Bridging the Gap: Complex Event Processing on Stream Processing Systems

Ariane Ziehn¹, Philipp M. Grulich¹, Steffen Zeuch¹, Volker Markl¹²
TU Berlin¹, DFKI GmbH², Germany
firstname.lastname@tu-berlin.de, firstname.lastname@dfki.de

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 support. 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 (ASPs), e.g., Flink [18], or Spark [75]. ASPs 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 ASPs 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 ASPs [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 ASPs [44]. The first approach, Stratio Decision [2], runs instances of CEP systems on worker nodes of an ASP-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, HSPs 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 CEP 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 ASPSs (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 data model. Users submit continuous requests to the SPS using its 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 processing model and 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.

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 $(a_1, ..., a_n)$, and all tuples $t_j \in S$ 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 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 $e = (T_1, ..., T_n)$ be the universe of event types and each event $e$ an instantiation of an event type $T_j \in e$ [15]. The event type can be either provided as an attribute or needs to be inferable [55]. We write $e \in e$ for the event type of $e$. Furthermore, the events of the output stream are matches of the pattern, i.e., compositions of the events $e_j$ that participated in the pattern detection process [12, 37, 44, 74, 76]. In particular, each match $M$ is a tuple $ce(e_1, ..., e_n, T_1, T_n)$, where for each pair $(e_i, e_j)$ it is true that $|e_i.ts - e_j.ts| < W$ [44]. Furthermore, $ce.T_1$ and $ce.T_n$ are the timestamps of the first and last occurred event in $M$.

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), ASPSs 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. Listing 2: Example pattern.

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, se-
quence, conjunction, disjunction, iteration, and negation. From a
unified perspective, the following two operators are semantically
equivalent in both stream processing paradigms:

(1) Selection \( \sigma_q(t) \) (ASP: filter) returns an input tuple \( t \) if the
user-defined set of predicates \( \emptyset \) is fulfilled or discards \( t \) from fur-
ther processing [18, 75].

(2) Projection \( \Pi_{\theta}((a_1, \ldots, a_N)) \) (ASP: map)
transforms the schema and attribute values of \( q \) 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 dis-
joint 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.

1 Processing Model: The processing model of an SPS trans-
sforms the user-provided requests into an internal, logical represen-
tation, which is optimized and translated to physical tasks for
query execution in ASPS and pattern detection in CEP systems.

In ASPSs, each query consists of three components: sources, oper-
ators, 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 mech-
nisms, 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 oper-
ators, such as the sequence operator that accepts events occurring
in temporal order, are essential for the CEP paradigm. These opera-
tors resemble regular expressions and lead to the prominent usage
in temporal order, are essential for the CEP paradigm. These opera-
tors resemble regular expressions and lead to the prominent usage
in temporal order, are essential for the CEP paradigm. These opera-
tors resemble regular expressions and lead to the prominent usage

3.1 Formal Semantics and Modifications
As with the majority of PSLs [64, 76], SEA only provides infor-
mal semantics, i.e., verbal descriptions, of its operators. Our map-
ing 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 tradi-
tional 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 algebra 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 \( e \) 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]:

\[
T(ts) = \begin{cases} 
\text{true, if } e \in T \\ 
\text{false, else} 
\end{cases}
\]

(1)

In contrast to batch-optimized database systems, SPSs use win-
dowing 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 con-
trast, 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 \( W = t_s e - t_s b \). 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 \( e_i, ts \in [t_s b, t_s k] \). We modify Equation 1 as follows:

\[
[T]^{t_s b}_{t_s k} = \begin{cases} 
\text{true, if } \forall e_i \in T \land e_i.ts \in [t_s b, t_s k] \\
\text{false, else}
\end{cases} 
\]

(2)

Additionally, we adapt the output of the function \([T]^{t_s b}_{t_s k} \) (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:

\[
[T^e]^{t_s b}_{t_s k} = \{ e \in T \land e.ts \in [t_s b, t_s k] \}
\]

(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 the 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 \( e.ts \in [t_s b, t_s k] \) is assigned to the finite substream \( T_k \). Formally,

\[
[T]^{t_s b}_{t_s k} = T_k = \{ e \in T \land e.ts \in [t_s b, t_s k] \}
\]

(4)

where each finite substream \( T_k \) has a time interval \([t_s b, t_s k]\) with the window length \( W = t_s k - t_s b \) [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:

\[
T_{k+l} = [T]^{t_s b}_{t_s k+l}
\]

(5)

where \( t_s b + s \cdot l \) is the first time \( (k, l \in \mathbb{N}) \).

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 \texttt{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 \( S_k \).

**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 \( e_i.ts, e_j.ts \in [t_s b, t_s k] \). 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).** By definition, for every event pair \( (e_i, e_j) \) in \( ce \) it is true that \( \text{max}(e_i.ts, e_j.ts) - \text{min}(e_i.ts, e_j.ts) < W \) [44]. It follows that \( W - 1 \) is the maximal time difference between a pair in \( ce \). A match \( M \) contains a pair \( (e_i, e_j) \) which is \( W - 1 \) time units apart only if detected in \( S_k = [S]^{t_s b}_{t_s k} \) if \( \text{min}(e_i.ts, e_j.ts) = t_s b \). Otherwise \( t_s b + W - 1 + n > t_s k - 1 \rightarrow \text{max}(e_i.ts, e_j.ts) \notin [t_s b, t_s k] \). Hence, we must ensure that there exists a \( S_k \) 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 \( \text{min}(e_i.ts, e_j.ts) = e_i.ts = t_s a - 1 \). To detect \( (e_i, e_j) \), we need to ensure that \( \exists t_s b + l \) so that \( e_i.ts = t_s k - 1 \land e_i.ts = t_s b + l \). It follows that \( e_i.ts = t_s b + l, e_j.ts = t_s b + s \land W = l = t_s k - t_s b \). 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 the \( \forall e \in S \exists S_{k+l}(e.ts = t_s b + l) \).

---

1Formal 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 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 SE 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]:

\[
(T_1 \land T_2)(\mathcal{S}) := \exists s_p, s_m; T_1(s_p) \land T_2(s_m) \land \max(s_p, s_m) = ts
\]

In contrast, SEAs require the pattern to occur within \( W \), i.e., the time interval \([t_{b}, t_{e}]\). Therefore, we modify Equation 6 as follows:

\[
[(T_1 \land T_2)]_{t_{b}, t_{e}}^W := \exists s_p, s_m; T_1(s_p) \land T_2(s_m) \land ts \in [t_b, t_e]
\]

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:

\[
(T_1 \land T_2) := \exists s_p, s_m; T_1(s_p) \land T_2(s_m)
\]

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 \([t_{b}, t_{e}]\) or an empty set \( \emptyset \). The resulting equation of the conjunction operator is defined as follows:

\[
(T_1 \land T_2)^* = \{(e_1, e_2) \mid e_1 \in T_1 \land e_2 \in T_2\}
\]

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

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,

\[
(T_1, T_2)^* = \{(e_1, e_2) \mid e_1 \in T_1 \land e_2 \in T_2 \land e_1.ts < e_2.ts\}
\]

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 \( \text{SEQ}(T_1, \text{SEQ}(T_2, T_3)) \), can be simplified to \( \text{SEQ}(T_1, T_2, T_3) \).

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,

\[
(T_1 \lor T_2)^* = \{e \mid e \in T_1 \lor e \in T_2\}
\]

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,

\[
(T^m)^* = \{ (e_1, ..., e_m) \mid 1 \leq i \leq m : e_i \in T \land (e_1.ts < ... < e_m.ts) \}
\]

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,

\[
\neg(T_2)[T_1, T_3](\mathcal{S}) = (\exists s_{t_1})(\forall s_{t_2})(T_1(s_{t_1}) \lor \neg T_2(s_{t_2}) \land T_3(s_{t_1}) \land \neg((t_1 < t_2 < t_3) \rightarrow \neg(T_2(s_{t_2}) \lor T_3(s_{t_2}))))
\]

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., \( \text{SEQ}(T_1, T_3) \). 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., \( \neg[T^m] \). 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:

\[
\neg(T_2)[\neg(T_1)(T_2, T_3)]^* = \{(e_1, e_2) \mid e_1 \in T_1 \land e_2 \in T_3 \land (e_1.ts < e_2.ts \land \neg \exists e_{t_2} \in (e_1.ts, e_2.ts) : e_2 \in T_2)\}
\]

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 ASPs. 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 × [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) T₁ and T₂. 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 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.

Listing 4: AND query.

Sequence: Mapping. Our formal definition in Equation 10, is equivalent to the definition of the relation Theta Join ⊲ ⊳ using the order by time of both events as join predicate θ [33]. In particular, all event pairs (e₁,e₂) 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 ∪ [33]. The union operator unifies two input streams into a new one, i.e., T₁ and T₂ are unified to T₁∪₂. Each event e₁,₂ in T₁∪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 ⊲ with 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., (T₁;T₂), and the negated existential quantifier that requires the absence of any event e₁∈T₂ within the time interval (e₁;e₃;e₄;e₅). 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 T₁ and T₂. Then, we apply a UDF window function that, for each event e₁ ∈ T₁, finds (if it exists) the next occurrence of e₂ ∈ T₂ within the time interval W of the pattern. To this end, we add an additional timestamp attribute a₄ to each event e₁. If an event e₂ occurs after e₁ within W a₄ = e₂, else a₄ = e₁ + W indicating that no e₂ occurred. Afterward, we perform SEQ(T₁,T₃) with the additional selection a₄ < e₁ to guarantee that no event e₁ ∈ T₂ occurred in the interval (e₁;e₅;e₆;e₇).

Listing 5: NSEQ pattern.

Listing 6: NSEQ query.

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, T₃ 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.

Listing 8: SEQ query.
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 \( e_2.ts \in (e_1.ts + \text{lowerBound}, e_1.ts + \text{upperBound}) \), with bounds defined as time measurement [5]. The bounds only depend on the window size \( W \), thus configuring 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 \( e_1 \) from \( T_1 \).

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 is iterative with the SEA iteration operator. However, many order-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 \( m \). If \( n = m \), the pattern is fulfilled under the selection policy skip-till-any-match. However, we denote \( O2 \) as approximate because aggregations return only 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 \( \text{ITER} \) results is barely supported in traditional CEP systems, which makes it rather cumbersome to, for instance, derive the average of an attribute \( a_j \in T(a_1,\ldots,a_n) \) for all events \( e_i \in \text{CE} (e_1,\ldots,e_m) \) [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 ASPs 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 \( e_i.a_n < e_i.a_0 \) 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 [79].

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., \( e_1.a_1 = e_2.a_1 \), 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 are always preferable as join keys when using our operator mapping. Other constraints, such as $\theta$ 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. 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.

### 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, .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, OnV-Data 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) 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 ($\text{Temp}$) and humidity ($\text{Hum}$) 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 (tps/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 matches [76] to determine the output selectivity $\epsilon_p$ of a pattern in %. We give the window size $W$ in minutes, and, following our findings in Section 3.1, we 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

---

**Table 2: Operator Support of FCEP and FASP.**

<table>
<thead>
<tr>
<th></th>
<th>FASP</th>
<th>AND</th>
<th>SEQ</th>
<th>OR</th>
<th>ITER</th>
<th>NSEQ</th>
<th>SF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FASP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FCEP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

---

7The 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.
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., SEQ1 (2) and ITER1 (1) with a sample of 10 million tuples (10M tuples) from QnV-Data (0.89GB) and NSEQ1 (3) with a sample of 10M tuples (0.5GB) from QnV- and AQ-Data. For all patterns, we use an output selectivity $\sigma_0 = 0.00005\%$ 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 SEQ1 and $\text{ITER}^1$ the throughput of FASP is, on average, 28% higher (min. 3%, max. 33%), the throughput of NSEQ1 differs severely, where FASP is up to 20x faster than FCEP. FASP-O1 (Interval Join) provides equivalent throughput for SEQ1 and $\text{ITER}^1$ and shows a throughput drop compared to FASP by 50% for NSEQ1. FASP-O2 (Aggregations) provides the highest throughput for $\text{ITER}^1$.

Discussion. We conclude that our elementary operator mappings outperform FCEP for patterns with small windows and low output selectivity. The significant performance differences for NSEQ1 can be explained as follows. In contrast to the other patterns, NSEQ1 requires events from an additional source, i.e., AQ-Data, that lead to an additional union for FCEP. The ordered-index evaluation mechanism of FCEP causes additional processing overhead by handling the negation constraint retrospectively. In particular, for NSEQ1 (T1; ¬(T2; T3)) initially, the matches of the SEQ1(T1, T3) are detected. Afterward, the negation constraint is evaluated for each match of SEQ1(T1, T3). This evaluation process requires buffering of events as well as calculating and pruning of partial matches SEQ1(T1, T3). Our mapping uses a UDF to identify whether a relevant event of the negated stream occurs after T1 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 T1 compared to T3 for NSEQ1. In contrast, SEQ1 and $\text{ITER}^1$ use similar frequencies for all involved streams. Finally, FASP-O2 leverages the lightweight count aggregation to determine the number of occurring events for $\text{ITER}^1$.

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 $\sigma_0$, 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 SEQ1 and increase its output selectivity $\sigma_0$ from 0.003% up to 30% by varying the filter selectivity of the types Q and V.

Observations. In Figure 3b, we present the throughput for increasing selectivities. First, we observe an initial throughput difference of 60% for $\sigma_0 = 0.003\%$ between FASP and FCEP. Second, FCEP’s throughput drops drastically for increasing selectivities, while FASP’s throughput remains constant for $\sigma_0 \leq 1\%$. Third, for $\sigma_0 = 30\%$, FASP’s throughput drops from 145k to 70k, while FCEP drops its throughput to below 500 t/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 $\sigma_0 = 0.003\%$ and 18 s for $\sigma_0 = 30\%$ for FCEP. FASP provides an average latency of 240 ms up to $\sigma_0 = 1\%$ and 2 s for $\sigma_0 = 30\%$. 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 FASP decreases to below 10k t/s with a latency of 1 s for $\sigma_0 = 1\%$, 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 SEQ1 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 \( \sigma_m \), i.e., from 0.00016% to 0.00032% for SEQ, 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 negligible 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 \( \text{ITER}^m \) patterns.

Nested Sequence. In this experiment, we run five SEQ(\( n \)) 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 \( \sigma_a = 0.00052\% \) 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(5) and 50% from SEQ(5) to SEQ(6). 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 SD5011 sensors (PM10 and PM2.5) for running SEQ(2) and SEQ(4), and the stream of DHT22 sensors (Temp and Hum) for creating SEQ(5) and SEQ(6). Since \( \sigma_m \) remains constant for all SEQ(\( n \)), FCEP yields nearly identical throughput results for patterns using the same number of sources, such as SEQ(2) to SEQ(5) and SEQ(3) to SEQ(6). In contrast, our mapping decomposes nested SEQ(\( n \)) 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 \( \text{ITER}^m \) using 10M tuples of QnV-Data. In particular, we use the event type V and run \( \text{ITER}^m \) with a constraint between subsequent events, i.e., \( \sigma_v = \text{value} < \text{threshold} \), and \( \text{ITER}^m \) with a threshold filter, i.e., \( \sigma_v = \text{value} < \text{threshold} \). For all patterns, we use an output selectivity \( \sigma_a = 0.003\% \) and a window size \( W = 15 \).

Observations. We present the throughput for increasing pattern lengths \( m \) for \( \text{ITER}^m \) in Figure 3e and for \( \text{ITER}^m \) 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 \( \text{ITER}^1 \), which involves threshold filters. Conversely, FASP and its optimizations provide similar throughput, which remains consistent for all \( \text{ITER}^m \) 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 \( \text{ITER}^m \). To maintain a constant \( \sigma_m \), 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 \( \text{ITER}^m \), 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 \( \text{ITER}^m \) patterns.

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 Eqi Joins. As a result, the CEP operator in FCEP and stateful ASP operators run in parallel. We run two patterns, i.e., SEQ(3) with an output selectivity \( \sigma_a = 0.003\% \) and a window size \( W = 15 \), and \( \text{ITER}^5(1) \) with an output selectivity \( \sigma_a = 0.001\% \) 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(3) and \( \text{ITER}^5 \), 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(3) and 30% from 16 to 128 keys for \( \text{ITER}^5 \). Fourth, with respect to O1 (Interval Joins), we observe that its throughput is below FASP-O3 for SEQ(3) while it outperforms it for \( \text{ITER}^5 \). With respect to O2 (Count Aggregations), we observe that it outperforms all approaches for \( \text{ITER}^5 \). 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 \( \text{ITER}^5 \), where each join decreases the output frequency. In contrast, Sliding Window Joins cannot reduce windows for subsequent Self Joins, limiting their performance. Second, the...
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 ITER4. In contrast, for small window sizes, as applied in SEQ7, the latest 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 SEQ7 and ITER4 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 SEQ7 and Figure 5b for ITER4. 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 SEQ7 and ITER4 with the data characteristic of 128 keys (appr. 45GB for ITER4 and 135GB for SEQ7) using QwN- and AQ Data. We 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 90s, fundamental concepts to handle unbounded data streams and continuous queries were introduced in seminal SPSSs, 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 ASPs 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...
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 ASPs 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 ASPs 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.