a large enough concurrent number of events in the stream. The reason for this assumption because with the small number of samples in the stream, the system does not need to drop any events to keep the latency; thus, without logic in score calculation and condition checking for CPU Utilization and Memory consumption, the naive approach become faster. However, in the bursty workload, due to the time for waiting and the event is processed, the naive approach needs to wait longer than the state management combined load-shedding improvement approach (because the number of events that need to be processed is reduced by dropping some of the partial matches), so the results of the improvement in the bursty workload would be better.

4 Conclusion

In Conclusion, in this project, we integrate the state management in multi-thread along with the adaptive load shedding system to optimize the Complex Event Processing system. Through experiments in a diverse number of event numbers, our system shows the effectiveness in latency reduction during the burststy workload. It demonstrates the efficiency of our systems across a diverse range of sample sizes in the data and various percentages of latency outbound, thereby showing the robustness of our system. Although the potential results of our system are promising, there are some limitations regarding the resource usage of our system, and another better hybrid load-shedding strategies should be employed to reduce the overall latency of our system.

5 Team contribution

This project was done by Quoc-Huy Trinh, Anh-Dao Dinh, and Pablo Rubio. During the implementation process, we set up triple programming once a week in the coffee shop, and we also have asynchronous meetings to discuss the idea and the architecture of the overall system. All of the implementations are committed on GitHub, and Quoc-Huy Trinh will merge all of the commits to construct our final codebase. During implementation, we tested different load-shedding methods, including a hybrid load-shedding approach, a semantic load-shedding approach, and an Adaptive load-shedding approach on 30% of the dataset before choosing the final method.

Regarding the role and implementation part for each member, we divide the role as follows:

- Quoc-Huy Trinh: Huy led the project, merged all implementations from Huy and team members to construct the final codebase. In the implementation, he handled the implementation for the input, output streams via CSV file in multiple threads, and improved by leveraging a deque data structure. Besides that, Huy also implemented the adaptive load-shedding method, and prepared all of the benchmarks in this project. During implementation, Huy faced difficulties in scaling the system as the inefficiency of the OpenCEP, but by leveraging the knowledge in data structure and algorithm, he understood the codebase flow and tried to reduce the time-complexity of some modules in the framework.
- Anh Dao: Anh did the implementation in the semantic load-shedding, hybrid load-shedding, and the state-management part. The most challenging in his work is the way to handle intensive streaming data from several data sources, as it is streamed in multiple threads, and he had to find a way to implement the state-management with shared data in multiple threads. Fortunately, he succeeded in dealing with that problem.
- Pablo Rubio: Pablo led the part of implementing the hot-path detection pattern query, analyzing the dataset, and trying to optimize the query. Because of the high time complexity of the KleenClosure operator, he faced several difficulties in testing with huge numbers of samples as it leads to the out-of-memory problem. However, after reading carefully the implementation of the operator, he could modify it and eventually optimized the time complexity of the query, thus facilitated for team to scale up the pipeline on the large amount of data.

References

[1] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. 2008. Efficient pattern matching over event streams. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 147–160.

[2] Bugra Gedik, Kun-Lung Wu, and S Yu Philip. 2008. Efficient construction of compact shedding filters for data stream processing. In *2008 IEEE 24th International Conference on Data Engineering*. IEEE, 396–405.

[3] Yuan Mei and Samuel Madden. 2009. Zstream: a cost-based query processor for adaptively detecting composite events. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. 193–206.

[4] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. 2003. Load shedding in a data stream manager. In *Proceedings 2003 vldb conference*. Elsevier, 309–320.

[5] Mingzhu Wei, Elke A Rundensteiner, and Murali Mani. 2010. Achieving high output quality under limited resources through structure-based spilling in XML streams. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 1267–1278.