Enabling Complex Event Processing in NebulaStream

Ariane Ziehn
BIFOLD, TU Berlin
Berlin, Germany
ariane.ziehn@tu-berlin.de

Lily Seidl
BIFOLD, TU Berlin
Berlin, Germany
l.seidl@tu-berlin.de

Samira Akili
BIFOLD, TU Berlin
Berlin, Germany
samira.akili@tu-berlin.de

Steffen Zeuch
BIFOLD, TU Berlin
Berlin, Germany
steffen.zeuch@tu-berlin.de

Volker Markl
BIFOLD, TU Berlin
Berlin, Germany
volker.markl@tu-berlin.de

ABSTRACT

Complex Event Processing (CEP) and Analytical Stream Processing (ASP) are two dominant paradigms for extracting knowledge from unbounded data streams. While CEP functionality is essential for detecting interesting patterns in vast data volumes, traditional CEP systems often face scalability limitations. To address these limitations, state-of-the-art solutions piggyback on cloud-optimized ASP systems for enhanced scalability and performance. The most common solution embeds CEP functionality as a single unary operator within the ASP execution pipeline. However, this design introduces conceptual bottlenecks, hindering the full utilization of ASP optimizations. To tackle this, we analyzed the synergies between both paradigms and proposed a general operator mapping in recent work. Our mapping translates CEP operators into their ASP counterparts, overcoming the bottlenecks of the unary operator solution. In this demonstration, we integrate our mapping with a declarative pattern specification language tailored to the requirements of ASP systems. This integration automates the mapping process, seamlessly translating high-level pattern definitions into optimized ASP query plans. Our demonstration showcases this approach within the ASP system NebulaStream. It allows the audience to submit declarative CEP patterns via its UI and explore the corresponding query plans resulting from our mapping. Additionally, we guide the audience through key optimization opportunities enabled by our mapping, which are unattainable with the unary operator solution.

1 INTRODUCTION

Analytical Stream Processing (ASP) and Complex Event Processing (CEP) are two common stream processing paradigms for extracting knowledge from unbounded data streams. ASP systems primarily focus on data transformation, employing operations such as aggregation and filtering to derive insights. In contrast, CEP systems are designed to identify intricate patterns within data streams. Specifically, CEP allows users to define patterns that connect events (i.e., time-stamped tuples) of multiple streams through temporal and causal relationships [4]. During execution, the system identifies and returns matches for these patterns, highlighting interesting behaviors in the data. These matches represent pre-interpreted results and thus allow for instantaneous automated actions, making CEP particularly valuable for large-scale monitoring and decision-making applications. While ASP systems are optimized for cloud environments and provide scalable solutions to handle the ever-growing volume of streaming data, CEP systems remain limited in their ability to fully utilize the cloud [4]. To address these scalability limitations, state-of-the-art approaches integrate CEP functionality into cloud-optimized ASP systems [8]. Representatives include KafkaStreamCEP[1] and FlinkCEP[2]. These systems leverage cloud-optimized ASP operators for efficient data gathering and pre-processing. However, the integration of CEP is typically implemented as a unary operator that encapsulates an entire pattern, leading to the following three performance limitations:

(1) Compute-Intensive Design: The unary operator, often as complex as a multi-way join, incurs significant computational overhead and a large state [6].

(2) Order-Based Evaluation Mechanism: The unary operator relies on a traditional order-based evaluation mechanism, i.e., a nondeterministic finite automaton (NFA). Consequently, it executes the pattern in the user-defined order, thereby bypassing optimization techniques inherent to ASP systems.

(3) Stream Union Overhead: The unary operator requires merging all involved streams into a single input stream. Thus, unnecessary events are processed, increasing the state size and further burdening state management.

To address these limitations, we recently proposed translating CEP patterns into ASP operators based on an operator mapping [12]. Our mapping addresses Limitations (1) and (2) by decomposing complex patterns into smaller subtasks that can be reordered to leverage optimization strategies, such as join ordering and filter extraction. Limitation (3) is addressed by leveraging the lazy evaluation inherent in ASP systems. Instead of merging and processing all streams simultaneously in a single operator, the execution is structured as a left-deep join tree. Thus, each operator processes data sequentially only when needed. This reduces unnecessary computations and significantly improves state management by limiting the size of intermediate results. As a result, our mapping enables ASP systems to efficiently execute CEP patterns and enhances the synergy between ASP and CEP, surpassing the capabilities of the unary operator approach.

In this demonstration, we present the implementation of our mapping in the beta-released ASP system NebulaStream[3] [11]. To this end, we have extended NebulaStream with a declarative pattern specification language (PSL), a parser, and two window join types, i.e., cross and interval join. We showcase our approach through the NebulaStream UI, which allows attendees to submit declarative patterns and visualize query plans and results. With this demonstration, we pave the way for seamlessly integrating CEP functionality in ASP systems while unlocking optimization opportunities for efficient pattern detection.

[1]https://github.com/fhussonnois/kafkastreams-cep
[2]https://nightlies.apache.org/flink/flink-docs-release-1.20/docs/libs/cep/
[3]https://www.nebula.stream

Table 1: Formal Definition of SEA Operators and their mapping.

Operator	Formal Definition	Mapping	Syntax
Conjunction	$(S_1 \wedge S_2)^* = \{(e_1, e_2) \mid e_1 \in S_1 \ \wedge \ e_2 \in S_2\}$	$S_1 \times S_2$	S_1 AND S_2
Sequence	$(S_1; S_2)^* := \{(e_1, e_2) \mid e_1 \in S_1 \ \wedge \ e_2 \in S_2 \wedge ts_{e_1} < ts_{e_2}\}$	$S_1 \bowtie_\theta S_2$	S_1 SEQ S_2
Disjunction	$(S_1 \vee S_2)^* := \{e \mid e \in S_1 \ \vee \ e \in S_2\}$	$S_1 \cup S_2$	S_1 OR S_2
Negated Sequence	$(S_1; \neg(S_2); S_3)^* := \{(e_1, e_3) \mid e_1 \in S_1 \ \wedge \ e_3 \in S_3 \wedge (ts_{e_1} < ts_{e_3}) \wedge \neg\exists ts_{e_2} \in [ts_{e_1}, ts_{e_3}] : e_2 \in S_2\}$	UDF$(S_1 \cup S_2) \bowtie_\theta S_3$	S_1 SEQ NOT S_2 SEQ S_3
Iteration	$(S^m)^* = \{e_n \mid e_i \in S, 1 \leq i \leq m \wedge (ts_{e_1} < ts_{e_2} < ... < ts_{e_m}) \wedge AGG(e_i)\}$	$S^1 \bowtie_\theta ... \bowtie_\theta S^m$	$S[m]$

The purpose of this demonstration is three-fold:

- We demonstrate how our mapping is seamlessly integrated into a general-purpose ASP system by highlighting the advantage of unifying CEP and ASP paradigms in a single system. Attendees can submit patterns and queries to explore this practical utility firsthand.
- We showcase the necessary adaptations for a declarative PSL in common ASP systems and discuss alternative directions and open issues with the attendees.
- We show the effectiveness of the enabled optimizations and reveal future directions by letting the audience explore how different data and pattern characteristics influence the query plan generation and its performance.

2 GENERAL OPERATOR MAPPING

Our previously introduced operator mapping [12] bridges the gap between common CEP operators and their ASP counterparts. For this mapping, we rely on the core CEP operator set defined by the Simple Event Algebra (SEA) [4], which includes selection, projection, window, conjunction, sequence, disjunction, negated sequence, and iteration. We formally defined these operators based on their verbal descriptions and the literature from related research lines [2]. To ensure compatibility with ASP systems, we adapted these operator semantics to (1) reach closure properties to enable seamless composition of operators, and (2) incorporate explicit windowing for all stateful CEP operators, a prerequisite for mapping to stateful ASP operators. In particular, the window operator in SEA is a time-based predicate that defines the maximal time interval $W = |ts_{e_2} - ts_{e_1}|$ in which all event pairs (e_1, e_2) of the pattern need to occur to form a match [4, 6]. Its semantics resemble an ASP sliding window with a slide per tuple [4, 12], and serve as a foundation for the integration of CEP operations.

Using our refined semantics, we identified five SEA operators that diverge from the native ASP operator set and mapped them to their respective counterparts. In the following, we briefly introduce these operators alongside their mapping. Table 1 summarizes our final operator semantics and the respective mappings.

Conjunction: The binary conjunction operator expects a pair of events, one from each stream S_i, i.e., $e_1 \in S_1$ and $e_2 \in S_2$, to occur within a window W. Based on its formal definition, a conjunction is equivalent to a relational cross join $\times$ (Cartesian product) [3], which composes two streams into one as a set of pairs. Each pair (e_1, e_2) is a pattern match.

Sequence: The binary sequence operator expects a pair of events $(e_1 \in S_1, e_2 \in S_2)$ to occur in temporal order, i.e., $ts_{e_1} < ts_{e_2}$, within a window W. Its formal definition is equivalent to the relational theta join $\bowtie_\theta$ using the order by time as join predicate θ [3]. In particular, all event pairs (e_1, e_2) fulfilling the condition of consecutive timestamps are a pattern match.

Disjunction: The binary disjunction operator requires at least one event from the specified streams S_1, S_2 to occur within a window W. Its formal definition is equivalent to the relational set union operator $\cup$ [3, 4]. The union operator merges two input streams into a new one, i.e., S_1 and S_2 are unified to $S_{1,2}$. Each event $e_{1,2}$ in $S_{1,2}$ is a match of the pattern.

Iteration: The iteration operator captures multiple occurrences m ($m > 0$) of events from a single stream S in a temporal sequence. Unlike Kleene* and Kleene$^+$, the SEA iteration operator enforces a bounded number of m occurrences [4]. Its formal semantics equals a nested sequence over a single event type S. Thus, the iteration is mapped to a sequence of m theta self joins $\bowtie_\theta$ using the order time constraint between consecutive event pairs as the join predicate θ [3].

Negated Sequence: The ternary negated sequence operator requires the absence of any events $e_2 \in S_2$ between a match of the sequence $(e_1 \in S_1, e_3 \in S_3)$. Unlike a negated predicate, the negation operator does not rely on specific attribute values but rather on the non-occurrence of the event itself. Its formal definition represents the combination of a sequence, i.e., $(S_1; S_3)$, and the negated existential quantifier that requires the absence of any event $e_2 \in S_2$ within the time interval $(e_1.ts, e_3.ts)$. Thus, we refer to the mapping of the sequence and add the negated quantifier as a sub-query.

By providing a semantically equivalent mapping for all operators, our mapping enables general-purpose ASP systems to execute CEP patterns as queries that leverage cloud-optimized ASP operators. By utilizing set operations, our mapping supports pattern detection under the least selective selection policy *skip-till-any-match*, while other policies can be implemented by applying additional filtering on the matches.

3 IMPLEMENTATION

We integrate our operator mapping with a declarative PSL into the ASP system NebulaStream [11]. NebulaStream is a novel, general-purpose, end-to-end data processing system that combines the advantages of fog and cloud environments to manage the vast amount of data generated by geographically distributed IoT devices on the way to the cloud. Enhancing NebulaStream with CEP features allows domain experts in areas such as traffic, air pollution, and water management to efficiently monitor and analyze massive streams of high-frequency data in a timely manner. In the remainder, we briefly introduce our PSL, highlight the optimizations achieved by integrating our mapping into NebulaStream, and describe the necessary extensions.

Pattern Specification Language. In contrast to many cloud-based CEP solutions that rely solely on low-level programming APIs [4], our CEP integration is based on a declarative PSL. Using a declarative language simplifies the transition for non-programming domain experts to more scalable solutions, thereby enhancing the systems' accessibility and usability. To achieve this, we designed a dedicated grammar for our PSL, prioritizing a syntax that aligns closely with existing SQL-like PSLs [5, 9]. At the same time, we integrated necessary adaptations to meet ASP system constraints, e.g., the definition of sinks or different window types, and strike a balance with SQL compliance. In particular, our approach aims to reduce inconsistencies, such as variations in expressing conditions or the usage of keywords, ensuring a more intuitive experience in a system that integrates both processing paradigms. Alternative directions to include our mapping are programming APIs [4] or optimizing for compatibility with a dedicated CEP language, e.g., SASE+ [5] or ZStream [9].

Our PSL supports the key operators discussed in Section 3, with their syntax outlined in Table 1. Additionally, it features a concise set of clauses: **PATTERN** specifies the pattern name, involved event streams, and their causal and temporal relationships; **FROM** identifies and optionally renames the streams upon which a respective pattern should be matched; **WHERE** applies conditions to stream attributes; **SELECT** defines the output structure; **WITHIN** sets window type and parameters; and **INTO** specifies the sink(s) for the result stream. We present the general structure of our PSL in Listing 1, showing how it integrates these clauses into an accessible and user-friendly syntax.

Listing 1: General structure.

```
PATTERN NAME SEP <event compositions>
FROM <set of streams [with renaming]>
[WHERE <set of predicates>]
[WITHIN <window W with type and parameters>]
[SELECT <output specifications>]
INTO <set of sinks>
```

Optimizations. Our mapping offers various angles for optimization, which can be leveraged depending on the workload characteristics and capabilities of the underlying ASP system. In our case, NebulaStream provides a rich set of streaming operators, including all traditional streaming ETL operators, and relevant features for our mapping, i.e., temporal aggregations, equi join with sliding windows in event-time and union. For this demonstration, we focus on optimizing the join order and window join type selection, which we identified in our prior work as one of the most impactful strategies for improving performance.

Window Join Reordering: Unlike order-based evaluation mechanisms, which impose strict sequential processing constraints, our mapping towards window joins allows for reordering and thus can improve performance and resource utilization. Join order optimization targets patterns involving multiple conjunctions and sequences, which are translated into multi-way window joins. In our PSL, users can specify the join order through the pattern, allowing them to suggest an efficient execution strategy. While we rely on manual join order selection, future work can automate our rewriting rules based on statistics, such as the rates and selectivities, to determine an optimal join order. By reordering the joins, the system can prioritize the most selective conditions earlier in the query execution plan, significantly reducing intermediate result sizes and improving overall processing efficiency.

Window Join Type Selection: Mappings based on join operations inherently allow for optimizations by specifying join predicates. Specifically, patterns that include equi or theta join predicates can significantly narrow the result space, reducing the computational overhead compared to less selective join types. However, the applicability of these optimizations depends on the operator support of the underlying ASP system. As most systems, NebulaStream natively supports equi joins, as they align with the key-based data partitioning strategies of ASP systems, in contrast to cross and theta joins [1]. To this end, we extend the operator set of NebulaStream with support for cross and interval joins. Whereas all other join types in NebulaStream create a sequence of windows based on time measures, the interval join creates windows based on the occurrence of tuples from the left join side. This window creation is a natural fit to CEP semantics and has the key benefit of duplicate-free match detection. In our implementation, the system automatically identifies patterns with an equi join predicate, whereas all other patterns are evaluated using a nested loop join. Furthermore, users can define different window types, e.g., sliding window or interval, in their patterns, observing the impact of different windowing strategies.

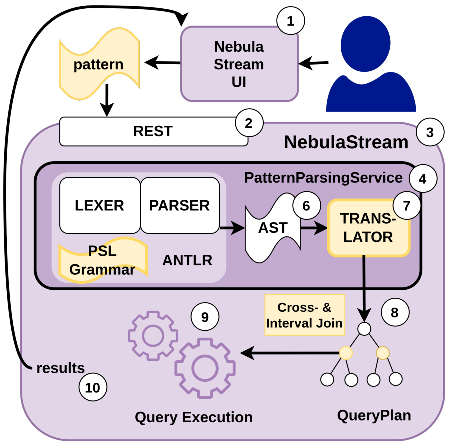
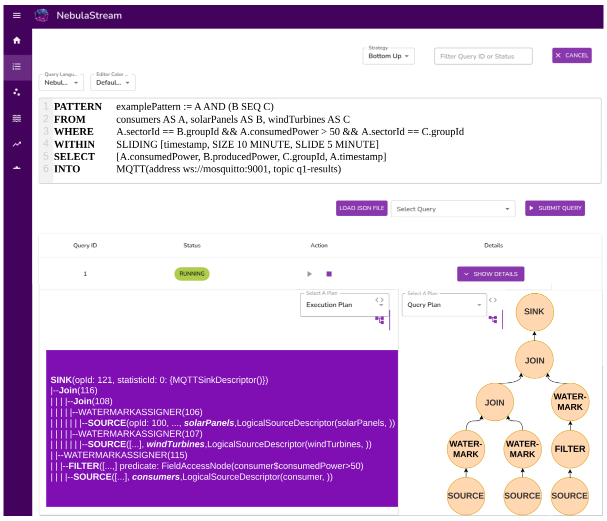


Figure 1: Workflow of our PSL in NebulaStream.

Workflow. Figure 1 illustrates the workflow for submitting and parsing a pattern in NebulaStream with our extensions highlighted in yellow. The user specifies a pattern using the NebulaStream UI (1). This pattern is submitted as a string to the NebulaStream coordinator via its REST API (2). Upon receiving the pattern string (3), the NebulaStream coordinator identifies it as a pattern submission and forwards it to the `PatternParsingService` (4), the essential extension to NebulaStream to incorporate our PSL. As NebulaStream supports both stream processing paradigms, users can also submit ASP queries using the same workflow, but these queries are directed to the `QueryParsingService`. To parse our pattern, we use the tool ANTLR [10], which provides a lexer and parser based on a defined grammar. The `PatternParsingService` incorporates the ANTLR grammar for our PSL and the auto-generated ANTLR lexer and parser code (5). The lexer processes the pattern string by tokenizing it into a stream of tokens, while the parser validates its syntactic structure and constructs the pattern's Abstract Syntax Tree (AST) (6). The AST serves as an intermediate structured representation of the pattern, encapsulating its syntax and structure while ensuring adherence to PSL language rules. It is passed to the translator (7), a module responsible for converting the syntactic representation into a logical query plan. The translator leverages ANTLR-generated listener classes, which implement enter and exit methods for each grammar rule, to extract relevant information from the AST [10]. Additionally, the translator incorporates our operator mapping. As a result, the logical query plan consists entirely of APS operators, representing a semantically equivalent query derived from the defined pattern. The resulting logical query plan (8) is then optimized and transformed into physical tasks for execution (9). Finally, the execution engine processes these tasks and returns detected pattern matches (10) to the NebulaStream UI, where they are presented to the user.

By unifying the translation of the PSL with our operator mapping, we automate the process of converting high-level pattern



```
PATTERN    examplePattern := A AND (B SEQ C)
FROM       consumers AS A, solarPanels AS B, windTurbines AS C
WHERE      A.sectorId == B.groupId && A.consumedPower > 50 && A.sectorId == C.groupId
WITHIN     SLIDING [timestamp, SIZE 10 MINUTE, SLIDE 5 MINUTE]
SELECT     [A.consumedPower, B.producedPower, C.groupId, A.timestamp]
INTO       MQTT(address ws://mosquitto:9001, topic q1-results)
```

Query ID	Status	Action	Details
1	RUNNING	▶ ■	SHOW DETAILS

```
SINK(opId: 121, statisticId: 0: {MQTTSinkDescriptor()})
|--Join(116)
| | | |--Join(108)
| | | | |--WATERMARKASSIGNER(106)
| | | | | | |--SOURCE(opId: 100, ..., solarPanels,LogicalSourceDescriptor(solarPanels, ))
| | | | |--WATERMARKASSIGNER(107)
| | | | | | |--SOURCE([...], windTurbines,LogicalSourceDescriptor(windTurbines, ))
| |--WATERMARKASSIGNER(115)
| | |--FILTER([...,] predicate: FieldAccessNode(consumer$consumedPower>50)
| | | |--SOURCE([...], consumers,LogicalSourceDescriptor(consumer, ))
```

Figure 2: Submitting an example pattern of our demonstration to the Query Catalog Page with visualization of the resulting query (right) and execution plan (left).

specifications to optimized low-level operations. Furthermore, it abstracts away underlying complexities of implementation details and our mapping rules for domain experts.

4 DEMONSTRATION

In this section, we first introduce the smart grid use case from the smart city simulation IoTropolis [7] as the application scenario for our demonstration, alongside the NebulaStream UI. Finally, we outline our demonstration plan that utilizes both frameworks.

Smart Grid Use Case of IoTropolis. This demonstration leverages the smart grid use case within IoTropolis [7]. In this use case, energy is produced by distributed wind turbines and solar panels, while households, offices, factories, and streetlights act as energy consumers. Data regarding the produced and consumed energy, along with the respective locations within the city, are transmitted to NebulaStream. IoTropolis offers interactive features that allow attendees to actively influence the environment, e.g., participants can adjust wind levels to control the energy production in the city. By leveraging IoTropolis, we create an engaging and dynamic setting with six distinct event types (streams) available for pattern specification. The interactive capabilities of IoTropolis enable attendees to manipulate the data to trigger patterns and observe how the system responds, providing a hands-on demonstration of our approach.

NebulaStream UI. The UI provides end users and attendees with an intuitive interface to interact with NebulaStream. It offers several useful features, e.g., the *Topology* page or the *Source Catalog*. The key functionality for this demonstration resides in the *Query Catalog*, which enables users to submit new queries and patterns. All registered queries are displayed in the query catalog directly below the submission component for further inspection. In Figure 2, we illustrate the process of submitting one of our example patterns. Users can write patterns using our PSL in the free-text field at the top of the page. The pattern is added to the query catalog upon pressing the *Submit Query* button. This catalog provides an organized overview of all submitted queries, including their current status and additional details, such as the logical query and the corresponding execution plan. We depict the query and execution plan corresponding to our example pattern at the bottom of Figure 2. Furthermore, the *Result Visualization* page plays a crucial role in our demonstration by allowing attendees to visualize results as charts, such as a line chart for detection latency. This enables attendees to observe and compare the impact of different patterns (or queries) and data characteristics.

Setup. The demonstration runs on a local machine, integrating NebulaStream for data processing and IoTropolis for smart grid simulation. The setup can be fully reproduced using the NebulaStream Tutorial[4].

Guided Demonstration. Attendees are invited to participate in our guided demonstration. We will start by briefly motivating our work. Then, we will introduce the demo setup, i.e., the smart grid use case in IoTropolis and key UI features.

- Attendees collaborate with us to discuss and submit three prepared patterns via the NebulaStream UI, including two semantically equivalent patterns with different join orders.
- Together, we analyze the resulting query plans, highlighting the effects of our mapping and its optimizations.
- Using the interactive features of IoTropolis, we alter the smart grid behavior to trigger the specified patterns, simulating real-world dynamics.
- Finally, we use the result visualization features of NebulaStream UI to visualize the detected matches and highlight the differences between CEP and ASP.

Interactive Engagement. Beyond the core demonstration, attendees are encouraged to take on the role of end users. By creating and deploying patterns and queries, attendees can experience firsthand the advantages of unifying CEP and ASP paradigms in a single platform. Our team will be available to answer questions, highlight key insights of our work, and discuss future directions.

ACKNOWLEDGMENTS

This work is funded by the German Federal Ministry of Education and Research under the grant BIFOLD25B. We thank the NebulaStream team and our outstanding students, especially Nils Schubert, Ricardo Martinez, Hamza Mnari, and Julian Springer, for their contributions.

REFERENCES

[1] Shijiu Cao, Haihong W, et al. 2018. Optimization of Data Distribution Strategy in Theta-join Process based on Spark. In *ICACS 2018, China*. ACM.
[2] Sharma Chakravarthy, V. Krishnaprasad, et al. 1994. Composite Events for Active Databases: Semantics, Contexts and Detection. In *VLDB, Chile*.
[3] E. F. Codd. 1972. Relational Completeness of Data Base Sublanguages. *Research Report / RJ / IBM / San Jose, California* (1972).
[4] Nikos Giatrakos, Elias Alevizos, et al. 2020. Complex event recognition in the Big Data era: a survey. *VLDB J.* (2020).
[5] Daniel Gyllstrom, Eugene Wu, et al. 2006. SASE: Complex Event Processing over Streams. *CoRR* abs/cs/0612128 (2006).
[6] Ilya Kolchinsky and Assaf Schuster. 2018. Join Query Optimization Techniques for Complex Event Processing Applications. *Proc. VLDB Endow.* 11, 11 (2018), 1332–1345. https://doi.org/10.14778/3236187.3236189
[7] Aljoscha P. Lepping et al. 2023. Showcasing Data Management Challenges for Future IoT Applications with NebulaStream. *Proc. VLDB Endow.* 16, 12 (2023).
[8] David Luckham. 2019. What's the Difference Between ESP and CEP? https://complexevents.com/2020/06/15/whats-the-difference-between-esp-and-cep-2/ Accessed Dec 2024.
[9] Yuan Mei and Samuel Madden. 2009. ZStream: a cost-based query processor for adaptively detecting composite events. In *SIGMOD 2009, USA*.
[10] Terence Parr. 2022. ANTLR4. https://www.antlr.org/ Accessed Nov 2024.
[11] Steffen Zeuch, Ankit Chaudhary, et al. 2019. The NebulaStream Platform: Data and Application Management for the Internet of Things. *CoRR* (2019).
[12] Ariane Ziehn et al. 2024. Bridging the Gap: Complex Event Processing on Stream Processing Systems. In *EDBT, Italy*.

[4]https://github.com/nebulastream/nebulastream-tutorial/