Bridging the Gap: Complex Event Processing on Stream Processing Systems

Ariane Ziehn[1], Philipp M. Grulich[1], Steffen Zeuch[1], Volker Markl[1,2]
TU Berlin[1], DFKI GmbH[2], Germany
firstname.lastname@(tu-berlin.de[1], dfki.de[2])

ABSTRACT

Analytical Stream Processing (ASP) and Complex Event Processing (CEP) extract knowledge from unbounded data streams. ASP solutions are optimized for scalable cloud environments to handle huge volumes of data in motion. In contrast, CEP solutions are designed for single-machine deployments, limiting their usage for large data volumes and distributed processing. A few hybrid solutions seek to address the lack of support for large-scale CEP by enabling its support in ASP systems and exploiting their data collection and distribution capabilities. However, these hybrid solutions assign the entire pattern workload to a single unary operator, which becomes the bottleneck of the entire execution pipeline. In addition, this composed operator prevents the application from utilizing the highly efficient stream processing optimization capabilities currently available in ASP systems. In this paper, we propose a novel operator mapping that overcomes the drawbacks of current hybrid solutions. In particular, we bridge the gap between CEP and ASP by mapping CEP to ASP operators, enabling the decomposition of the pattern workload into multiple operators. As a result, our mapping enables CEP workloads to piggyback on the scalability and efficiency of cloud-based ASP systems. Our results demonstrate that our proposed mapping outperforms the single-operator solution for semantically equivalent ASP queries by a factor of up to 150x and enables workloads that current CEP solutions do not sustain. As a result, our mapping truly unlocks the benefits of both paradigms in one system by enabling a broad range of CEP functionalities in general-purpose ASP systems.

1 INTRODUCTION

CEP is a stream processing paradigm that has emerged to detect interesting behavior in data streams based on user-defined patterns [32, 56, 62]. With the rise of the Internet of Things, CEP functionality is required for various emerging application scenarios such as traffic congestion monitoring, smart street lighting, or vehicle pollution control [11, 41, 78]. Utilizing CEP functionality for data-intensive and time-sensitive applications requires scalable CEP systems that leverage distributed computation environments [44, 66]. Nowadays, cloud environments have become the preferred computational platform for state-of-the-art data processing systems, including analytical stream processing systems (ASPSs), e.g., Flink [18], or Spark [75]. ASPSs efficiently gather data from external sources centrally in the cloud, enabling access to potentially unlimited resources for data processing. In order to maximize the utilization of cloud resources, these ASPSs provide advanced features such as parallel processing and flexible resource allocation to deal with large data volumes and high ingestion rates [26, 42, 44]. However, no general-purpose CEP system exists yet that can leverage the capabilities of cloud environments to the same extent as ASPSs [26, 42]. The main limitation of traditional CEP systems is that they rely on stateful models such as automata and are primarily designed for single-node execution or centralized architectures with serial processing models [44].

To provide CEP in cloud environments, two approaches exist that integrate CEP functionality into cloud-optimized ASPSs [44]. The first approach, Stratio Decision [2], runs instances of CEP systems on worker nodes of an ASPS-managed cluster. Thus, this approach leverages the data-gathering capabilities of the ASPS and enables workload distribution based on patterns. However, this approach integrates traditional CEP systems as a black box for the ASPS. Consequently, it includes the limitations of traditional CEP systems [44] and prevents leveraging ASPS optimizations beyond data-gathering. The second approach used by *Esper on Storm* [1], KafkaStreamCEP [52], and FlinkCEP [3], incorporates CEP functionality as an additional unary operator that can be combined with other operators in an execution pipeline of the ASPS. We refer to this approach as a hybrid stream processing system (HSPS) that unifies the functionality of ASP and CEP in a single system.

The benefit of an HSPS is three-fold: First, it mitigates the scalability limitations of state-of-the-art CEP solutions [26, 42, 44] by enabling ASPS optimization such as parallel execution and load balancing of the CEP operator. Second, from the user perspective, an HSPS allows running workloads of both paradigms in a single system by providing more flexible functionality and ease of use compared to the usage of multiple systems [44]. Third, from a system perspective, HSPSs can leverage the synergies of both paradigms instead of maintaining and optimizing similar execution environments separately. On the downside, the integration of CEP functionality as a single operator has limitations. First, an unary CEP operator can only be applied to a single stream, while CEP typically composes events from potentially various heterogeneous streams. Thus, this unary operator forces the union of all involved streams before pattern detection [51, 59]. Second, the single operator approach composes an expensive stateful computation task with a complexity equivalent to a multi-way join [55] into one operator. To this end, this expensive operator limits the maximal sustainable throughput of the entire ASP execution pipeline by preventing pipeline parallelization and operator reordering [51, 54].

In this paper, we introduce a general operator mapping that bridges the gap between both paradigms and enables the translation of CEP patterns into ASP queries. Our mapping provides the benefits of HSPS by eliminating the drawbacks of the state-of-the-art single CEP operator approach. To reach this goal, we first exploit the synergies between ASP and CEP and leverage similar functionalities of both stream processing paradigms, such as event time processing, continuous queries, and windowing [36, 60, 69]. Second, we formally define the set of common CEP operators described in Simple Event Algebra (SEA) [44] to investigate the semantic similarities and differences between the operators in both paradigms. Third, we map each SEA operator to its semantically equivalent ASP representations and show optimization potentials. With this work, we enable a wide range of CEP workloads in HSPSs by leveraging existing optimizations of ASPSs. In particular, our mapping decomposes the pattern workload into its different operators and thus leverages pipeline parallelism. For this reason, it outperforms the current single CEP operator approach, whose performance is severely affected by multiple sources and increased selectivities. Furthermore, we show that the complex workload composed in the stateful CEP operator leads to extensive memory consumption, preventing the execution pipeline from coping with high ingestion rates. Furthermore, the extensive memory consumption makes the application error-prone. In particular, in the presence of high ingestion rates, the stateful model incorporated in this operator builds up outdated intermediate results, which lead to garbage collection stalls and even system failure. In contrast, our mapping avoids performance degradation and execution failures for these challenging workloads. As a result, our mapping enables pattern detection in dynamic stream processing environments with multiple sources and high-frequent streams. In summary, our contributions are as follows:

- We contrast both stream processing paradigms to identify similarities and conceptual differences (see Section 2).
- We formally define common CEP operators based on SEA to provide clear semantics for CEP patterns (see Section 3).
- We map SEA operators using our formal definitions into their ASP counterparts to enable CEP pattern detection on a general-purpose ASPS (see Section 4).
- We evaluate the efficiency of our mapping using Apache Flink as a representative HSPS under a variety of pattern parameters and workloads (see Section 5).

Finally, we conclude this paper with an overview of related work in Section 6 and a summary in Section 7.

2 CONTRASTING ASP & CEP

In the following, we compare and highlight the conceptual differences between ASP and CEP based on the four component models that form general stream processing systems (SPSs).

High-Level Overview of an SPS: We first provide a high-level overview of an SPS from a unified perspective in Figure 1. An SPS receives streams of data generated by data producers as input. Internally, the SPS transforms each received data item from the input stream into a representative item of its 1 data model. Users submit continuous requests to the SPS using its 2 language model. These requests operate on the input stream, process individual data items, and produce an output stream as a result. The SPS uses its 3 processing model and 4 time model to apply the requested operations to the input streams. The derived results of these operations are sent back to the user as an output stream. In the remainder, we introduce the four models in detail and contrast them for ASP and CEP.

1 **Data Model:** A stream S is a continuous and unbounded list of data items generated by distributed data producers [14, 36, 44, 47]. SPSs commonly consider a data model, i.e., the representation of a data item, of tuples [14, 17, 36, 44, 73]. A tuple t is a list of attributes t(a_1,...,a_n), and all tuples $t_i \in S$ share the same attribute list, i.e., a common schema $S(a_1,...,a_n)$. For a tuple t, we write $t.a_i$ for the attribute $a_i \in S(a_1,...,a_n)$. *ASP* refers to its data model as a stream of tuples. In contrast, *CEP* considers a stream of events. An event e is a tuple containing a time attribute $e.ts$ that specifies when the event was created by its producer [36, 44, 54]. In particular, we assume that each producer creates a sequence of events with discrete and continuously increasing timestamps. Thus, the data models of both paradigms are equivalent, i.e., one can map

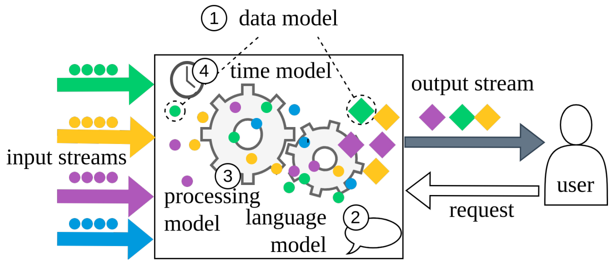


Figure 1: High-level overview of a SPS.

an event of the CEP model to an ASP tuple with an additional timestamp attribute. Moreover, CEP distinguishes events based on their content in so-called event types [44, 54]. Let $\epsilon = \{T_1,...,T_n\}$ be the universe of event types and each event e an instantiation of an event type $T_i \in \epsilon$ [15]. The event type can be either provided as an attribute or needs to be inferable [55]. We write $e \in \epsilon$ for the event type of e. Furthermore, the events of the output stream are matches of the pattern, i.e., compositions of the events e_i that participated in the pattern detection process [12, 37, 44, 74, 76]. In particular, each match M is a tuple $ce(e_1,...e_n,ts_b,ts_e)$, where for each pair (e_i,e_j) it is true that $|e_i.ts - e_j.ts| < W$ [44]. Furthermore, $ce.ts_b$ and $ce.ts_e$ are the timestamps of the first and last occurred event in M.

2 **Language Model:** Users specify continuous requests using the provided language(s) of the SPS to extract knowledge from the data stream. Requests are called queries in *ASP* and patterns in *CEP*. We split the language model discussion into two aspects, i.e., programming language and operators.

Programming Language. In order to provide high flexibility for query specification (transformations), *ASP*Ss provide low-level programming APIs that enable the definition of arbitrary data transformations, e.g., map() or UDFs [18, 36]. Furthermore, many ASPSs also provide a declarative language based on SQL [18, 75]. In contrast, *CEP* systems commonly provide declarative pattern specification languages (PSLs) to simplify specification for domain experts (non-programmers). Many PSLs use a SQL-like syntax, e.g., SASE+ [48] or CCL [78]. We use the SASE+ language with the general structure presented in Listing 1 and an example pattern with the sequence operator (SEQ) in Listing 2. One non-declarative exception is the language model of FlinkCEP [3], which is a functional programming API.

Listing 1: General structure.

```
PATTERN <pattern structure>
[WHERE <predicates>]
[WITHIN <window>]
[RETURN <output definition>]
```

Listing 2: Example pattern.

```
PATTERN SEQ (T_1 e_1, T_2 e_2, T_3 e_3)
WHERE e_1.value ≤ e_2.value
      ∧ e_3.value ≤ 10
WITHIN 4 MINUTES
```

Operators. Both language models mainly differ in their supported operators. *ASP* focuses on data transformation and enhances SQL-based operators such as joins and filters with flexible UDFs. In contrast, *CEP* relates data items by time and cause using temporal and logical operators. Since the CEP paradigm originated from several research lines, e.g., active databases [68], publish-subscribe systems [39], or data stream management systems [22], it has no universally agreed language model [36, 44]. For instance, logic-based CEP systems use event [57] and interval calculus [19], which offer various temporal operators, such as *within* and *before*, but do not support iterations. In contrast, iteration and sequence are the core operators of ordered-based CEP systems, which do not provide the variety of temporal operators of interval calculus. SEA is the result of current research efforts [17, 44] that is consistent with related work [16, 36, 64] and provides a trade-off between complexity and expressiveness. For this reason, we opted to choose

SEA as the baseline for our mapping. In particular, SEA contains the following eight operators: selection, projection, window, sequence, conjunction, disjunction, iteration, and negation. From a unified perspective, the following two operators are semantically equivalent in both stream processing paradigms:

(1) *Selection* $\sigma_\theta(t)$ *(ASP: filter)* returns an input tuple t if the user-defined set of predicates θ is fulfilled or discards t from further processing [18, 75]. (2) *Projection* $\Pi_m(t(a_1,...,a_N))$ *(ASP: map)* transforms the schema and attribute values of a_i of the input tuple t according to a set of mapping expressions m and returns the transformed tuple [18, 75]. The remaining SEA operators are disjoint from ASP operators and require an in-depth analysis of their semantics due to the heterogeneous language models of CEP [44]. We introduce and formally define these operators in Section 3 to further investigate similarities between ASP and CEP operators.

(3) **Processing Model:** The processing model of an SPS transforms the user-provided requests into an internal, logical representation, which is optimized and translated to physical tasks for query execution in ASPS and pattern detection in CEP systems. In *ASP*Ss, each query consists of three components: sources, operators, and sinks. A source forwards the tuples of an input stream to an operator. Each operator consumes input tuples from one or more sources and produces output tuples, which can be forwarded to another operator. Operators can be stateless, i.e., they process each tuple independently, or stateful, i.e., the processing depends on multiple tuples and is blocked until all required tuples arrive. Finally, a sink consumes the produced output tuples. ASPSs use directed graphs as a processing model that connects all operators between sources and sinks [13, 26, 36]. The operator order can be optimized to improve processing performance, e.g., the order of multiple joins or filter push-downs. Furthermore, one operator can be split into independent sub-operations using key assignments, which are processed in parallel and on different nodes. Shuffling steps between two operators might be required to re-partition the output tuples of several sub-operations to the next operator.

CEP systems use a variety of so-called pattern detection mechanisms, e.g., state machines for order-based mechanisms [39, 76] or tree structures for tree-based mechanisms [64]. CEP relates events by time and cause, i.e., within a certain time interval, one event causes the occurrence of another event. Therefore, temporal operators, such as the sequence operator that accepts events occurring in temporal order, are essential for the CEP paradigm. These operators resemble regular expressions and lead to the prominent usage of order-based evaluation mechanisms [36, 42, 44, 45, 55]. We briefly introduce this mechanism using a common representative, i.e., a nondeterministic finite automaton (NFA). An NFA consists of a finite set of states Q with one initial state q_0 and one or more final states F_n. Each state q_n ($n > 0$) represents a partial match of the pattern, particularly for order-based mechanisms, it denotes a prefix of the pattern. The partial matches of each state need to be stored and are combined with new arriving events accepted by the state (stateful processing) [37, 42, 54, 55]. States are connected with transitions given the specified order of event types in the pattern. Every transition accepts a new event e (and transitions to the following state q_{n+1}) if it is of the accepting event type and fulfills the corresponding user-defined predicates. For instance, for the pattern in Listing 2, the detection is triggered by the arrival of an event $e_1 \in T_1$, which is a partial match of q_1. The final state F is reached with the successive arrivals of all events, and a full pattern match is detected.

(4) **Time Model:** The notion of time is essential for SPSs to relate tuples in a timely order or to create finite substreams to process stateful operators [13, 36]. *CEP* systems consider the time model of event time [36]. This model processes events based on their time attribute. In contrast, *ASP* additionally provides the time model of processing time, which processes tuples based on the system clock. While the event time of a tuple never changes, the processing time is updated after every operator.

We conclude from our unified overview of SPS models that both systems share basic functionalities. In particular, CEP restricts its data and time model to a subset of the ASP counterparts, i.e., time-stamped tuples (events) and event time processing. The main difference between both paradigms is the systems' internals, i.e., the supported operations on the streams, which we investigate in detail in the remainder of this paper.

3 CEP OPERATORS

In this section, we introduce the CEP operators, which differ from the traditional ASP operators (described in Section 2), and their formal semantics. First, we describe the required modifications to well-defined operators using the literature from related research lines in Section 3.1. Second, we present the formal semantics of the remaining SEA operators, i.e., conjunction, sequence, disjunction, iteration, and negation, in Section 3.2.

3.1 Formal Semantics and Modifications

As with the majority of PSLs [64, 76], SEA only provides informal semantics, i.e., verbal descriptions, of its operators. Our mapping requires formally defined semantics to map SEA operators into their ASP counterparts. Investigating the literature [21, 27–29, 57, 68] for formal semantics disclosed that no well-defined, commonly agreed-upon definition of these operators exists across systems. In order to achieve principled and well-defined semantics for our mapping, we give our own formal definition based on the literature. To this end, we modify well-defined operators of traditional event algebras in active databases using the representative Snoop [29] to enable their usage for SPS. In particular, we apply the following modifications:

3.1.1 Modifications. Traditional event algebras define their operators as Boolean functions to detect patterns in a point-in-time manner. Let us assume a simple pattern applied to the universe of event types ϵ that requires the occurrence of an event $e \in T$. To detect this pattern, the Boolean function T(ts) returns $true$ if an event $e \in T$ occurs at the point in time ts, else $false$ [29]:

$$T(ts) = \begin{cases} True, \text{ iff } e \in T \land \text{e.ts = ts} \\ False, \text{else} \end{cases} \quad (1)$$

In contrast to batch-optimized database systems, SPSs use windowing to cope with unbounded streams. In particular, windowing introduces a bounded lifetime of an event, i.e., how long an event is valid before it can be discarded from further processing. However, there is a major difference in dealing with window constraints between the processing model of CEP and ASP systems: Order-based CEP systems, as well as most CEP systems, use an implicit windowing, i.e., the system contains no actual window logic, and the window constraint is transformed into predicates [44]. In contrast, ASPSs (and some CEP systems such as ZStream [64] and RTEC [21]) rely on explicit windowing that splits the input stream into finite and subsequent substreams of length W (window size) for processing [28, 43]. We focus on explicit windowing to map SEA operators to ASP operators. To this end, we replace the point-in-time detection of events in Equation 1 with a window constraint. We use ts_b and ts_e to define an arbitrary time interval $[ts_b, ts_e)$

such that $W = ts_e - ts_b$. Thus, we adjust the input of T to a set of events E = $\{e_1,...,e_n\}$, where for each event e_i, it is true that $e_i.ts \in [ts_b,ts_e)$. We modify Equation 1 as follows:

$$[T]_{ts_b}^{ts_e} = \begin{cases} True, \text{ iff } \exists e_i \in T \land e_i.ts \in [ts_b,ts_e) \\ False, \text{ else} \end{cases} \tag{2}$$

Additionally, we adapt the output of the function $[T]_{ts_b}^{ts_e}$ (Equation 2) to fulfill the closure properties of SEA. In particular, the function either returns the set of events satisfying all pattern constraints or an empty set instead of a Boolean value. To this end, we formally define our operator semantics as follows:

$$[T^*]_{ts_b}^{ts_e} = \begin{cases} \{e|e \in T \land e.ts \in [ts_b,ts_e)\} \\ \emptyset, \text{else} \end{cases} \tag{3}$$

In sum, we adapt the semantics of traditional event algebra to address the requirements for stream processing and the specification of SEA for our operator semantics. To this end, we focus on explicit windowing, which makes the window operator a core component of each pattern. Thus, we turn to essential details of window semantics.

3.1.2 Window Operator. *Definition.* The window operator is a temporal operator with a commonly time-based constraint W, such as 20 minutes, that requires all events of a match to occur within a maximal time difference of W. This definition represents a time-based sliding window for explicit windowing [44]. Explicit windowing incorporates the following two semantic components:

(1) Intra-Window Semantic. The intra-window semantic defines which events are assigned to which finite substream(s) T_k. For time-based windows, each event e with a timestamp $e.ts \in [ts_{b_k},ts_{e_k})$ is assigned to the finite substream T_k. Formally,

$$[T]_{ts_b}^{ts_e} = T_k = \{e|e \in T \land e.ts \in [ts_{b_k},ts_{e_k})\} \tag{4}$$

, where each finite substream T_k has a time interval $[ts_{b_k},ts_{e_k})$ with the window length $W = ts_{e_k} - ts_{b_k}$ [47]. Operators combined with the window specify further constraints on the events $e_i \in T_k$ to form a match, e.g., event types or temporal order.

(2) Inter-Window Semantic. The inter-window semantic of a window operator defines how subsequent windows are created and, thus, how the stream is discretized into substreams. In particular, for sliding windows, a fixed slide size s is specified by the user that declares when subsequent windows start [44, 47]. Thus, sliding windows create a sequence of subsequent, potentially overlapping substreams T_{k+l} as follows:

$$T_{k+l} = [T]_{ts_{e_{k+l}}}^{ts_{b_{k+l}}} \tag{5}$$

, where $ts_{k+l} = ts_k + s \cdot l$, respectively ($k,l \in \mathbb{N}$).

Syntax. Explicit windowing combines stateful operators with the window operator for processing. Thus, all operators in Section 3.2 have to be combined with a window operator. The window operator is specified with the keyword *WITHIN (W,s)*.

3.1.3 Correctness of Operator Semantics. Since we adapt traditional operator semantics, we need to ensure the correctness of our mapping. The essential correctness criteria are the detection of all matches contained in a stream S. In particular, by incorporating the window operator into our operator semantics, we must ensure that no match $M = ce(e_1,..,e_n)$ is lost by discretizing the stream S.

Theorem 1. *Given a pattern P and a substream S_k, our intra-operator semantics detected all matches of the pattern in S_k.*

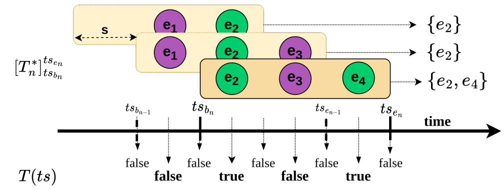


Figure 2: Contrasting the semantics of T(ts) and $[\mathrm{T}^*]_{\mathrm{ts_b}}^{\mathrm{ts_e}}$.

Proof. Let the complex event $ce(e_i,e_j)$ be a valid match M of the pattern P in S_k. Then, by definition, $e_i,e_j \in S_k$ and $e_i.ts,e_j.ts \in [ts_b,ts_e)$. Thus, ce is an output tuple of our operator. □

Theorem 2. *Given a match $M = ce(e_1,..,e_n)$ of pattern P and a stream S, there exists at least one substream S_k such that $e_i,e_j \in S_k$ and, thus, M is detected by our operator.*

Proof (sketch).[1] By definition, for every event pair $(e_i,e_j) \in ce$ it is true that $max(e_i.ts,e_j.ts) - min(e_i.ts,e_j.ts) < W$ [44]. It follows that $W-1$ is the maximal time difference between a pair in ce. A match M that contains a pair (e_i,e_j) which is $W-1$ time units apart is only detected in $S_k = [S]_{ts_b}^{ts_e}$, if $min(e_i.ts,e_j.ts) = ts_b$. Otherwise $ts_b + W - 1 + n > ts_e - 1 \rightarrow max(e_i.ts,e_j.ts) \notin [ts_b,ts_e)$. Hence, we must ensure that there exists a S_{k+l} in which e_i and e_j occur. To this end, let us consider the worst-case scenario where $e_i \in S_k$ and $min(e_i.ts,e_j.ts) = e_i.ts = ts_{e_k} - 1$. To detect (e_i,e_j), we need to ensure that $\exists S_{k+l}$ so that $e_i.ts = ts_{e_k} - 1 \land e_i.ts = ts_{b_{k+l}}$. It follows that if $e_i.ts = ts_{b_{k+l}}$, $e_j.ts = ts_{b_{k+l}} + W - 1 = ts_{e_{k+l}}$. Thus, $e_j \in S_{k+l}$ and M is detected. This implies a slide size of one for slide-by-tuple sliding windows or a slide size smaller or equal to the frequency of the stream with the highest arrival rate to guarantee that $\forall e \in S\ \exists S_{k+l}(e.ts = ts_{b_{k+l}})$. □

3.1.4 Impact of Modifications. We modify the input and output of our operator semantics to ensure the closure properties of SEA and introduce explicit windowing for stream processing. We visualize the resulting difference between the traditional event algebra and our semantics in Figure 2, where events of type T and other types occur over time. The traditional Boolean function $T(ts)$ (Equation 1) evaluates for each point in time if an event matches all conditions of $T(ts)$. In contrast, $[T^*]_{ts_e}^{ts_b}$ (Equation 3) evaluates the set of events occurring within the specified time interval $[ts_b,ts_e)$ of each subsequent substream and returns a set matches. In Figure 2, each yellow rectangle depicts a substream and outgoing arrows the detected matches. The differences between both functions lead to four major impacts on how events are processed:

First, the traditional function $T(ts)$ uses implicit windowing to ensure eager pattern detection by immediate returns of a match, i.e., the condition $e.ts = ts$. In contrast, explicit windowing uses lazy pattern detection, thus buffering all occurring events till ts_e before processing [44]. For this reason, explicit windowing implies a higher detection latency than implicit windowing. However, the slide size s introduces an upper bound to this latency overhead. Second, to ensure the correctness of our semantics, we create overlapping substreams. This leads to the detection of duplicate matches. Duplicate matches are irrelevant for idempotent actions but need to be maintained otherwise, e.g., by the operator state. Third, as the content of a window is unordered, windowing restricts the selection policies supported by our approach. Selection policies express additional constraints on the temporal order of relevant and partially irrelevant events to form a match [44, 77].

[1] Formal proof available at github.com/arianeziehn/CEP2ASP.

As our final semantics describe operations on sets, they correspond to the most common selection policy *skip-till-any-match*. *Skip-till-any-match* considers any combination of relevant events for a match, regardless of whether irrelevant events occur in between [44, 76, 77]. Thus, *skip-till-any-match* is the most flexible as well as most computationally expensive policy with worst-case exponential growth [55]. Other common policies are *skip-till-next-match*, which ignores the occurrence of irrelevant events until the next relevant event occurs, and *strict-contiguity*, which requires all events participating in a match to occur directly after another (without an irrelevant event in-between). The matches derived by *skip-till-any-match* are supersets of these policies [44, 76]. To this end, *skip-till-next-match* results can be constructed from *skip-till-any-match*, while *strict-contiguity* requires ordered window content to determine valid matches. Fourth, in contrast to traditional CEP systems, the specification of a window operator is mandatory for every pattern using our semantics. However, window constraints define a time interval in which an event is valid. Without this constraint, events are valid forever, leading to an ever-growing state in stream processing scenarios. For this reason, the window constraint is considered to be a common composition of the pattern regardless of the CEP system [16, 44, 49, 67]. For the rare number of patterns without a window constraint, this limitation implies the overhead for the user identifying the lifetime of an event.

3.2 Operator Semantics

In the following, we use the introduced modifications to formally define SEA operators. To this end, we first provide a detailed example of how these modifications are applied to the conjunction. Second, we present the final semantics of the remaining operators, i.e., sequence, disjunction, iteration, and negation, based on the well-defined operators of the CEP-related research line active databases. We refer to Snoop [29] for the traditional operator semantics.

Conjunction: *Definition.* The conjunction is a binary operator that expects the occurrence of both events $e_1 \in T_1$ and $e_2 \in T_2$ together within W [44]. Traditional event algebra formally defines conjunction as follows [29]:

$$(T_1 \wedge T_2)(ts) := \exists ts_n, ts_m : T_1(ts_n) \wedge T_2(ts_m) \wedge max\{ts_n, ts_m\} = ts \quad (6)$$

In contrast, SEA requires the pattern to occur within W, i.e., the time interval $[ts_b, ts_e)$. Therefore, we modify Equation 6 as follows:

$$[(T_1 \wedge T_2)]_{ts_b}^{ts_e} := \exists ts_n, ts_m : T_1(ts_n) \wedge T_2(ts_m) \wedge ts_n, ts_m \in [ts_b, ts_e) \quad (7)$$

Equation 7 presents the combination of the conjunction and the window operator. Thus, we extract the window operator and its time constraint from Equation 7 as follows:

$$(T_1 \wedge T_2) := \exists ts_n, ts_m : T_1(ts_n) \wedge T_2(ts_m) \quad (8)$$

Finally, we define the output of a pattern as a set of matches instead of a Boolean value. Thus, our operator function returns either the composition of both occurred events within the time interval $[ts_b, ts_e)$ or an empty set $\emptyset$. The resulting equation of the conjunction operator is defined as follows:

$$(T_1 \wedge T_2)^* = \{(e_1, e_2) \mid e_1 \in T_1 \wedge e_2 \in T_2\} \quad (9)$$

Syntax. A conjunction is specified with the keyword *AND* and is associative and commutative. Nested Patterns with multiple conjunctions, such as $AND(T_1, AND(T_2, T_3))$, can be simplified to $AND(T_1, T_2, T_3)$.

Sequence: *Definition.* The sequence is a binary temporal operator that expects the occurrence of an event $e_1 \in T_1$ followed by an event $e_2 \in T_2$ within W and where $e_1.ts < e_2.ts$ [17, 44, 74]. Formally,

$$(T_1; T_2)^* = \{(e_1, e_2) \mid e_1 \in T_1 \wedge e_2 \in T_2 \wedge e_1.ts < e_2.ts \quad (10)$$

Syntax. A sequence is specified with the keyword *SEQ* and is associative. Due to the temporal constraints, a sequence is not commutative, as the order of event occurrences is relevant. However, it can reach this property using additional time constraints to guarantee the order of events [71, 78]. Patterns with nested sequences, such as $SEQ(T_1, SEQ(T_2, T_3))$, can be simplified to $SEQ(T_1, T_2, T_3)$.

Disjunction: *Definition.* The disjunction is a binary operator that expects either $e_1 \in T_1$ or $e_2 \in T_2$ to occur within W [17, 44]. Formally,

$$(T_1 \vee T_2)^* = \{e \mid e \in T_1 \vee e \in T_2\} \quad (11)$$

Syntax. A disjunction is specified with the keyword *OR* and is associative and commutative [71]. Patterns with nested disjunctions, such as $OR(T_1, OR(T_2, T_3))$, can be simplified to $OR(T_1, T_2, T_3)$.

Iteration: *Definition.* The iteration is a unary operator that allows for m event occurrences ($m > 0$) of the event type T in a sequence [17, 44]. Formally,

$$(T^m)^* = \{(e_1, ..., e_m) \mid \forall\, 1 \leq i \leq m : e_i \in T \wedge (e_1.ts < ... < e_m.ts)\} \quad (12)$$

Note that in contrast to the Kleene∗ and Kleene+ operator of standard regular expressions, the SEA iteration operator is bounded to the exact occurrence of m events [17, 44].
Syntax. An iteration is specified with the keyword $ITER^m$.

Negation: *Definition.* The negation is a unary operator that requires the absence of any event $e \in T$ in W to match the pattern [17, 29, 44]. Formally,

$$\neg(T_2)[T_1, T_3](ts) = (\exists ts_1)(\forall ts_2)(T_1(ts_1) \wedge \sim T_2(ts) \wedge T_3(ts) \wedge ((ts_1 \leq ts_2 < ts) \rightarrow \sim (T_2(ts_2) \vee T_3(ts_2)))) \quad (13)$$

Following Equation 13, the negation detects the absence of T_2 in the closed interval of consecutive occurred events of type T_1 and T_3, i.e., SEQ(T_1,T_3). Thus, it restricts the usage of negation in the center of a sequence as a ternary operator, often referred to as negated sequence [16, 64]. In contrast, SEA verbally describes the negation as a unary operator, i.e., $\neg[T]_{ts_b}^{ts_e}$. However, a unary negation violates the closure properties of SEA (see Section 3.1) by returning a Boolean value instead of a set of tuples. Thus, we discard unary negation and use the ternary operator negated sequence for our mapping. The negated sequence is in line with the operator functionality offered by common CEP systems [3, 55, 64, 76] and formally defined as follows:

$$(T_1; \neg(T_2); T_3)^* = \{(e_1, e_3) \mid e_1 \in T_1 \wedge e_3 \in T_3 \wedge (e_1.ts < e_3.ts \wedge \neg \exists e_2.ts \in (e_1.ts, e_3.ts) : e_2 \in T_2\} \quad (14)$$

Syntax. The negated sequence is specified with the keyword *NSEQ* and is neither associative nor commutative.

4 GENERAL OPERATOR MAPPING

In this section, we introduce our operator mapping that enables the transformation of patterns into queries to execute them in cloud-optimized ASPSs. To this end, we use our formal operator definitions in Section 3 to identify each operator's ASP counterparts in Sections 4.1. We refer to the formal definitions of relational algebra operators [33, 34] for ASP operators and to Negri et al. [65] for the definition of semantic equivalence of two queries. In particular, two queries are semantically equivalent if, for all input tuples, the output tuples obtained are equivalent after executing the queries and eliminating duplicates. Furthermore, we discuss the generalization of mapping for ASPS in Section 4.2. Finally, we investigate optimization opportunities in Section 4.3 and summarize our findings in Table 1.

4.1 Operator Mapping

In the following, we present the mappings for all SEA operators as defined in Section 3. We provide a detailed example with mapping directives of the conjunction and enhance mappings with a brief discussion if deeper insights or analysis are required.

Conjunction: *Mapping.* Our formal definition in Equation 9 is equivalent to the definition of the relational Cartesian product $\times$ [33], which composes two streams into one as a set of pairs. Each pair is a pattern match.

Mapping Directive. We present the overall structure of a conjunction pattern in Listing 3. The PATTERN clause contains the conjunction operator AND and the input streams (event types) T_1 and T_2. Each stream of the $PATTERN$ clause is added to the $FROM$ clause of the query in Listing 4. The $WHERE$ clauses of both requests are identical and contain the pattern predicates. The $WITHIN$ clause contains the time interval W of the pattern. In the query, W defines the $Range$ of the $WINDOW$ clause and s the slide size. Finally, the $SELECT$ clause of the query is defined with $*$. Note that the output tuple can also be modified in the pattern by adding a $RETURN$ clause, which by default returns the concatenation of all the attributes of the events participating in a match.

Listing 3: AND pattern.

PATTERN AND($T_1 e_1$,$T_2 e_2$)
WHERE <predicates>
WITHIN W

Listing 4: AND query.

SELECT *
FROM Stream T_1, Stream T_2
WHERE <predicates>
Window [Range W, s]

Sequence: *Mapping.* Our formal definition in Equation 10, is equivalent to the definition of the relation Theta Join $\bowtie_\theta$ using the order by time of both events as join predicate θ [33]. In particular, all event pairs (e_1,e_2) that fulfill the condition of consecutive timestamps match the constraints of a sequence and are returned as pattern matches. We present the translation of a sequence pattern as example in Listing 7 and 8.

Disjunction: *Mapping.* Our formal definition in Equation 11 is equivalent to the formal definition of the relational set union operator $\cup$ [33]. The union operator unifies two input streams into a new one, i.e., T_1 and T_2 are unified to $T_{1,2}$. Each event $e_{1,2}$ in $T_{1,2}$ is a match of the pattern.

Discussion. Our mapping creates semantically equivalent queries but requires union compatibility of both event types. Union compatibility is a restriction compared to the traditional Boolean function that handles events of different types regardless of their schema. However, ASPSs provide the map operator (see Section 2) that allows the transformation of schemata to archive union compatibility at the minor cost of an additional stateless computation.

Iteration: *Mapping.* Our formal definition in Equation 12 equals a nested sequence over a single event type T. 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 join predicate θ [33].

Negated Sequence: *Mapping.* Our formal definition in Equation 14 represents the combination of a sequence, i.e., $(T_1;T_3)$, and the negated existential quantifier that requires the absence of any event $e_2 \in T_2$ 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 to the $WHERE$ clause of the sequence query. We present the overall structure of a negated sequence pattern in Listing 5 and its translation into a query in Listing 6.

Discussion. Our mapping creates semantically equivalent queries but uses quantifiers, which are not commonly available in ASPSs. However, the flexibility of UDFs allows the expression of $NSEQ$ patterns. In particular, we first union T_1 and T_2. Then, we apply a UDF window function that, for each event $e_1 \in T_1$, finds (if it exists) the next occurrence of $e_2 \in T_2$ within the time interval W of the pattern. To this end, we add an additional timestamp attribute a_{ts} to each event e_1. If an event e_2 occurs after e_1 within W $a_{ts} = e_2.ts$, else $a_{ts} = e_1.ts + W$ indicating that no e_2 occurred. Afterward, we perform $SEQ(T_1,T_3)$ with the additional selection $\sigma_{a_{ts}>e_3.ts}$ to guarantee that no event $e_2 \in T_2$ occurred in the interval $(e_1.ts,e_3.ts)$.

Listing 5: NSEQ pattern.

PATTERN SEQ($T_1 e_1$,$\neg T_2 e_2$,$T_3 e_3$)
WHERE <predicates>
WITHIN W

Listing 6: NSEQ query.

SELECT *
FROM Stream T_1, Stream T_3
WHERE <predicates> $\wedge$ $T_1.ts < T_3.ts$ $\wedge$
NOT EXISTS (**SELECT** *
FROM Stream T_2
WHERE <predicates> $\wedge$
$T_1.ts < T_2.ts \wedge T_2.ts < T_3.ts$)
Window [Range W, s]

4.2 Generalization

We target the general applicability of our mapping in common ASPSs. To this end, we first discuss the support of identified CEP counterparts in ASPSs in Section 4.2.1. Second, we introduce our means to cope with nested patterns in Section 4.2.2.

4.2.1 Language Model Selection. Similar to CEP, no universally agreed language models exist for ASP. Thus, our mapping requires investigating the support of the identified target operators in common ASPSs [23, 44]. To this end, we review the Stream APIs of a selection of ASPSs, i.e., Beam [4], Flink [5], Spark [7], Storm [8], and Kafka Streams [6]. We include Beam as a representative abstract query model adapted by many state-of-the-art ASPSs beyond our selection, e.g., Google Cloud Dataflow or Samza, to underline the general applicability of our mapping. Our ASPS review yields the support of all necessary counterparts for our mapping except the Cartesian product and Theta Join. To overcome the lack of the Cartesian product, a precedent map operation that assigns a uniform key to each event of the involved types, T_n can be applied before joining. Similarly, our mapping can bypass the lack of support for the Theta Join by creating the Cartesian product and filtering the results by the Theta Join predicate θ to guarantee a timely order of events. To this end, our mapping allows basic CEP functionality on common ASPSs, where the only implementation overhead is the UDF of the $NSEQ$. Thus, it enables CEP workloads on ASPSs that currently do not provide CEP support, e.g., Spark or Storm.

4.2.2 Nested Patterns. Another insight from our ASPSs review is that except Beam, no ASPS allows to specify multi-way Window Joins, i.e., the composition of more than two streams per Window Join. In particular, let us consider the SEQ example in Listing 7, which can be translated into the multi-way Window Join in Listing 8 using our mapping.

Listing 7: SEQ pattern.

PATTERN SEQ($T_1 e_1$,$T_2 e_2$,$T_3 e_3$)
WHERE <predicates>
WITHIN W

Listing 8: SEQ query.

SELECT *
FROM Stream T_1, Stream T_2, Stream T_3
WHERE $T_1.ts < T_2.ts \wedge T_2.ts < T_3.ts$
$\wedge$ <predicates>
Window [Range W, s]

Table 1: Operator Mapping Overview.

Mappings	Optimization Opportunities				
		Pattern and Data Characteristics	Func.	Perf.	Eval-uation
Conjunction ($T_1 \wedge T_2$) - AND					
$T_1 \times T_2$	O1	-	✓	✓	-
$T_1 \bowtie_c T_2$	O3	join predicate c		✓	-
Sequence ($T_1;T_2$) - SEQ					
$T_1 \bowtie_\theta T_2$	O1	-	✓	✓	§ 5.2.1- 5.2.5
$T_1 \bowtie_c T_2$	O3	join predicate c		✓	§ 5.2.3- 5.2.5
Disjunction ($T_1 \vee T_2$) - OR					
$T_1 \cup T_2$		union compatibility			-
Iteration (T^m) - $ITER^m$					
$T^1 \bowtie_\theta \ldots \bowtie_\theta T^m$	O1	-	✓	✓	§ 5.2.1- 5.2.5
$\gamma_{count(*)}(T)$	O2	unbounded m	✓	✓	§ 5.2.1- 5.2.5
$T^1 \bowtie_c \ldots \bowtie_c T^m$	O3	join predicate c		✓	§ 5.2.3- 5.2.5
Negated Sequence $\neg T_3[T_1;T_2]$ - $NSEQ$					
UDF($T_1 \cup T_2$) $\bowtie_\theta T_3$		O1, O3 as SEQ			§ 5.2.1

As Listing 8 is only applicable in Beam, two consequent Window Joins are necessary for other ASPSs and require an explicit redefinition of the event time attribute after each Window Join. In general, it is the minimum timestamp of the output pair (e_1,e_2) for a partial match of a nested pattern and the maximum timestamp for a complete match. For our example, that is the timestamp attribute of T_1 after $T_1 \bowtie T_2$. Furthermore, our mapping allows us to leverage the commutative and associative properties of CEP operators and reorder joins. In addition, we can combine different window types within one pattern to further optimize performance.

4.3 Optimization Opportunities

In the following, we investigate optimization opportunities in terms of functionality (Func.) and performance (Perf.) of our mappings, which are summarized in Table 1 with references to the respective evaluation sections.

4.3.1 Alternative Windowing with Interval Joins (O1). As discussed in Section 3.1, windowing and its parameters are crucial for the correctness of our mapping. We follow Giatrakos et al. [44] and use time-based sliding windows as they cover general CEP use cases [44] and are commonly available in ASPSs [4, 36]. Sliding windows create consecutive overlapping windows, which guarantee the detection of all complex events if sliding by tuple is applied. However, our mapping suggests using small slide sizes, leading to many concurrent windows with a negative performance impact. Additionally, due to the overlap of consecutive windows, our mapping produces duplicates.

Functionality: An alternative windowing solution that prevents setting stream-depend parameters and duplicates is the Interval Joins (available in Flink [5]). This join composes two events $e_1 \in T_1$ and $e_2 \in T_2$ given a key condition and a window condition $e_2.ts \in (e_1.ts + lowerBound, e_1.ts + upperBound)$, with bounds defined as time measurement [5]. The bounds only depend on the window size W, thus circumventing the setting of a slide size. In particular, for the conjunction, the bounds are defined as follows $(e_1.ts-W, e_1.ts+W)$. All other operators use $(e_1.ts+0, e_1.ts+W)$. Thus, the Interval Join creates content-based windows defined by events of T_1. To this end, the Interval Join detects all matches and prevents the creation of duplicates as all relevant $e_2 \in T_2$ are assigned to the unique window of $e_1 \in T_1$.

Performance: Utilizing Interval Joins yields performance benefits over Sliding Window Joins if the frequency of T_1 is significantly lower than the frequency of T_2. This improvement stems from the content-based creation of windows based on T_1 events, resulting in fewer windows and reduced computational workload. In contrast, sliding windows are created apriori and independent of actual tuple occurrences. Thus, its performance is independent of the frequency differences between both streams. Both Window Joins perform alike for similar frequencies, while Sliding Window Joins outperform Interval Joins when the frequency of T_1 is significantly higher than the frequency of T_2.

4.3.2 Leverage Aggregations for Iterations (O2). The mapping of the iteration is in line with the SEA iteration operator. However, many ordered-based CEP systems support unbounded iterations where the number of contributing events $\geq m$ instead of $= m$, resembling the Kleene$*$ and Kleene+ operator [36, 44]. While the mapping towards Theta Joins does not support any Kleene operation, we can utilize ASP aggregations to overcome this limitation [77].

Functionality: In particular, we first apply a window aggregation that returns the count n of relevant events of T in the window W. Afterward, we compare n with the user-defined m, i.e., if $n \geq m$, the pattern is fulfilled under the selection policy *skip-till-any-match*. However, we denote O2 as approximate because aggregations return one tuple of the same schema as the input stream per window instead of multiple tuples with the composition of events as the iteration operator. On the other hand, composing all events of an unbounded iteration may lead to extensive result tuples with potentially duplicate or irrelevant information. Furthermore, retrieving any accumulated information from $ITER^m$ results is barely supported in traditional CEP systems, which makes it rather cumbersome to, for instance, derive the average of an attribute $a_j \in T(a_1,...,a_n)$ for all events $e_i \in ce(e_1,...,e_m)$ [36, 44]. O2 enables further analysis by the usage of additional aggregation functions, e.g., mean or max, supported in some CEP systems [48]. Note that some ASPSs allow users to implement UDF aggregation functions, which can return multiple output tuples per window and sort the window content to support conditions between the contributing events, such as $e_i.a_n < e_{i+1}.a_n$, and other selection policies. Finally, ASP window aggregations do not trigger a window that has no event assigned. Thus, O2 cannot support Kleene$*$ operations. As a result, O2 supports a variation of the Kleene+ operation under *skip-till-any-match*, which can be extended to the full functionality of Kleene+ by using UDFs.

Performance: Due to the approximation of the result, aggregations reduce the computational load and thus provide better performance. Note that from the performance perspective, it is recommended to utilize natively supported operators from the ASP API instead of UDFs. Native calls can undergo detailed analysis by the system optimizer, resulting in significantly better performance outcomes compared to UDFs [70].

4.3.3 Data Partitioning using Equi Joins (O3). Following our mapping, four out of five SEA operators are mapped to a join type, i.e., Cross Joins and Theta Joins. Both join types are rarely available in ASPSs as they incorporate challenges of data partitioning by key and, thus, introduce massive computing and communication overhead during data processing in distributed settings [25]. Thus, O3 does not provide any optimization on functionality but a performance optimization for a subset of pattern workloads.

Performance: Common CEP use cases provide a subset of patterns that require matching attribute values, e.g., for IDs or region keys [33, 44]. Those use cases contain a join condition c with an equality operator, i.e., $e_1.a_i = e_2.a_j$, and can be translated into an Equi Join. Equi Joins enable data partitioning by key in ASPSs and, thus, enable higher degrees of parallelization compared to Cross Joins or Theta Joins. In particular, a precedent map operation that assigns a single key to all events leads to no parallelization potential. Consequently, Equi Join predicates c are always preferable as join keys when using our operator mapping. Other constraints, such as θ from the mapping of the sequence, are executed after the Equi Join. Furthermore, O3 can be combined with O1 and O2.

5 EVALUATION

In this section, we evaluate the performance of our mapping in the representative HSPS Flink. To this end, we first describe our experimental setup in Section 5.1. Then, we compare the performance of our mapping against FlinkCEP in Section 5.2.

5.1 Experimental Setup

In the following, we introduce our experimental setup in Section 5.1.1, analyze the supported SEA operators of FlinkCEP in Section 5.1.2, and present data and workloads in Section 5.1.3.

5.1.1 Hardware and Software. We conduct our experiments on a five-node cluster. Each node has a 16-core Intel Xeon Silver CPU (4216 2.10GHz) and 528 GB of main memory. We use one node exclusively as a master and the others as workers. For the experiments in Section 5.2.1-5.2.4, we use one worker and scale out to multiple workers in Section 5.2.5. We investigate the four introduced hybrid solutions, i.e., Stratio Decision [2], Esper on Storm [1], KafkaStreamsCEP [52], and FlinkCEP [3]. The former three solutions are outdated, partially archived research projects, with commits more than four years ago and deprecated dependencies for support versions of the respective ASPS[2] To this end, Flink is the only ASPS that provides an actively maintained CEP feature with FlinkCEP (FCEP) and allows us to compare the benefits and drawbacks of our solution within one SPS, excluding cross-system differences. For these reasons, we use Apache Flink (v1.11.6) for all our experiments. In particular, the performance of FCEP patterns serves as a baseline, which we compare to their corresponding FlinkASP (FASP) queries translated with our mapping.

Table 2: Operator Support of FCEP and FASP.

	AND	SEQ	OR	$ITER$	$NSEQ$	SP
FASP	✓	✓	✓	✓	✓	stam
FCEP	✗	✓	✗	✓	✓	stam
	✗	✓	✗	✓	✓	stnm
	✗	✓	✗	✓	✓	sc

5.1.2 Flink Implementation Details. FCEP uses an order-based evaluation mechanism [44], which is implemented as a unary operator that creates an NFA given a user-defined pattern. The unary CEP operator can only be applied to a single input stream, which requires the previous union of all input streams. Furthermore, as with all order-based CEP systems, FCEP uses no actual window implementation but logical windowing, i.e., predicates, to ensure that time constraints are met [44]. To define patterns, FCEP supports three of the five SEA operators exclusively provided by CEP, i.e., SEQ, $ITER$, and $NSEQ$. In contrast, our mapping enables the entire SEA operator set, as shown in Table 2. Furthermore, our mapping supports the most common selection policies (SP) skip-till-any-match (stam) [44], while FCEP additionally supports skip-till-next-match (stnm) and strict-contiguity (sc) (see Section 3.1.4). For this reason, FCEP has multiple options for its operators, e.g., for SEQ: .next() for sc, .followedBy() for stnm, and .followedByAny() for stam. To compare equivalent workloads, we use the following FCEP operators, all corresponding to stam: *.followedByAny()* for SEQ, .times(n).allowCombinations() for $ITER$, and *.notfollowedBy()* for $NSEQ$. We use exclusively these three FCEP operators for patterns and the provided FASP operators in Flink's DataStream API for queries. We exclude third-party systems and sockets from our evaluation as they would be identical for both approaches but may introduce performance bottlenecks. Thus, instead of using connectors as interfaces for data providers, we extract a fixed time frame of the data (see Section 5.1.3) as CSV files and employ a simple source operator for reading. Additionally, we ensure a fair comparison by using identical source and sink functions for all pattern-query pairs.

Finally, we turn toward the parallelization of FCEP. FCEP can leverage partitioning by key and otherwise runs on a single thread. Our mapping incorporates a similar bottleneck. In particular, if the pattern has no Equi Join condition between each stream pair, their join is performed in a global window. However, our mapping allows us to decompose the pattern workload into consecutive joins, i.e., multiple operators, and to adjust the join order to improve its performance. Furthermore, it prevents the previous union of all streams and simplifies the garbage collection of processed tuples.

5.1.3 Workloads. In the following, we give details about data and the representation of event types and patterns.

Data: We use two real-world sensor data sources for our evaluation. First, *QnV-Data*[3] represents traffic congestion management data that includes sensor readings from almost 2.5k road segments in Hessen (Germany). Each tuple contains the number of cars, i.e., quantity (Q), and their average speed, i.e., velocity (V), for one minute on a road segment defined by coordinates. Second, *AirQuality-Data* (AQ-Data) [72] contains data from $SDS011$ sensors that measure air quality, i.e., particulate matter with $PM10$ and $PM2.5$ values (particles of 10 and 2.5 micrometers or smaller). Additionally, $DHT22$ sensors provide temperature ($Temp$) and humidity (Hum) measurements. Both sensors collect data every three to five minutes. To represent these event types, we create a POJO class with a common schema for all data sources, i.e., (id,lat,lon,ts,$value$), and a child class for each measurement, i.e., Q, V, $Temp$, Hum, $PM10$, and $PM2.5$. Thus, also simplifying the union of sources for FCEP.

Metrics: We measure the maximum sustainable throughput in tuples per second (tpl/s) and the detection latency of a pattern. The detection latency results from subtracting the maximum event time of all events contributing to the output from the current system time when the output reaches the sink operator [53]. As we produce all the data in the cloud, we use the creation time of a tuple instead of its event time to derive the detection latency.

Pattern Parameters: We denote the number of event types contributing to a pattern in brackets. For instance, $SEQ(2)$ describes a SEQ of two streams. We use $\frac{\#matches}{\#events}$ [76] to determine the output selectivity σ_o of a pattern in %. We give the window size W in minutes, and, following our findings in Section 3.1, use a slide size of one minute for all sliding window queries.

5.2 Performance Evaluation

We now compare the performance of our operator mapping against the unary CEP operator approach of FCEP. We first compare the performance of elementary operators in Section 5.2.1, followed by the impact of pattern parameters in Section 5.2.2. For both sets

[2]We investigated KafkaStreamCEP [52] and migrated the project to a recent KafkaStream version. However, KafkaStreamCEP was only able to process small data sets (under 300 MB) with a low throughput of 5k tpl/s. For larger data sets, an internal buffer for pattern detection overloads, and the query fails. Thus, we exclude KafkaStreamsCEP from further evaluation.

[3]The data is no longer available on the public data portal $mCLOUD$ [63]. All utilized samples are provided in our GitHub repository: github.com/arianeziehn/CEP2ASP.

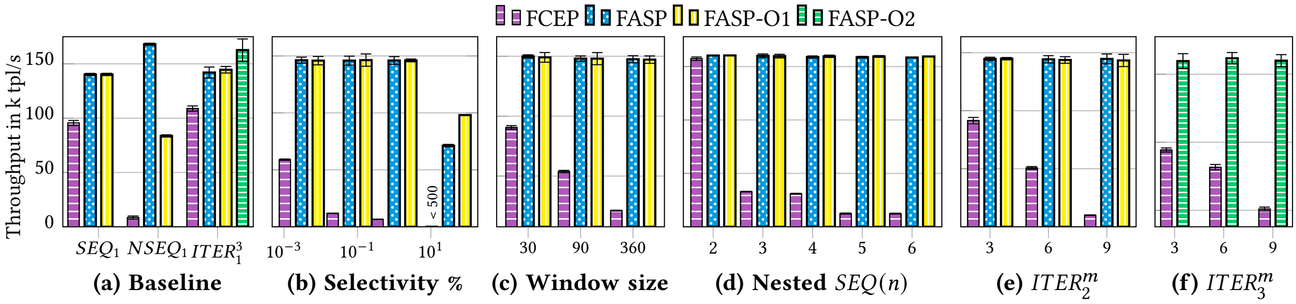


Figure 3: Elementary operator performance with the impact of different parameters.

of experiments, we use patterns that do not allow for naive key partitioning and thus skip the evaluation of O3. In Section 5.2.3, we evaluate the effect of data characteristics, followed by the study of resource utilization in Section 5.2.4. Finally, we turn towards the scalability of both approaches in Section 5.2.5.

5.2.1 *Elementary Operator Performance.*

In this experiment, we evaluate three simple patterns that each consist of one elementary operator, i.e., $SEQ_1(2)$ and $ITER_1^3(1)$ with a sample of 10 million tuples (10M tuples) from QnV-Data (0.89GB) and $NSEQ_1(3)$ with a sample of 10M tuples (0.5GB) from QnV- and AQ-Data. For all patterns, we use an output selectivity $\sigma_o = 0.00005\%$ and a window size $W = 15$.

Observations. In Figure 3a, we show the results of our baseline evaluation. We observe that the throughput of FASP is higher than FCEP for all patterns. While for SEQ_1 and $ITER_1^3$ the throughput of FASP is, on average, 28% higher (min. 3%, max. 33%), the throughput of $NSEQ_1$ differs severely, where FASP is up to 20x faster than FCEP. FASP-O1 (Interval Join) provides equivalent throughput for SEQ_1 and $ITER_4^3$ and shows a throughput drop compared to FASP by 50% for $NSEQ_1$. FASP-O2 (Aggregations) provides the highest throughput for $ITER_1^3$.

Discussion. We conclude that our elementary operator mappings outperform FCEP for patterns with small windows and low output selectivity. The significant performance differences for $NSEQ_1$ can be explained as follows. In contrast to the other patterns, $NSEQ_1$ requires events from an additional source, i.e., AQ-Data, that lead to an additional union for FCEP. The ordered-based evaluation mechanism of FCEP causes additional processing overhead by handling the negation constraint retrospectively. In particular, for $NSEQ_1$ (T_1;¬(T_2);T_3), initially, the matches of the $SEQ(T_1,T_3)$ are detected. Afterward, the negation constraint is evaluated for each match of $SEQ(T_1,T_3)$. This evaluation process requires buffering of events as well as calculating and pruning of partial matches $SEQ(T_1,T_3)$. Our mapping uses a UDF to identify whether a relevant event of the negated stream occurs after T_1 within W. Thus, our mapping circumvents the buffering of events and retrospective evaluation of the negation constraint. The lower throughput of FASP-O1 stems from the much higher frequencies of T_1 compared to T_3 for $NSEQ$. In contrast, SEQ_1 and $ITER_1^3$ use similar frequencies for all involved streams. Finally, FASP-O2 leverages the lightweight count aggregation to determine the number of occurring events for $ITER_1^3$.

5.2.2 *Impact of Pattern Parameters.*

In the following, we investigate the performance impact of three essential pattern parameters [54, 76], i.e., output selectivity σ_o, window size W, and pattern length n.

Selectivity: In this experiment, we determine the impact of increasing output selectivities on throughput and detection latency. We use SEQ_1 and increase its output selectivity σ_o from 0.003% up to 30% by varying the filter selectivity of the types Q and V.

Observations. In 3b, we present the throughput for increasing selectivities. First, we observe an initial throughput difference of 60% for $\sigma_o = 0.003\%$ between FASP and FCEP. Second, FCEP's throughput drops drastically for increasing selectivities, while FASP's throughput remains constant for $\sigma_o \leq 1\%$. Third, for $\sigma_o = 30\%$, FASP's throughput drops from 145k to 70k, while FCEP drops its throughput to below 500 tpl/s. Fourth, FASP-O1 (Interval Join) provides equivalent throughput results for selectivities up to 1%. For a selectivity of 30%, we observe that the Interval Join outperforms the Sliding Window Join (FASP) with a 27% higher throughput by circumventing duplicate calculations of overlapping windows. Finally, we turn toward the observed detection latency. In accordance with the throughput behavior, the latency of FCEP and FASP increases with higher selectivities. In particular, we observe an average latency of 414 ms for $\sigma_o = 0.003\%$ and 18 s for $\sigma_o = 30\%$ for FCEP. FASP provides an average latency of 240 ms up to $\sigma_o = 1\%$ and 2 s for $\sigma_o = 30\%$. FASP-O1 provides the lowest latency results with 75 ms over all runs.

Discussion. Higher selectivity increases the number of valid tuples. Thus, more (partial) matches are created, resulting in an increase in computational workload and a corresponding drop in performance. The severity FCEP is affected by increasing selectivities is critical because patterns usually contain various selections to define interesting behavior in the data [46]. Moreover, stream processing is dynamic, including a high correlation between events. Thus, high selectivities may appear during peak times when the system must detect matches efficiently. We observe that the throughput of FCEP decreases to below 10k tpl/s with a latency of 1 s for $\sigma_o = 1\%$, whereas in realistic scenarios, the selectivity can go as high as 10% [76]. In contrast, the effects of high selectivities on our mapping are less severe and enable more efficient pattern detection in dynamic stream processing scenarios with up to 150x higher throughput than FCEP.

Window Size: In this experiment, we examine the effect of varying window sizes W on throughput and detection latency. We use SEQ_1 and increment its window sizes W from 30 to 360.

Observations. In Figure 3c, we present the throughput for increasing window sizes. First, we observe an initial throughput difference of 40% for $W = 30$. Second, FCEP's throughput drops by 76% from window size 30 to 360, while the throughput of FASP is constant overall patterns. Third, FASP-O1 (Interval Join) provides equivalent throughput results as FASP for all window sizes. Finally, we turn toward the observed detection latency. The latency of FCEP increases with higher window sizes. In particular, we observe an average latency of 265 ms for $W = 30$ and 590 ms for $W = 360$. In contrast, both FASP and FASP-O1 provide a constant average latency over all runs, with 210 ms for FASP and 85 ms for FASP-O1.

Discussion. Larger windows prolong the lifetime of events and, thus, increase the state. The relaxed time constraints on pattern matches raise σ_o, i.e., from 0.00016% to 0.00032% for SEQ_1, primarily causing FCEP's throughput drop. For FASP, larger windows increase the window state and, due to the usage of sliding windows with a slide size equal to one minute, also the number of concurrent windows. However, since FASP and FASP-O1 perform similarly, we conclude that the overhead of concurrent windows and the state increase are neglectable for low selective workloads.

Pattern Length: We assess the performance implications of pattern length, i.e., the impact of an increased number of events contributing to a match. To this end, we examine how throughput behaves when dealing with nested $SEQ(n)$ and $ITER^m$ patterns.

Nested Sequence. In this experiment, we run five $SEQ_n(n)$ to investigate how the pattern length n affects throughput. We incrementally increase n from 2 to 6 by progressively combining all available event types from QnV- and AQ-Data (10M tuples (0.5GB)). For all SEQ_n, we use an output selectivity $\sigma_o = 0.00032\%$ and a window size $W = 15$.

Observations. In Figure 3d, we present the throughput for increasing pattern lengths n. We observe that FCEP is severely affected, dropping its throughput by 80% from SEQ_2 to SEQ_3 and 50% from SEQ_4 to SEQ_5. In contrast, FASP and FASP-O1 provide stable throughput for all patterns with a 13x higher throughput for pattern lengths beyond 4.

Discussion. Nested $SEQ(n)$ compose events from n event types contained in potentially different streams. The drop in throughput for FCEP is primarily due to the additional union of these streams. In particular, we introduce the stream of $SDS011$ sensors ($PM10$ and $PM2.5$) for running SEQ_3 and SEQ_4, and the stream of $DHT22$ sensors ($Temp$ and Hum) for creating SEQ_5 and SEQ_6. Since σ_o remains constant for all SEQ_n, FCEP yields nearly identical throughput results for patterns using the same number of sources, such as SEQ_3 to SEQ_4 and SEQ_5 to SEQ_6. In contrast, our mapping decomposes nested $SEQ(n)$s into n-1 consecutive joins. Thus, it avoids the union of input streams and leverages pipeline parallelism, enabling it to maintain a consistent throughput for extended nested patterns.

Iteration. In this experiment, we increase the pattern length m from 3 to 9 for $ITER^m$ using 10M tuples of QnV-Data. In particular, we use the event type V and run $ITER_2^m$ with a constraint between subsequent events, i.e., $v_n.value < v_{n+1}.value$, and $ITER_3^m$ with a threshold filter, i.e., $v_n.value < threshold$. For all patterns, we use an output selectivity $\sigma_o = 0.003\%$ and a window size $W = 15$.

Observations. We present the throughput for increasing pattern lengths m for $ITER_2^m$ in Figure 3e and for $ITER_3^m$ in Figure 3f. We observe that FCEP decreases its throughput with increasing m regardless of the applied constraint. However, this effect is less pronounced for $ITER_3$, which involves threshold filters. Conversely, FASP and its optimizations provide similar throughput, which remains consistent for all $ITER^m$ and is up to 15x higher than FCEP.

Discussion. The higher the pattern length m, the more events need to occur in W to form a match of $ITER^m$. To maintain a constant σ_o for all $ITER^m$, we increase the selectivity of the constraints for all m. Thus, more relevant events occur that need to be kept in the operator state. For this reason, FCEP decreases its throughput for higher m. Constraints between subsequent events are more restrictive than threshold filters and additional force the testing of the ancestor event in the partial match, contributing to the greater throughput decrease. In contrast, FASP maintains a constant throughput for all $ITER^m$, irrespective of the constraints and optimizations applied. FASP and FASP-O1 benefit from breaking the pattern into m-1 joins, while FASP-O2 employs an aggregation to approximate $ITER^m$ patterns.

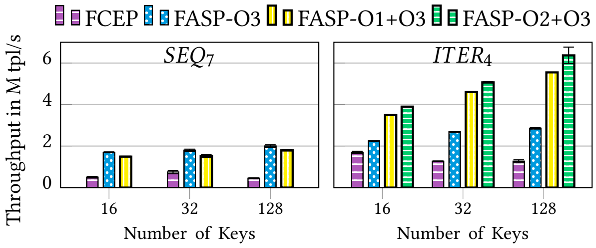


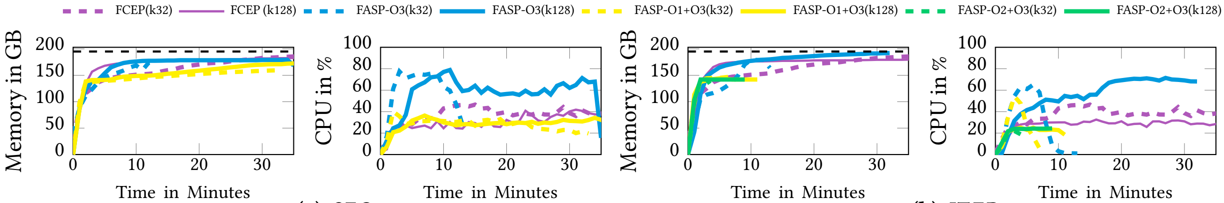
Figure 4: Impact of varying data characteristics.

5.2.3 *Data Characteristics.*

In this experiment, we measure the performance impact of different data characteristics on FCEP and FASP. To this end, we enable data partitioning by key and thus FASP-O3, i.e., the usage of Equi Joins. As a result, the CEP operator in FCEP and stateful ASP operators run in parallel. We run two patterns, i.e., $SEQ_7(3)$ with an output selectivity $\sigma_o = 1\%$ and a window size $W = 15$, and $ITER_4^4(1)$ with an output selectivity $\sigma_o = 1\%$ and a window size $W = 90$, using QnV- and AQ-Data samples. We use the sensor *id* as a key attribute and to control the data characteristics, i.e., each sensor increases the data volume (approx. 8.5M tuples (0.35GB) per event type) and the number of keys. We evaluate the throughput for both patterns on one worker with 16 task slots.

Observations. In Figure 4, we show the impact of an increasing number of keys on the throughput. Our observations are five-fold. First, comparing the overall performance of SEQ_7 and $ITER_4$, we observe that both approaches decrease throughput when multiple streams are involved and the data volume is higher. In particular, FCEP is severely affected by an throughput drop of 70%, while FASP drops by 40% for 16 keys. Second, for all workloads, we observe that our approach outperforms FCEP. Third, with respect to the increasing number of keys, we observe that FCEP stagnates or even drops its throughput for data characteristics beyond 16 keys, i.e., cases where the number of keys is larger than the number of available task slots. In contrast, all FASP queries slightly increase their throughput by, on average, 15% from 16 to 128 keys for SEQ_7 and 30% from 16 to 128 keys for $ITER_4$. Fourth, with respect to O1 (Interval Joins), we observe that its throughput is below FASP-O3 for SEQ_7 while it outperforms it for $ITER_4$. With respect to O2 (Count Aggregations), we observe that it outperforms all approaches for $ITER_4$. Fifth, while exploring the throughput for FCEP, we observe that FCEP is severely affected by high ingestion rates. In particular, we encounter execution failure due to memory exhaustion for any ingestion rate higher than 1.3M tpl/s.

Discussion. Our experiment shows that both approaches leverage key partitioning. However, our mapping outperforms FCEP by an, on average, 60% higher throughput (min. 25%, max. 80%). Furthermore, while FCEP fails for ingestion rates beyond 1.3M tpl/s, our mapping handles ingestion rates in the range of 2.4M (FASP-O3) to 6.8M tpl/s (FASP-O2+O3). Furthermore, our evaluation reveals that the two suggested window types for joins, Sliding Window Joins (FASP-O3) and Interval Joins (FASP-O1+O3), leverage distinct data and pattern characteristics. First, the Interval Join creates windows only when events occur, benefiting from reordering streams based on their frequency to reduce window creation for less frequent streams. This can lead to superior performance in scenarios like $ITER_4$, where each join decreases the output frequency. In contrast, Sliding Window Joins cannot reduce windows for subsequent Self Joins, limiting their performance. Second, the



(a) SEQ_7. **(b)** $ITER_4$.

Figure 5: Resource usage for SEQ_7 and $ITER_4$.

small slide size of our mapping has a negative performance impact on larger windows. In particular, the number of concurrent windows, duplicate computations, and the operator state increase, leading to a maintenance overhead that limits the throughput for FASP-O3, as shown in $ITER_4$. In contrast, for small window sizes, as applied in SEQ_7, the late creation of windows delays the Interval Join and leads to the slight performance benefit of the Sliding Window Join. Note that our mapping allows us to reorder joins and combine Sliding Window Joins and Interval Joins in one query. Furthermore, FASP-O2 can leverage the lightweight ASP count aggregation to determine the number of occurring events for a single stream in a specified time window and can, therefore, outperform FCEP and other mapping solutions.

5.2.4 Resources Utilization. In this experiment, we investigate the resource utilization, i.e., CPU and memory usage, of FCEP and FASP for SEQ_7 and $ITER_4$ using 32 and 128 keys.

Observations. In Figure 5, we show the measured CPU and memory usage with 32 and 128 keys for both patterns, i.e., Figure 5a for SEQ_7 and Figure 5b for $ITER_4$. First, we observe that the memory usage of FCEP is equivalent to or higher than the memory usage of FASP even though FCEP's ingestion rate is at least 50% lower. Second, we observe that all approaches do not fully exploit available CPU resources, whereas FASP-O3, which constantly creates and processes sliding windows, has the highest CPU consumption.

Discussion. Our evaluation shows that the performance of all approaches depends on the available memory resources. Whereas FASP leverages available memory to cope with higher ingestion rates, the high memory consumption of FCEP is caused by the usage of its stateful model, i.e., the NFA. In particular, the unary CEP operator of FCEP uses implicit windowing. To this end, the operator is required to maintain partial matches, i.e., discard outdated partial matches that cannot lead to a full match anymore or keep them otherwise. This cumbersome maintenance process leads to high memory consumption, which is the reason for the observed performance bottleneck of FCEP in Section 5.2.3. In particular, with an ingestion rate of 1.3M tpl/s, FCEP demands almost all available memory. If higher ingestion rates occur, Flink throttles the sources to prevent memory exhaustion, as the data cannot be processed at the speed of the ingestion rate. This behavior is known as backpressure and is prevented by determining the maximal sustainable throughput the system can provide without creating any backpressure [53]. However, as the memory usage of FCEP still increases as the operator state grows while processing, the system fails due to memory exhaustion, as observed in Section 5.2.3. As opposed to FCEP, our mapping leverages explicit windowing to discard processed tuples efficiently. As a result, it utilizes the available resources more efficiently and can thus support high ingestion rates.

5.2.5 Scalability. In this experiment, we run the workloads of SEQ_7 and $ITER_4$ with the data characteristic of 128 keys (appr. 45GB for $ITER_4$ and 135GB for SEQ_7) using QnV- and AQ Data. We

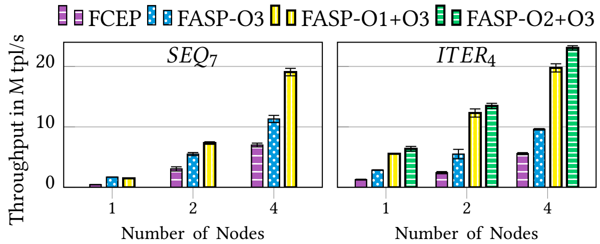


Figure 6: Scalability.

scale out to four workers and increase the number of parallel tasks by 16 slots per worker to assess the scalability of both approaches.

Observations. In Figure 6, we show the impact on throughput for increasing numbers of workers. Our observations are two-fold. First, both approaches leverage the additional memory and successfully increase throughput. Second, FCEP benefits most from the additional memory and increases its throughput by up to 6x (min. 1.4x, avg. 3.2x), while FASP increases, on average, by 2.6x (min. 1.7x, max.4.2x). However, FCEP cannot reach the throughput of our mapping, which is, on average, 60% (min. 24%, max. 82%) higher.

Discussion. Our results suggest that both approaches scale out over several nodes and leverage additional resources to increase their performance. However, FCEP is not capable of reaching the throughput obtained with our mapping, which decapsulates the pattern workload into multiple operators and thus leverages both key partitioning and pipeline parallelism. To this end, the usage of our mapping is more robust in the presence of high ingestion rates, leverages the workload distribution over multiple operators and the reordering of operators.

6 RELATED WORK

In this section, we contrast our mapping to related work in the intersection between CEP and ASP.

The origin of CEP and ASP: Although both paradigms, ASP and CEP, introduce different flavors of stream processing [60], both address the need to process streams instead of bounded batches. Thus, they share a common history visualized in Figure 7. As depicted on the left, in the early 00s, fundamental concepts to handle unbounded data streams and continuous queries were introduced in seminal SPSs, e.g., STREAM [20], Aurora [10], Borealis [9], or TelegraphCQ [31]. These systems provide essential stream processing features but lack CEP requirements to specify the time and cause relationships between events [36, 61], e.g., before 2016, most ASPSs did not provide windowing by event time [14], or dealt with out-of-order arrivals [26, 60, 69]. Thus, some years later, the first generation of CEP systems appeared with common representatives such as SASE [49], ZStream [64], and Esper [40]. In contrast to ASP, CEP is highly influenced by a diversity of other research lines (integrated by jump-in arrows in Figure 7) such as seminal ASPSs (CEDR [22]), active databases [29, 68], pub/sub-systems

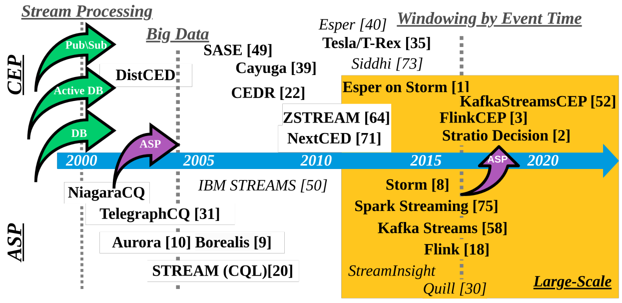


Figure 7: History of ASP and CEP systems.

(Cayuga [39]), or logical Prolog-based systems (RTEC [21]). This diversity leads to heterogeneous CEP solutions with distinct languages and different pattern detection mechanisms.

After the introduction of MapReduce [38] in 2004, modern ASPSs evolved for processing large data volumes [26] and scale-out over hundreds of nodes in cloud environments. Prominent examples are open-source projects, e.g., Flink [18], Spark [75], or Kafka Streams [58], as well as commercial systems such as Microsoft Quill [30] and IBM Streams [50]. As opposed to ASP, traditional CEP systems are optimized for single-node execution or centralized architectures with serial processing models, which are cumbersome for parallel and distributed processing. Thus, these solutions provide limited resource capacities for the ever-increasing data volume and velocity [26, 42]. Some CEP solutions tackle these limitations by enabling horizontal scaling over several nodes and messaging as communication strategy to handle missing shared memory, e.g., T-Rex [35] or NextCED [71]. Most distributed CEP solutions are mainly research prototypes that focus on parallelizing ordered-based evaluation mechanisms on homogeneous cluster environments without considering network bandwidth or high ingestion rates [35, 71]. Other limitations are the missing control over the degree of parallelism, which prevents load balancing [42] and multi-query optimization for serial processing models [24]. In sum, no CEP system exists that provides a complete set of features, i.e., parallel processing, flexible resource allocation, window distribution, and multi-query optimization, to fully leverage cloud environments with potentially unlimited scaling capabilities [26, 42, 44].

ASPSs with CEP Features: Two approaches exist that enable CEP on cloud-optimized ASPSs: Approach (1) proposes to run instances of CEP systems on worker nodes of an ASP-manged cluster, i.e., Stratio Decision [2]. Approach (2) is to provide CEP functionality as an additional unary operator of the ASPS. *Esper on Storm* [1] with Esper running in Storm Bolts, as well as Flink and KafkaStreams with built-in support for CEP (FlinkCEP [3], KafkaStreamsCEP [52]) are representatives of approach (2). Both approaches leverage the cloud-optimized data gathering and distribution features of the underlying ASPS. However, all solutions bring along the limitations of their order-based evaluation mechanisms, e.g., multi-pattern optimization or parallel execution, and design shortcomings, i.e., an unary CEP operator. In contrast, we solve the problem independent of traditional CEP systems by translating CEP patterns into ASP queries. Thus, our derived mapping overcomes the limitations of traditional CEP detection mechanisms and leverages the provided optimization of the state-of-the-art ASPS. To the best of our knowledge, we are the first to propose a general mapping of CEP operators towards ASP to enable large-scale distributed CEP. Nevertheless, the relationship between both paradigms has also been studied from the opposite direction, i.e., CEP systems that leverage ASP features.

CEP Systems with ASP Features: ZStream [64] introduces the tree-based pattern detection mechanism for CEP by implementing its operators as join variants. Its advantage compared to the common order-based detection mechanism is the possibility to optimize pattern detection plans, i.e., the order of event type compositions. By using tree-based pattern plans and referring to its operator implementation as join variants, ZStream is in line with our findings. In contrast to our mapping, ZStream does not allow for parallel execution or multi-pattern optimization, which prevents its use for cloud environments and large data volumes. One recent theoretical and experimental study by Kolchinsky and Schuster [55], proves the equivalence of CEP pattern plan and multi-join query plan generation. In particular, they observe that tree-based pattern detection plans and logical join query plans look alike and apply multi-join optimization techniques for pattern plans. In contrast to our solution, Kolchinsky and Schuster use the inverted direction and apply database optimizations on CEP systems. Our mapping is based on different join types and profits from available optimization in the target domain, as well as manual reordering based on known data characteristics.

7 CONCLUSION

In this paper, we investigated how to efficiently combine ASP and CEP in one HSPS for distributed cloud environments. To this end, we first show that CEP and ASP have a joint base for operations on sets of time-stamped tuples that allow the mapping between their operators. Second, we derive formal definitions for the complete set of CEP operators proposed in SEA and map each operator to its ASP counterpart. As a result, we enable common ASPSs to provide a wide range of CEP functionality. Our evaluation shows that our mapping outperforms the state-of-the-art solution FlinkCEP under various parameters, data characteristics, and distributed settings with an, on average, 60% higher throughput and up to 6x higher ingestion rates. To do so, our mapping decomposes the pattern workload into multiple operators and leverages explicit windowing, whereas FlinkCEP composes an entire pattern into a single operator with a stateful model. Thus, our mapping leverages pipeline parallelism and provides high throughput for challenging workloads with high selectivities or ingestion rates. In contrast, FlinkCEP massively decreases its throughput or even fails the entire execution for such workloads due to its excessive memory consumption and garbage collection stalls. As a result, our mapping empowers common general-purpose ASPS to operate as HSPS that efficiently evaluate CEP patterns in distributed settings and on a large scale, leveraging their cloud optimizations.

Future work in this area might target the specification of a PSL for Big Data and the IoT combined with a parser that automatically transforms declarative patterns into their respective execution pipeline using the API of the ASPS. Furthermore, collecting information on data and pattern characteristics such as frequency and selectivity enables the automated application of the proposed optimization opportunities. Finally, our formal definitions may encourage engineers to implement CEP-specific join variants, such as the Interval Join, to optimize ASPSs for CEP workloads.

ACKNOWLEDGMENTS

This work was funded by the German Federal Ministry of Education and Research as BIFOLD - Berlin Institute for the Foundations of Learning and Data (ref. 01IS18025A and ref. 01IS18037A). We thank the NebulaStream team for their insightful comments and fruitful discussions.

REFERENCES

[1] 2012. Esperonstorm. Accessed Mar. 2023: https://github.com/tomdz/storm-esper.

[2] 2016. Stratio Decision. Accessed Mar. 2023: https://github.com/Stratio/Decision.

[3] 2019. FlinkCEP - Complex event processing for Flink. Accessed May 2023: https://ci.apache.org/projects/flink/flink-docs-stable/dev/libs/cep.html.

[4] 2022. Apache Beam. Accessed Mar. 2023: https://beam.apache.org.

[5] 2022. Apache Flink. Accessed Sept. 2023: https://flink.apache.org.

[6] 2022. Apache Kafka. Accessed Jan. 2023: https://kafka.apache.org.

[7] 2022. Apache Spark. Accessed Jan. 2023: https://spark.apache.org.

[8] 2022. Apache Storm. Accessed Jan. 2023: https://storm.apache.org.

[9] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley B. Zdonik. 2005. The Design of the Borealis Stream Processing Engine. In *Second Biennial Conference on Innovative Data Systems Research, CIDR 2005, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*. www.cidrdb.org.

[10] Daniel J. Abadi, Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. 2003. Aurora: a new model and architecture for data stream management. *VLDB J.* (2003).

[11] Arif Ahmed, HamidReza Arkian, Davaadorj Battulga, Ali J. Fahs, Mozhdeh Farhadi, Dimitrios Giouroukis, Adrien Gougeon, Felipe Oliveira Gutierrez, Guillaume Pierre, Paulo R. Souza Jr., Mulugeta Ayalew Tamiru, and Li Wu. 2019. Fog Computing Applications: Taxonomy and Requirements. *CoRR* (2019).

[12] Mert Akdere, Ugur Çetintemel, and Nesime Tatbul. 2008. Plan-based complex event detection across distributed sources. *Proc. VLDB Endow.* (2008).

[13] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. 2015. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *Proc. VLDB Endow.* (2015).

[14] Tyler Akidau, Slava Chernyak, et al. 2018. *Streaming systems: the what, where, when, and how of large-scale data processing.* " O'Reilly Media, Inc.".

[15] Samira Akili. 2019. On the Need for Distributed Complex Event Processing with Multiple Sinks. In *Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems, DEBS 2019, Darmstadt, Germany, 2019*. ACM.

[16] Samira Akili and Matthias Weidlich. 2021. MuSE Graphs for Flexible Distribution of Event Stream Processing in Networks. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM.

[17] Elias Alevizos, Anastasios Skarlatidis, Alexander Artikis, and Georgios Paliouras. 2017. Probabilistic Complex Event Recognition: A Survey. *ACM Comput. Surv.* (2017).

[18] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. 2014. The Stratosphere platform for big data analytics. (2014).

[19] James F. Allen and George Ferguson. 1994. Actions and Events in Interval Temporal Logic. (1994).

[20] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. 2016. STREAM: The Stanford Data Stream Management System. In *Data Stream Management - Processing High-Speed Data Streams*, Minos N. Garofalakis, Johannes Gehrke, and Rajeev Rastogi (Eds.). Springer.

[21] Alexander Artikis, Marek J. Sergot, and Georgios Paliouras. 2015. An Event Calculus for Event Recognition. *IEEE Trans. Knowl. Data Eng.* (2015), 895–908.

[22] Roger S. Barga, Jonathan Goldstein, Mohamed H. Ali, and Mingsheng Hong. 2007. Consistent Streaming Through Time: A Vision for Event Stream Processing. In *Third Biennial Conference on Innovative Data Systems Research, CIDR 2007, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*. www.cidrdb.org.

[23] Edmon Begoli, Tyler Akidau, Fabian Hueske, Julian Hyde, Kathryn Knight, and Kenneth L. Knowles. 2019. One SQL to Rule Them All - an Efficient and Syntactically Idiomatic Approach to Management of Streams and Tables. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM.

[24] Lars Brenna, Johannes Gehrke, Mingsheng Hong, and Dag Johansen. 2009. Distributed event stream processing with non-deterministic finite automata. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems, DEBS 2009, Nashville, Tennessee, USA, July 6-9, 2009*, Aniruddha S. Gokhale and Douglas C. Schmidt (Eds.). ACM.

[25] Shijiu Cao, Haihong W, Meina Song, and Ken Zhang. 2018. Optimization of Data Distribution Strategy in Theta-join Process based on Spark. In *Proceedings of the 2018 2nd International Conference on Algorithms, Computing and Systems, ICACS 2018, Beijing, China, July 27-29, 2018*. ACM.

[26] Paris Carbone, Gábor E. Gévay, Gábor Hermann, Asterios Katsifodimos, Juan Soto, Volker Markl, and Seif Haridi. 2017. Large-Scale Data Stream Processing Systems. In *Handbook of Big Data Technologies*, Albert Y. Zomaya and Sherif Sakr (Eds.). Springer.

[27] Iliano Cervesato, Massimo Franceschet, and Angelo Montanari. 2000. A Guided Tour through Some Extensions of the Event Calculus. *Comput. Intell.* (2000).

[28] Iliano Cervesato and Angelo Montanari. 2000. A Calculus of Macro-Events: Progress Report. In *Seventh International Workshop on Temporal Representation and Reasoning, TIME 2000, Nova Scotia, Canada, 2000*. IEEE Computer Society.

[29] Sharma Chakravarthy, V. Krishnaprasad, Eman Anwar, and S.-K. Kim. 1994. Composite Events for Active Databases: Semantics, Contexts and Detection. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo (Eds.). Morgan Kaufmann.

[30] Badrish Chandramouli, Raul Castro Fernandez, Jonathan Goldstein, Ahmed Eldawy, and Abdul Quamar. 2016. Quill: Efficient, Transferable, and Rich Analytics at Scale. *Proc. VLDB Endow.* (2016).

[31] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Frederick Reiss, and Mehul A. Shah. 2003. TelegraphCQ: Continuous Dataflow Processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, Alon Y. Halevy, Zachary G. Ives, and AnHai Doan (Eds.). ACM.

[32] Jianxia Chen, Lakshmish Ramaswamy, David K. Lowenthal, and Shivkumar Kalyanaraman. 2012. Comet: Decentralized Complex Event Detection in Mobile Delay Tolerant Networks. In *13th IEEE International Conference on Mobile Data Management, MDM 2012, Bengaluru, India, July 23-26, 2012*, Karl Aberer, Anupam Joshi, Sougata Mukherjea, Dipanjan Chakraborty, Hua Lu, Nalini Venkatasubramanian, and Salil S. Kanhere (Eds.). IEEE Computer Society.

[33] E. F. Codd. 1972. Relational Completeness of Data Base Sublanguages. *Research Report / RJ / IBM / San Jose, California* (1972).

[34] E. F. Codd. 1983. A Relational Model of Data for Large Shared Data Banks (Reprint). *Commun. ACM* (1983).

[35] Gianpaolo Cugola and Alessandro Margara. 2012. Complex event processing with T-REX. *J. Syst. Softw.* (2012).

[36] Gianpaolo Cugola and Alessandro Margara. 2012. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.* (2012).

[37] Gianpaolo Cugola and Alessandro Margara. 2015. The Complex Event Processing Paradigm. In *Data Management in Pervasive Systems*, Francesco Colace, Massimo De Santo, Vincenzo Moscato, Antonio Picariello, Fabio Alberto Schreiber, and Letizia Tanca (Eds.). Springer.

[38] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, Eric A. Brewer and Peter Chen (Eds.). USENIX Association.

[39] Alan J. Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, and Walker M. White. 2007. Cayuga: A General Purpose Event Monitoring System. In *Third Biennial Conference on Innovative Data Systems Research, CIDR 2007, Asilomar, CA, USA, 2007, Online Proceedings*. www.cidrdb.org.

[40] EsperTech. 2006. Complex Event Processing Streaming Analytics. Accessed Aug. 2023: http://www.espertech.com/.

[41] Wang Fengjuan, Zhang Xiaoming, et al. 2013. The research on complex event processing method of internet of Things. In *ICMTMA*. IEEE.

[42] Ioannis Flouris, Nikos Giatrakos, Antonios Deligiannakis, Minos N. Garofalakis, Michael Kamp, and Michael Mock. 2017. Issues in complex event processing: Status and prospects in the Big Data era. *J. Syst. Softw.* (2017).

[43] Antony Galton and Juan Carlos Augusto. 2002. Two Approaches to Event Definition. In *Database and Expert Systems Applications, 13th International Conference, DEXA 2002, Aix-en-Provence, France, September 2-6, 2002, Proceedings (Lecture Notes in Computer Science)*, Abdelkader Hameurlain, Rosine Cicchetti, and Roland Traunmüller (Eds.). Springer.

[44] Nikos Giatrakos, Elias Alevizos, Alexander Artikis, Antonios Deligiannakis, and Minos N. Garofalakis. 2020. Complex event recognition in the Big Data era: a survey. *VLDB J.* (2020).

[45] Alejandro Grez, Cristian Riveros, and Martín Ugarte. 2017. Foundations of Complex Event Processing. *CoRR* (2017).

[46] Alejandro Grez, Cristian Riveros, and Martín Ugarte. 2019. A Formal Framework for Complex Event Processing. In *22nd International Conference on Database Theory, ICDT 2019, March 26-28, 2019, Lisbon, Portugal (LIPIcs)*, Pablo Barceló and Marco Calautti (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[47] Philipp M. Grulich, Sebastian Breß, Steffen Zeuch, Jonas Traub, Janis von Bleichert, Zongxiong Chen, Tilmann Rabl, and Volker Markl. 2020. Grizzly: Efficient Stream Processing Through Adaptive Query Compilation. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM.

[48] Daniel Gyllstrom, Jagrati Agrawal, Yanlei Diao, and Neil Immerman. 2008. On Supporting Kleene Closure over Event Streams. In *Proceedings of the 24th International Conference on Data Engineering, ICDE, 2008, Cancún, Mexico*, Gustavo Alonso, José A. Blakeley, and Arbee L. P. Chen (Eds.). IEEE Computer Society.

[49] Daniel Gyllstrom, Eugene Wu, Hee-Jin Chae, Yanlei Diao, Patrick Stahlberg, and Gordon Anderson. 2006. SASE: Complex Event Processing over Streams. *CoRR* abs/cs/0612128 (2006).

[50] Martin Hirzel, Henrique Andrade, et al. 2009. SPL stream processing language specification. *IBM Research Report: RC24897 (W0911 044)* (2009).

[51] Martin Hirzel, Robert Soulé, Scott Schneider, Bugra Gedik, and Robert Grimm. 2013. A catalog of stream processing optimizations. *ACM Comput. Surv.* (2013).

[52] Florian Hussonnois. 2018. Complex Event Processing on top of Kafka Streams Processor API. Accessed July 2023: https://github.com/fhussonnois/kafkastreams-cep.

[53] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. 2018. Benchmarking Distributed Stream Data Processing Systems. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society.

[54] Ilya Kolchinsky and Assaf Schuster. 2018. Efficient Adaptive Detection of Complex Event Patterns. *Proc. VLDB Endow.* (2018).

[55] Ilya Kolchinsky and Assaf Schuster. 2018. Join query optimization techniques for complex event processing applications. *Proc. VLDB Endow.* (2018).

[56] Ilya Kolchinsky and Assaf Schuster. 2019. Real-Time Multi-Pattern Detection over Event Streams. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM.

[57] Robert A. Kowalski and Marek J. Sergot. 1986. A Logic-based Calculus of Events. *New Gener. Comput.*

[58] Jay Kreps. 2016. Introducing Kafka Streams: Stream Processing Made Simple. Accessed Jan. 2023: https://www.confluent.io/blog/introducing-kafka-streams-stream-processing-made-simple/.

[59] Samuele Langhi, Riccardo Tommasini, and Emanuele Della Valle. 2020. Extending Kafka Streams for Complex Event Recognition. In *2020 IEEE International Conference on Big Data (IEEE BigData 2020), Atlanta, GA, USA, December 10-13, 2020*, Xintao Wu, Chris Jermaine, Li Xiong, Xiaohua Hu, Olivera Kotevska, Siyuan Lu, Weija Xu, Srinivas Aluru, Chengxiang Zhai, Eyhab Al-Masri, Zhiyuan Chen, and Jeff Saltz (Eds.). IEEE.

[60] David Luckham. 2019. What's the Difference Between ESP and CEP? Accessed June 2023: https://complexevents.com/2019/07/15/whats-the-difference-between-esp-and-cep-2/.

[61] David C. Luckham. 2008. The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Springer.

[62] MBA Prashan Madumal et al. 2016. Adaptive event tree-based hybrid CEP computational model for Fog computing architecture. In *ICTer*. IEEE.

[63] mCloud. 2015. Public Data in Motion (Öffentliche Daten in Bewegung). Accessed Dez. 2023: https://www.mcloud.de/.

[64] Yuan Mei and Samuel Madden. 2009. ZStream: a cost-based query processor for adaptively detecting composite events. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul (Eds.). ACM.

[65] Mauro Negri, Giuseppe Pelagatti, and Licia Sbattella. 1991. Formal Semantics of SQL Queries. *ACM Trans. Database Syst.* (1991).

[66] Guadalupe Ortiz, Adrian Bazan-Muñoz, et al. 2023. Evaluating the integration of Esper complex event processing engine and message brokers. *PeerJ Computer Science* (2023).

[67] Adrian Paschke. 2006. ECA-RuleML: An Approach combining ECA Rules with temporal interval-based KR Event/Action Logics and Transactional Update Logics. *CoRR* (2006).

[68] Norman W Paton and Oscar Díaz. 1999. Active database systems. *ACM Computing Surveys (CSUR)* (1999).

[69] Srinath Perera and Sriskandarajah Suhothayan. 2015. Solution patterns for realtime streaming analytics. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems, DEBS '15, Oslo, Norway, June 29 - July 3, 2015*, Frank Eliassen and Roman Vitenberg (Eds.). ACM.

[70] Lana Ramjit, Matteo Interlandi, Eugene Wu, and Ravi Netravali. 2019. Acorn: Aggressive Result Caching in Distributed Data Processing Frameworks. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019*. ACM, 206–219.

[71] Nicholas Poul Schultz-Møller, Matteo Migliavacca, and Peter R. Pietzuch. 2009. Distributed complex event processing with query rewriting. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems, DEBS 2009, Nashville, Tennessee, USA, July 6-9, 2009*, Aniruddha S. Gokhale and Douglas C. Schmidt (Eds.). ACM.

[72] SENSOR.COMMUNITY. 2022. Global Sensornetzwork. Accessed Jan. 2022: https://sensor.community/de/.

[73] Sriskandarajah Suhothayan, Kasun Gajasinghe, Isuru Loku Narangoda, Subash Chaturanga, Srinath Perera, and Vishaka Nanayakkara. 2011. Siddhi: a second look at complex event processing architectures. In *Proceedings of the 2011 ACM SC Workshop on Gateway Computing Environments, GCE 2011, Seattle, WA, USA, 2011*, Rion Dooley, Sandro Fiore, Mark L. Green, Cameron Kiddle, Suresh Marru, Marlon E. Pierce, Mary P. Thomas, and Nancy Wilkins-Diehr (Eds.). ACM.

[74] Eugene Wu, Yanlei Diao, and Shariq Rizvi. 2006. High-performance complex event processing over streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis (Eds.). ACM.

[75] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: a unified engine for big data processing. (2016).

[76] Haopeng Zhang, Yanlei Diao, and Neil Immerman. 2014. On Complexity and Optimization of Expensive Queries in Complex Event Processing. In *SIGMOD*.

[77] Haopeng Zhang, Yanlei Diao, and Neil Immerman. 2014. On complexity and optimization of expensive queries in complex event processing. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu (Eds.). ACM.

[78] Shuhao Zhang, Hoang Tam Vo, Daniel Dahlmeier, and Bingsheng He. 2017. Multi-Query Optimization for Complex Event Processing in SAP ESP. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*. IEEE Computer Society.