TPStream: Low-Latency Temporal Pattern Matching on Event Streams

Michael Körber Nikolaus Glombiewski Bernhard Seeger
Database Systems Group, University of Marburg
{koerberm,glombien,seeger}@mathematik.uni-marburg.de

ABSTRACT

Complex Event Processing (CEP) has emerged as the state-of-the-art technology for continuously monitoring and analyzing streams of events in time-critical applications. The key feature in CEP is *sequential pattern matching* to detect a user-defined sequence of conditions on event streams. However, many CEP applications are not restricted to events only, but require native support for *situations* (aggregated event data lasting periods of time) and expressive temporal pattern matching among these situations. These important requirements regarding situations are not sufficiently addressed in the CEP literature so far.

In this paper we present *TPStream*, a novel event-processing operator for both deriving situations from event streams and detecting temporal patterns among situations. First, we provide a formal foundation of situations and *TPStream*. Then, we propose a low-latency algorithm for *TPStream* that delivers situations and temporal matches at the earliest possible point in time. Furthermore, we utilize a simple, yet effective cost model in order to adapt to changing workloads on the fly and with negligible cost for migrating operator states. The results of our experimental evaluation show that *TPStream* is capable of processing high-volume event streams with low latency and outperforms applicable CEP solutions from academia and industry.

1 INTRODUCTION

During the last decade, Complex Event Processing (CEP) has emerged to the technology of choice for analyzing massive streams of events in near real time. Typically, CEP systems detect composite (complex) events by combining, aggregating and filtering streams of simple or other composite events and report matches to registered event sinks. In turn, these sinks react to the detected event by triggering appropriate actions in a timely manner. CEP can be applied to a wide variety of application domains, including IT infrastructure monitoring, traffic monitoring, health care and financial applications.

Problem Statement: A noticeable fraction of currently available CEP systems is build upon point based temporal semantics. That is, each event is associated with a timestamp (e.g., the moment a measurement was made) and event streams are ordered accordingly. With only a *single* timestamp the expressible *temporal relations* between two events are limited to before/after/at the same time relationships. However, many real-world scenarios comprise the detection of complex *temporal patterns* among situations lasting for periods of time. Consider the following traffic-monitoring application:

Example: A traffic monitoring system is continuously receiving sensor data from connected cars (i.e., position, speed, acceleration). One of the systems goals is to notify drivers about potential dangers around their locations, such as an aggressively driving car. Among others, the American Automobile Association has identified the following two actions being indicators for aggressive driving[1]: *"Operating the vehicle in an erratic, reckless, careless, or negligent manner or suddenly changing speeds"* and *"Driving too fast for conditions or in excess of posted speed limit"*. From these definitions, a pattern to detect aggressive drivers could be stated as: "A sharp acceleration followed by hard braking, both accompanied by a period of speeding."

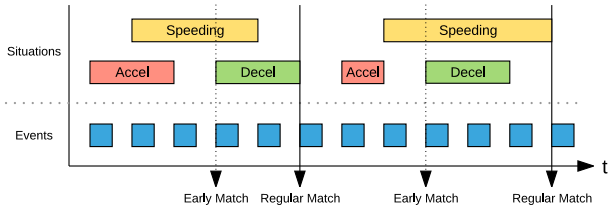

Figure 1: Detecting aggressive driving with situations

Challenges: This example illustrates three key-features a solution for temporal pattern matching on instantaneous events should provide: (i) Situations are derived from point events on the fly, by identifying subsequences of the input stream for which the defined condition holds true (e.g., `speed` $>$ `70 mph`). Optional constraints can be applied to restrict the duration of situations. In addition to its validity (expressed as a time interval), a situation is enriched with meaningful summarizations of underlying events (e.g., the average speed of the speeding phase). (ii) The pattern language offers support for alternatives. That is, the order of the situations' start and end points is not required to be fully fixed. Consider, for example, the two matches sketched in Figure 1: In the first match, the three defined situations *overlap*, while in the second match deceleration happens *during* the speeding situation. (iii) The pattern should be detected with the *lowest possible latency*. As depicted in Figure 1, both matches may be concluded at the beginning of the deceleration situation, since at this point in time speeding still holds true and the pattern allows any combination of their endpoints. Technically, this means the system should be able to conclude a successful match without exact knowledge about the validity of all situations.

State-of-the-Art: To the best of our knowledge, the only work on complex *temporal relations* in event stream pattern matching is the *ISEQ* operator [20]. However, *ISEQ* has several shortcomings concerning the desired features: First, the operator expects interval-events (i.e., situations) as input, leaving all aspects of (i) to an unspecified external entity. Being unaware of the origin of interval-events severally limits the operator in processing power (in terms of plan optimization) and most importantly renders a detection with the lowest possible latency (iii) impossible since there is no way to directly access an incomplete situation or indirectly manipulate the building of a situation through constraints.

[1] http://www.iii.org/fact-statistic/aggressive-driving

Second, a *temporal pattern* is specified using a conjunction of endpoint relationships (i.e., an ordering on start (ts) and end (te) of intervals). This way, alternatives are expressed by omitting one or more endpoints. For example, the pattern $A.ts < B.ts < A.te \leqslant B.te \vee A.ts < B.ts < B.te < A.te$ on two situations A and B is expressed as $A.ts < B.ts < A.te$. Hence, disjunctions like $A.ts < B.ts < A.te < B.te \vee B.ts < A.ts < B.te < A.te$ are not expressible in a single query. Instead, they require multiple queries in an approach without any specified optimization component to detect shared processing opportunities. Finally, *ISEQ* relies on auxiliary index structures and punctuation mechanisms for efficient query execution, complicating the integration into existing systems.

Straw Man's Approach: Besides *ISEQ*, we identified two approaches to solve the task of temporal pattern matching with point event streams. Thus, we can provide a point of comparison to CEP systems featuring pattern matching via regular expressions or equivalent techniques. The first approach works in two phases: In the first phase, a pattern matcher is deployed for each defined situation, computing its duration (start/end timestamp) and the desired aggregates. Technically, this means matching patterns of the form `!S S+ !S` with S being derived from the input stream using the situation's condition (e.g., $speed > 70\ mph$). This results in a dedicated stream per defined situation. Each of these streams is ordered according to the end timestamp, which allows to map the *temporal pattern* to a sequence of situations (reflecting the order of end timestamps, possibly containing alternatives). In the second phase, a dedicated pattern matching operator is used to find all matching sequences, whereby the proper ordering of start timestamps is checked via additional predicates. Even though this approach satisfies requirements (i) and (ii), it fails to produce early results (iii), because, just like in *ISEQ*, situations are fully derived before they are available for pattern matching.

The second approach uses a single pattern matching operator and expresses the *temporal pattern* as a single sequence of point events. To express temporal overlaps, the conditions of all involved situations must be connected via a logical AND. For example, `Acceleration overlaps Speeding` is expressed as `A B+ C` with the following conditions: $A : accel > 8\ m/s^2$, $C : speed > 70\ mph$ and $B : A \wedge C$. Since patterns are expressed on the granularity of events, early results (iii) are achieved, by simply omitting the last portion of the pattern. At the same time summarizations of single situations and the validation of duration constraints (i) are left to a post-processing step, since situations are disassembled to express temporal overlaps.

Solution: We present *TPStream*, a holistic operator for complex temporal pattern matching on point event streams. Compared to the presented approaches, our contributions are:

- *TPStream* is the first CEP operator to closely couple derivation of situations with pattern matching, enabling match detection at the *earliest possible point in time*.
- We also improve upon existing work on temporal pattern matching (e.g. *ISEQ*) by allowing arbitrary alternatives and duration constraints in pattern definitions.
- We introduce an optimizer component for interval-based pattern matching which continuously adapts its execution strategy to deal with fluctuating data rates and changes in the data distribution of incoming streams.
- *TPStream* provides native query support for temporal pattern matching, making it easier to formulate and read *temporal patterns* in comparison to most existing CEP solutions relying on a straw man's approach.
- Unlike *ISEQ*, the operator and its low latency optimizations can easily be implemented in commonly available point-based systems, because time-intervals are used only internally and results are again point event streams.
- In experiments, we show our latency improvements and the performance limits of two existing CEP solutions from academia and industry when handling situations. We present that *TPStream* can outperform these systems by an order of magnitude and that alternatives, which have great impact on the performance of sequential pattern matching, influence *TPStream*'s matching performance only marginally.

The rest of the paper is organized as follows. Section 2 reviews related work, before we introduce *TPStream*'s query language in section 3. In section 4 we model all aspects of *TPStream* in an algebra. Efficient evaluation strategies, the algorithm for low-latency matching and our optimization techniques are presented in section 5. We evaluate the performance of *TPStream* in section 6 and conclude this paper in section 7.

2 RELATED WORK

So far, native ways to work with situations in systems capable of CEP have not been sufficiently addressed. Nevertheless, the concept can be related to working with time intervals, aggregating information and performing temporal joins, all of which have seen recent contributions. Since these are broad areas, we will loosely group the most relevant approaches under three headlines: Event Pattern Matching, Context/State in CEP and Spatio-Temporal Database Systems.

Event Pattern Matching: Systems capable of CEP (e.g. [3, 9]) are generally closely associated with a pattern matching operator. [27] features a discussion on the several different semantics of the operator and a recent survey [12] covers several implementations, that employ different techniques such as NFAs or Graphs, each featuring their own unique optimization techniques. Regardless of specific details, most approaches focus on data referring to points in time and thus lack *native* capabilities to query complex relations between time intervals as stated in [2] - a crucial aspect for dealing with long-lasting situations. Cayuga [8], ZStream [22] and Microsoft StreamInsight [1] are well-known approaches that associate time intervals with data, showcasing the interest in working with interval-based events. ZStream in particular shares similarities with our join-based, adaptive processing approach for pattern matching. However, each of the respective pattern languages is based around a strictly sequential relation (interval *i* ends before interval *j* begins) and/or explicitly order-independent relations (conjunction, disjunction). Not only does this limit their respective algorithmic support for complex *temporal relations*, but, just like point-based systems, formulating derivation queries naturally leads to the straw man's approach mentioned above using Kleene Operators (or FOLDS in [8]). As we will show in our experiments, this approach results in significant performance deficiencies.

Context/State in CEP: There has been recent work on introducing *contexts* into a CEP environment. CAESAR [23] associates queries to long-lasting context windows, detects them from incoming events as soon as they start and suspends queries of inactive contexts. Similarly, contexts in [11] are used to group up event types to process them together. While contexts and

Relation (R)	Equivalent (R)	Visualization	Definition (δ_R)
A before B	B after A		A.ts < A.te < B.ts < B.te
A starts B	B started-by A		A.ts = B.ts < A.te < B.te
A meets B	B met-by A		A.ts < A.te = B.ts < B.te
A overlaps B	B overlapped-by A		A.ts < B.ts < A.te < B.te
A during B	B contains A		B.ts < A.ts < A.te < B.te
A finishes B	B finished-by A		A.ts < B.ts < A.te = B.te
A equals B			A.ts = B.ts < A.te = B.te

Table 1: Allen's Interval Algebra

```
FROM    CarSensors CS PARTITION BY CS.car_id
DEFINE  A AS CS.accel >  8m/s² at least 5s,
        B AS CS.speed > 70 mph between 4s AND 30s,
        C AS CS.accel < -9m/s² at least 3s
PATTERN A meets B;A overlaps B;A starts B;A during B
    AND C during B;B finishes C;B overlaps C;B meets C
    AND A before C
WITHIN  5 MINUTES
RETURN  first(B.car_id) AS id,
        avg(B.speed) AS avg_speed;
```

Listing 1: Agressive drivers query

situations are related concepts, the key difference is that contexts are purposefully decoupled from events. Therefore, it is not possible to query the relation of different contexts to each other. In contrast, *TPStream* focuses on efficient, adaptive and low-latency implementations of those *temporal relations*. Likewise, work on states [18] and on aggregating windows [14, 17] focuses on derivation, but lacks interval relations [2] or pattern matching.

Spatio-Temporal Database Systems: The spatial databases community studied the problem of *spatio-temporal pattern queries* (STPQ) in trajectory databases [10]. In general, these approaches cannot be directly applied to an event processing environment, because they are built on top of a *persistent* trajectory database model, where movement histories are already stored and indexed in the database. However, the design of [25] in particular served as a foundation for our proposed *TPStream* operator as *TPStream* adapts similar concepts of temporal predicates and constraints. Furthermore, our evaluation method is related to temporal joins (see [13] for an excellent survey), but as most of the work is not based in stream processing, unique and important issues such as continuously arriving data, continuous query optimization and early result detection are overlooked. In comparison to join algorithms on streams [6, 15] as well as adaptive approaches [5], *TPStream* combines both the derivation of situations and the detection of patterns. Thus, the operator can offer new techniques for early result detection unique to CEP-style pattern matching.

3 QUERY LANGUAGE

In order to express *temporal relations* between situations, we adopt Allen's Interval Algebra [2] depicted in Table 1 for two generic intervals `A` and `B`. Each interval has a starting point (`ts`) and an ending point (`te`), resulting in a total of four points. `te` is the first point in time when the interval is not valid, i.e. the interval is *half-open*. The *relation* (R) between `A` and `B` is represented through the relation between these four points as given by the *definition* (δ_R). As an example depicted in Table 1 `A before B` means the interval `A` ends before the interval `B` begins. Similarly, `A during B` means `A` happens during `B`, because `A.ts` and `A.te` are between both points of `B`. We introduce the *TPStream* query language by formulating and explaining the query to detect aggressively driving cars from the introductory example in Listing 1.

The operator works on streams containing data referring to points in time. In the case of aggressive drivers, we work on a singular stream `CS`, providing sensor data from cars, which is specified as an input (`FROM`). This stream is partitioned by the `car_id` to evaluate each driver individually (`PARTITION BY`). The important aspect of deriving situations from the stream is handled in the `DEFINE` clause: The acceleration situation is represented with the symbol `A`, the condition `CS.accel` $>$ `8` $\mathrm{m/s^2}$ and the (optional) duration constraint `AT LEAST 5s`, while speeding and deceleration are defined by `B` and `C` respectively. The derived situations are analyzed with a `PATTERN`. For aggressive drivers, an acceleration (`A`) may `meet`, `overlap`, `start` or occur `during` a phase of speeding (`B`). These are alternatives in the pattern definition, separated with semicolons in the query language. The same applies for deceleration (`C`) and speeding (`B`). The pattern is fulfilled if at least one of *each* alternatives is true. We apply a window condition on the evaluation period (`WITHIN`), specifying that the pattern should only be searched within situations derived in the past `5` minutes. Finally, in case of a match, we `RETURN` aggregated results from each situation, in this case the `car_id` and the average `speed`.

3.1 Expressiveness

Most common CEP systems define patterns based on symbols connected via regular expressions. Specific extensions, like aggregations, put the expressiveness of those languages between regular and context-free grammars [27]. However, only *ISEQ* provides a *native* way to process patterns based on *temporal relations*. This deficit is also reflected in the respective languages.

By design, *TPStream* can express all *temporal relations* (and unlike *ISEQ* alternatives among them) in a *single* query. In contrast, a single pattern matching query in CEP systems is designed to detect a sequence, i.e., a `before` relation. Nevertheless, as shown by both straw man's approaches in our introduction, in a system supporting Kleene-closure it is possible to express other *temporal relations* through either multiple queries (decoupling derivation and detection) or a single query (without aggregation capabilities and the validation of duration constraints). Thus, our language does not express more than the *full* language of other systems.

Instead, we focus on enabling the user to express complex *temporal patterns* in a single, readable and maintainable query via the widely-known interval algebra (Table 1). For this purpose, we made two notable design choices that differ from some sequence-based approaches. First, some languages [27] allow to skip events while matching. In contrast, we derive the longest possible contiguous sequence of events, because this aligns well with the idea of long lasting situations and avoids ambiguity whether a situations is still ongoing during *other* events. Second, some languages [9] allow symbols to access aggregates of other symbols. Due to ambiguity in the expected results when dealing with situations, we do not allow this. For example, consider modifying the definition for symbol `B` in Listing 1 to `B AS CS.speed > max(A.speed)`. Then, for `A overlaps B` it is unclear whether `max(A.speed)` is accessed when `A` finishes, when `B` starts or is continuously monitored for each `B`. For a precise presentation of our approach, we chose those two concessions and will work on mitigating them in the future.

We would also like to sketch that, apart from those concessions, it is possible to express a purely sequence-based pattern with *TPStream*: A sequence can be expressed with a `before` relation and the implicit ongoing nature of situations can be eliminated with a duration constraint. Nevertheless, the basis for our implementation [19] features a standard sequence-based pattern matching operator that is optimized and thus preferable for this purpose. Similarly, our implementation can be easily integrated into other systems, because *TPStream* consumes and produces point-based event streams. In conclusion, this means that we do not change the expressiveness of other approaches, but by extending a query language with Allen's Interval Algebra, our benefits can be almost universally adopted.

4 ALGEBRA

The goal in designing *TPStream* is to develop an operator capable of continuously deriving situations from a stream of events and relate those situations to each other. To achieve this, we need to be able to express both the derivation and relation. For this purpose, we will formally model those aspects (streams, data, deriving situations and temporal pattern matching) in an algebra.

4.1 Stream Model

Definition 1 (Data Stream)**.** A data stream D is a potentially unbounded sequence of data items $\langle d_1, d_2, ...\rangle$ totally ordered by a relation $<_D$. $d_i \in D$ refers to the i-th data item in the stream according to that order and all data items are from the same domain $\mathcal{D}$. $\langle\rangle$ refers to an empty data stream.

In order to refer to multiple data streams, we will utilize the notation $D^1, D^2, D^3, \ldots$ with $D^i = \langle d_1^i, d_2^i, \ldots\rangle$, i.e. a superscript labels separate streams, while a subscript refers to the order within a stream. For the sake of simplicity and legibility, we will generally assume that each item in a data stream is unique and refer to previous work on the matter of handling potentially equal elements [8]. $\langle\rangle$ is mainly used to specify the case of *no output* in upcoming definitions.

Definition 2 (Continuous Subsequence)**.** Based on a data stream D, $D_{[i,j]} = \langle d_i, \ldots, d_j\rangle$ with $i < j$ refers to a continuous subsequence containing every data item as it pertains to $<_D$.

Definition 3 (Union)**.** The union $\uplus$ of two data streams D^1 and D^2 both totally ordered with $<_D$ results in a data stream D' with the same order $<_D$:

$$\uplus(D^1, D^2) := D' = \langle d_1', d_2', \ldots\rangle$$

such that D' contains each element from both D^1 and D^2. Analogous to set theory, the union of n data streams $D^1, \ldots D^n$ is abbreviated with the notation $\biguplus_{i=1}^{n} D^i$.

4.2 Data Model

Our operator involves two kinds of data which we need to define: *events* and *situations*. In general, events refer to a notification that something happened instantaneously at exactly one point in time while situations span multiple points in time and contain aggregated information for that time period.

Definition 4 (Event)**.** An event e is a pair (p, t) consisting of a payload p and an event timestamp t. p is from some domain $\mathcal{D}$ and t is from a discrete and totally ordered time domain $\mathcal{T}$. The validity of p is the instant t.

Definition 5 (Situation)**.** A situation s is a triple (p, ts, te) consisting of a payload p and two timestamps: ts (start timestamp) and te (end timestamp). p is from some domain $\mathcal{D}$. ts and te are from a discrete and totally ordered time domain $\mathcal{T}$ with $ts < te$. The half-open time interval $[ts, te)$ specifies the validity of p.

Event streams are ordered by the event timestamp and will be represented with E. **Situation streams** are ordered by the end timestamp of situations and will be represented with S. We focus our efforts on presenting algorithms for streams with data arriving in-order and leave the adjustment to out-of-order data by adapting previous research on out-of-order pattern matching [7, 21] for future work.

4.3 Derivation

Situations are derived from event streams through aggregation and predicate evaluation. We will first formally define aggregation on continuous event subsequences before deliberating on predicates and how to derive situation streams.

Definition 6 (Aggregated Event Subsequence)**.** An aggregate γ_{agg} is applied to an event stream subsequence $E_{[i,j]}$ by applying the aggregate agg to the events in the subsequence:

$$\gamma_{agg}(E_{[i,j]}) := (agg(e_i, \ldots, e_j), e_i.ts, e_{j+1}.ts)$$

When obvious from context, we abbreviate γ_{agg} with γ.

The result in Definition 6 technically already is a situation. However, for the derivation process as a whole, we want to discover situations for which a set of circumstances hold true. In order to provide an unambiguous process to identify these situations we are looking for the longest possible sequences for which these circumstances apply.

Definition 7 (Derived Situation)**.** Situations are derived with a function $derive_{\phi,\tau,\gamma}$ which aggregates information of a continuous event subsequence $E_{[i,j]}$ by applying γ iff the events in $E_{[i,j]}$ are the longest possible sequence of events to fulfill a given predicate ϕ and the covered timespan is within the given duration constraint $\tau := [d_{min}, d_{max}]$:

$$derive_{\phi,\gamma,\tau}(E_{[i,j]}) = \begin{cases} & \forall l \in [i,j] : \phi(e_l) \wedge \\ \gamma(E_{[i,j]}) & \text{if } \ !\phi(e_{i-1}) \wedge !\phi(e_{j+1}) \wedge \\ & (e_{j+1}.ts - e_i.ts) \in \tau \\ \langle\rangle & \text{otherwise} \end{cases}$$

Example. Assume the query in Listing 1 derives a speeding situation for a car with the time interval $[2, 10)$. This means `CS.speed` $\leqslant$ `70 mph` at $t = 1$ and $t = 10$ and in between those timestamps `CS.speed` $>$ `70 mph`. From an algebraic standpoint, assuming knowledge about the whole event stream, this aligns well with a natural interpretation: There are not multiple situations (e.g. $[2, 3), [2, 4), \ldots$) but rather one continuous speeding phase which fulfills the duration constraint ($d_{min} = 4s$ and $d_{max} = 30s$). For that reason and because it results in unique situations, we choose to derive the longest possible subsequence in Definition 7.

Definition 8 (Derived Situation Stream)**.** The $deriveStream_{\phi,\gamma,\tau}$ function derives a stream of situations from a given event stream E by applying the function $derive_{\phi,\gamma,\tau}$ to all possible subsequences and unifying the results:

$$deriveStream_{\phi,\gamma,\tau}(E) = \biguplus_{j} \biguplus_{i=1}^{j} derive_{\phi,\gamma,\tau}(E_{[i,j]})$$

Note that, due to assumption that each event in an event stream has a unique timestamp and the fact that $derive_{\phi,\gamma,\tau}$ derives the longest situations possible, it is easy to show that $deriveStream_{\phi,\gamma,\tau}$ produces a stream of situations with disjoint time intervals. This implies, that the order of situations using start timestamps is the same as the order using end timestamps, resulting in a beneficial pattern for query processing [16]. Due to space limitations, we omit a formal proof here.

4.4 Pattern Matching

TPStream matches multiple situation streams to a *temporal pattern* and produces a result event stream according to the given definitions. A *temporal pattern* is composed of *temporal constraints* between situation streams, which in turn comprise multiple *temporal relations* between exactly two streams. In this section, we present formal definitions of these terms, the output of a successful match and ultimately the *TPStream* operator.

Definition 9 (Temporal Relation)**.** Given two situation streams S^A, S^B, a *temporal relation* $R^{A,B}$, defines a valid relationship between two situations $s^A \in S^A$ and $s^B \in S^B$ according to Allen's Interval Algebra (cf. Table 1). s^A and s^B fulfill $R^{A,B}$, iff they satisfy the corresponding algebraic definition (δ_R).

Definition 10 (Temporal Constraint)**.** A *temporal constraint* $C^{A,B}$ between two situation streams S^A, S^B is a set of *temporal relations* $\{R_1^{A,B}, ..., R_m^{A,B}\}$. Two situations $s^A \in S^A$ and $s^B \in S^B$ fulfill $C^{A,B}$, iff they at least fulfill one of the *temporal relations*.

In other words, *temporal constraints* allow to specify multiple valid relations between two situation streams, providing the desired flexibility in expressing alternatives.

Definition 11 (Temporal Pattern)**.** For any number of situation streams $(S^1, ..., S^m)$, a *temporal pattern* $(\mathcal{P})$ is a set of *temporal constraints* $\{C^{i,j} | 1 \leqslant i < j \leqslant m\}$. A *temporal pattern* is matched by a *temporal configuration* $\overline{s} = (s^1 \in S^1, ..., s^m \in S^m)$, iff $\overline{s}$ satisfies every *temporal constraint*:

$$match_{\mathcal{P}}(\overline{s}) :\Leftrightarrow \forall C^{i,j} \in \mathcal{P} : \exists R^{i,j} \in C^{i,j} : \delta_{R^{i,j}}(s^i, s^j)$$

Example. Consider the example query of Listing 1 and let s^A be an acceleration situation as defined by `A` and s^B, s^C be a speeding (`B`) and deceleration (`C`) situation respectively. The `PATTERN` describes how pairs of situations in $\overline{s} = (s^A, s^B, s^C)$ can relate to each other via *temporal constraints*: For s^A and s^B the *temporal relation* can be either `A meets B`, `A overlaps B`, `A starts B` or `A during B`. It does not matter if acceleration overlaps speeding or if speeding contains acceleration. Both cases may lead to the result of detecting aggressive drivers. The *temporal pattern* on the other hand is a conjunction of *temporal constraints*: In order to match the pattern, each *temporal constraint* must be fulfilled.

Definition 12 (Pattern Matching Output)**.** A temporal pattern matching operator $PM_{w,\hat{\gamma}}$ matches a *temporal configuration* $\overline{s} = (s^1, s^2, \ldots, s^m)$ to a *temporal pattern* $\mathcal{P}$. It aggregates the information of $\overline{s}$ with some suitable aggregate $\hat{\gamma}$ and checks the *window* condition (cf. `WITHIN` clause):

$$window(\overline{s}, w) = w \leqslant \max_{s \in \overline{s}}(s.te) - \min_{s \in \overline{s}}(s.ts)$$

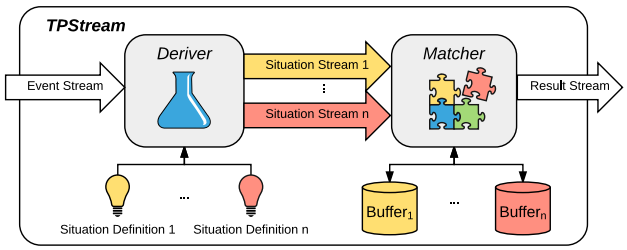

Figure 2: *TPStream* Architecture

The operator produces an output, if the *temporal configuration* matches the pattern during the specified window, i.e.:

$$PM_{w,\hat{\gamma}}(\overline{s}, \mathcal{P}) := \begin{cases} (\hat{\gamma}(\overline{s}), \max_{s \in \overline{s}}(s.te)) & \text{if } match_{\mathcal{P}}(\overline{s}) \wedge window(\overline{s}, w) \\ \langle\rangle & \text{otherwise} \end{cases}$$

Similarly to how we extended derived situations to derived situation streams (Definition 7 to 8), we can extend Definition 12 to situation streams:

Definition 13 (TPStream)**.** $TPStream_{w,\hat{\gamma}}$ matches multiple situation streams $S^1, \ldots, S^m$ to a *temporal pattern* $\mathcal{P}$ by applying the corresponding pattern matching operator $PM_{w,\hat{\gamma}}$ to the cross product of the situation streams and unifying the results:

$$TPStream_{w,\hat{\gamma}}(S^1, \ldots, S^m, \mathcal{P}) := \biguplus_{\overline{s} \in \times_{i=1}^{m} S^i} PM_{w,\hat{\gamma}}(\overline{s}, \mathcal{P})$$

Note that $TPStream_{w,\hat{\gamma}}$ results in an event stream and can thus easily be integrated into common CEP processing pipelines.

5 ALGORITHMS & IMPLEMENTATION

In this section, we present our algorithms and implementation details for detecting *temporal patterns* among streams of point events. Following the definitions from the previous section, the general architecture consists of two main components, as depicted in Figure 2. The *deriver-component* consumes events from the input stream and derives the defined situation streams. Then, those streams are passed to the *matcher-component*, which performs the actual pattern matching. In the following two subsections we will explain both components in detail. For the sake of simplicity we defer low latency detection to section 5.3 and initially wait for the end timestamp of derived situations before invoking the *matcher*. The last part of this section describes how *TPStream* computes efficient execution plans and dynamically adapts to changing workloads.

5.1 Deriving Situations

Definition 8 introduced derived situation streams, using knowledge about the whole input-stream. To compute situation streams incrementally as new events arrive the *deriver-component* manages a buffer for ongoing situations (B) and the situation stream definitions (D). Algorithm 1 shows how they are used to derive situations on-the-fly. For each defined situation, 3 cases are checked: If there is no started situation on the buffer, but the predicate holds true, a new situation is started. Therefore, we compute initial values for all defined aggregates (e.g., `p.speed` for an `max(speed)` aggregate). Those values are bundled with the event's timestamp and stored on the buffer (Lines 4,5). The

Algorithm 1: DeriveSituations

Input: (p, t): event
Data: $B := [(p', ts)_i]$: active situation buffer,
$D := [(\phi, \gamma, \tau)_i]$: situation definitions

```
1  R ← ∅;
2  foreach i ∈ |D| do
3      (φ, γ, τ) ← D[i];
4      if B[i] = ∅ ∧ φ(p) then
5          B[i] ← (initAgg(p, γ), t);
6      else if φ(p) then
7          updateAgg(p, B[i], γ);
8      else if B[i] ≠ ∅ then
9          if (t − B[i].ts) ∈ τ then
10             R ← R ∪ {(B[i].p', B[i].ts, t)};
11         B[i] ← ∅;
12 if R ≠ ∅ then
13     updateMatcher(R, t);
```

Algorithm 2: UpdateMatcher

Input: S: set of finished situations, t: the current time

```
1  purgeBuffers(t);
2  foreach s ∈ S do
3      addToBuffer(s);
4      performMatch({s}, 0);
```

temporal validity of a started situation is prolonged, if the current event fulfills the predicate. In this case, the buffered aggregates are updated using the event's payload (p) (Lines 6,7). Finally, a situation is finished on the first event not satisfying the defined predicate. In this case, the situation's end timestamp is fixed to the current time, it is added to the result-set R (provided it satisfies the duration constraint τ) and the corresponding buffer slot is cleared (Lines 8-11). After updating the state of each situation stream, the result-set is passed to the *matcher-component* (Lines 12,13).

5.2 Matching the Pattern

The *matcher* implements an incremental version of Definition 13 ($TPStream_{w,\hat{\gamma}}$). In other words, it detects matches on-the-fly as new situations are handed over from the *deriver-component*. The general idea is to employ a buffer for each situation stream and perform the pattern detection via a multi-way join between those buffers, using the *temporal constraints* as join-conditions. Recap that all situations within a stream are disjoint and thus imply the same order on both the start and end timestamps (Definition 8). We will use this fact to ensure efficient execution of the matcher component.

Each time the *deriver* distills new situations, Algorithm 2 is invoked: At first, expired situations are purged from the buffers (Line 1). That is, removing all situations `s` with `s.ts` $<$ `t`$-$`window`. Because of the mentioned ordering, this effectively means, finding the first situation `s'` with `s'.ts` $\geqslant$ `t` $-$ `window` and discarding all previous events. The buffers are implemented via array-backed ring buffers, which efficiently support these operations.

After purging outdated situations, each new situation is first added to its corresponding buffer, before the actual matching

Algorithm 3: PerformMatch

Input: *ws*: working-set, *sc*: current step count
Data: order: evaluation order

```
1  if sc = order.getNumSteps() then
2      publishResult(ws);
3      return;
4  step ← order.getStep(sc);
5  if step.isSet(ws) ∧ step.checkConstraints(ws) then
6      performMatch(ws,sc + 1);
7  else if !step.isSet(ws) then
8      foreach (p, t_s, t_e) ∈ findMatches(step, ws) do
9          ws ← ws ∪ {(p, t_s, t_e)};
10         performMatch(ws, sc + 1);
11         ws ← ws\{(p, t_s, t_e)};
```

algorithm (Algorithm 3) is invoked (Lines 2-4). We force the new situation to be part of any successful match, by passing it as a parameter. This ensures the desired incremental creation of results, because we pass a new, not yet considered situation on every invocation.

The matching algorithm relies on a so called *evaluation order*, which we describe briefly upfront. An *evaluation order* determines the order in which situation buffers are joined and provides the required information for each processing step (a reference to the situation buffer and the set of *temporal constraints* to be fulfilled). Using this information and a partial *temporal configuration* (*working-set*), Algorithm 3 matches the *temporal pattern* as follows: In each step, the corresponding situation buffer is searched for situations satisfying all applicable *temporal constraints* (Line 8). Applicable means, that the counterpart of the constraint is already present in the *working-set*. Then, all returned situations are successively added to the *working-set* and for each of the new partial *temporal configurations*, the algorithm proceeds to the next step (Lines 9-11). Lines 5 and 6 intercept the evaluation, if the *working-set* already contains a situation for the current step, which accounts for situations passed as a parameter from Algorithm 2. In this case, the corresponding buffer is ignored and the step's *temporal constraints* are checked directly. Finally, a match is detected if the *working-set* contains a situation from every buffer (Lines 1-3). The *publishResult* function consumes this *working-set*, assembles the result and pushes it into the output stream.

Obviously, the evaluation performance of Algorithm 3 mainly depends on the efficiency of the *findMatches* function. A naïve approach would be, to scan the entire buffer and check the *temporal constraints* for each situation separately. With R_i denoting the i-th intermediate result, B_i the buffer traversed in step i and $|R_1| = |B_1|$, the costs (C) of *performMatch* following this approach can be estimated with:

$$C = |R_n| + \sum_{i=1}^{n-1} |R_i| \cdot |B_{i+1}| \tag{1}$$

To speed up the computation, we again use the order of situation streams: Because the order is reflected on the buffers, we are able to find all matching situations using binary searches.

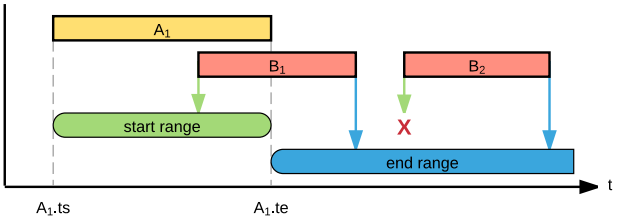

Figure 3: Temporal Matching via Range Queries

Relation (R)	Definition (δ_R)	t_d(R)	Prefix-Group (G)	t_d(G)
A before B	A.ts < A.te < B.ts < B.te	B.ts	A.ts < A.te ≤ B.ts	B.ts
A meets B	A.ts < A.te = B.ts < B.te	B.ts		
A starts B	A.ts = B.ts < A.te < B.te	A.te	A.ts = B.ts	B.ts
A equal B	A.ts = B.ts < A.te = B.te	A.te = B.te		
A started-by B	A.ts = B.ts < B.te < A.te	B.te		
A overlaps B	A.ts < B.ts < A.te < B.te	A.te	A.ts < B.ts	B.ts
A finishes B	A.ts < B.ts < A.te = B.te	A.te = B.te		
A contains B	A.ts < B.ts < B.te < A.te	B.te		

Table 2: Low-Latency Analysis

We first discuss how this is done for a single *temporal relation* before extending it to (multiple) *temporal constraints*. Recall that a *temporal relation* explicitly defines a relationship between all four endpoints of two situations. For instance, this is `A.ts` $<$ `B.ts` $<$ `A.te` $<$ `B.te` for `A overlaps B`. Now, given an instance of situation `A`, we can obtain matching instances of `B` by (i) issuing two range-queries on the buffer of `B`, using the timestamps of `A` as boundaries and (ii) intersecting the results of those queries. For the example relation, these queries are:

(1) `A.ts` $<$ `ts` $<$ `A.te` for the start-timestamp and
(2) `A.te` $<$ `te` $<\infty$ for the end timestamp.

It is easy to see, that each situation falling into both ranges fulfills the given *temporal relation*. Figure 3 illustrates this using 3 situations: Situation A_1 in combination with the *temporal relation* is used to build the two search ranges. After intersecting the results ($\{B_1\}$ for the start range and $\{B_1, B_2\}$ for the end range), we receive our final result B_1. Note that for *temporal relations* allowing more than one result (e.g., `A before B`), this strategy additionally eliminates the need for checking each combination individually.

Typically, a *temporal constraint* contains more than one *temporal relation*, stating each of them as a valid relationship between two situations. This can be easily integrated by executing the search separately for each of the defined relations and subsequently building the union of the obtained results. The conjunction of multiple *temporal constraints* can be implemented as an intersection of the results from the respective individual queries. Because the buffers are backed by a contiguous array, we can represent the search results as index-ranges and thus efficiently compute the required unifications and intersections. This approach reduces the estimated costs of *performMatch* to:

$$C = \sum_{i=2}^{n} \left(|R_{i-1}| \cdot |R_i| + C_{findMatches}(|B_i|) \right) \qquad (2)$$

with $C_{findMatches}(|B_i|)$ being bounded by $|\mathcal{P}| \cdot 13 \cdot 4 \cdot \log_2(|B_i|)$. The constant factors 13 and 4 arise from the possible *temporal relations* per constraint and the binary searches to execute for each of them, respectively.

5.3 Low-Latency Matching

In this section, we will determine the earliest points in time (t_d) to detect a *temporal relation* ($t_d(R)$), *temporal constraint* ($t_d(C)$) and *temporal pattern* ($t_d(\mathcal{P})$). Then, we adjust our algorithms from the previous section to deliver matches as early as possible.

5.3.1 Analysis. Two situations `A,B` can only be related once we know they exist, making $\max(\texttt{A.ts}, \texttt{B.ts}) \leqslant t_d(R)$ a trivial lower bound for *all* relations. For exact $t_d(R)$ consider a relation's definition δ_R depicted in Table 2. Let $\texttt{t}_1 \leqslant \texttt{t}_2 \leqslant \texttt{t}_3 \leqslant \texttt{t}_4$ be the timestamps in the order they appear in δ_R. It is easy to see that the ordering of $\texttt{t}_4$ can implicitly be derived at $\texttt{t}_3$, because $\texttt{t}_3 \leqslant \texttt{t}_4$ and there are no timestamps beyond that. Furthermore, at $\texttt{t}_1$ and $\texttt{t}_2$ there are other relations sharing the same definitions up to that point, i.e., it is not possible to distinguish them from each other. To show this, we have grouped relations starting with `A.ts` as *prefix groups* in Table 2 (`B.ts` groups are analogous). For those reasons we can conclude $t_d(R) = \texttt{t}_3$.

A *temporal constraint* $C = (R_1, \ldots, R_n)$ for `A,B` matches if at least one relation matches. Therefore, the earliest detection time is $t_d(C) = \{t_d(R_1), \ldots, t_d(R_n)\}$. Note that $t_d(C)$ is a set and the actual detection time of two situations depends on the fulfilled relation. Further, if C contains *all* relations of a *prefix group* (cf. Table 2), the detection time of these relations is shifted to the trivial lower bound ($t_d(G)$).

Finally, for a pattern $\mathcal{P} = (C_1, ..., C_m)$, each constraint must be matched. However, a single *temporal configuration* matching $\mathcal{P}$ fulfills exactly one *temporal relation* ($\overline{R}$) from each constraint, making $t_d(\mathcal{P}) = max(t_d(\overline{R} \in C_1), \ldots, t_d(\overline{R} \in C_m))$. In general, $t_d(\mathcal{P})$ is among the constraint detection points: $t_d(\mathcal{P}) \subseteq \bigcup_{i=1\ldots m} t_d(C_i)$.

5.3.2 Implementation. For the ease of presentation, we ignore the optional duration constraints on situations as well as *prefix groups* during the development and discuss the required changes at the end of this section. We gained two implementation-relevant insights from the low-latency analysis. First, new matches can only be detected if a new situation starts or a situation ends. Second, only a subset of the defined situations can possibly produce a match at $t_d(\mathcal{P})$. Thus, the matching process can be delayed until a situation with at least one endpoint in $t_d(\mathcal{P})$ occurs without affecting the latency. We call those situations *trigger situations* since only they should *trigger* a *performMatch* call. These insights affect our algorithms in the following ways. Situations must be available for matching from their start on, which can easily be achieved by adjusting the *deriver*. Additionally, we need to determine for each situation stream if the derived situations are *trigger* situations. For *trigger* situations we need to identify the point in time to execute *performMatch* (at its start, end or both).

However, the following cases must be considered during the matching process. If a situation requires matching on both endpoints, care must be taken not to produce duplicate results. Further, started situations must not be visible to the matcher in all cases: If two situations are related via `finishes` or `equals`, they could be mistakenly matched, because their temporary end timestamps (i.e., the current time) are equal. On the other hand, if two situations are not explicitly related via a *temporal constraint* they may participate in a successful match, even if both end timestamps are unknown. To illustrate this, consider the following pattern on four situations (`A, B, C, D`): `A before B AND A before C AND A before D AND (D during C OR C`

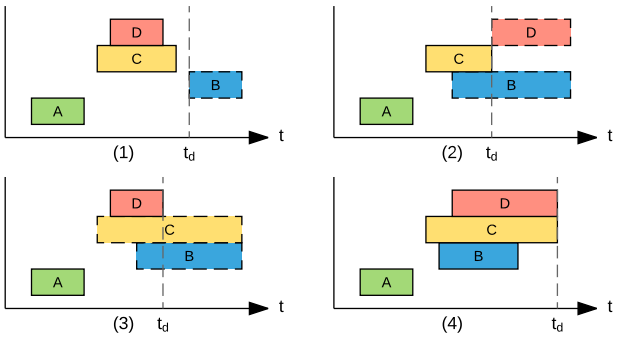

Figure 4: Earliest detection time (t_d) of different *temporal configurations* for the same pattern

`finishes D OR C meets D`). It defines situation `A` as starting point of every match. `B` is not explicitly related to `C` and `D` and required to happen after `A`. Consequently, `B` is a *trigger* situation and `B.ts` is in $t_d(\mathcal{P})$. For `D` both `D.ts` (via `meets`) and `D.te` (via `during`, `finishes`) are in $t_d(\mathcal{P})$. Figure 4 shows four representative *temporal configurations* for this pattern, highlighting the earliest point of detection (t_d). Configuration 1 showcases `B` as a *trigger* situation with t_d = `B.ts`. For the second configuration, `D` is the *trigger* with t_d = `D.ts`. This case also shows that two started relations may participate in a match if their end is unknown (`B,D`). The remaining configurations highlight `D` as a *trigger*, but with t_d = `D.te`.

Instead of handling these cases explicitly, our low latency algorithm avoids them by ensuring a unique combination of situations in the *working-set*, before passing it to the matching algorithm. In particular, this means started situations are managed in a separate buffer, inaccessible for the matching algorithm, and all valid combinations among them (i.e., all combinations of started situations, not explicitly related to the current one) are built upfront inside the *working-set*. Furthermore, to avoid duplicate results, the following fact is exploited: *temporal relations* enforcing matching on a situation's start require its counterpart to be finished in the past. On the other hand, *temporal relations* triggering matching on a situation's end require its counterpart to be either started (and not yet finished) or finished at the same time (cf. Table 2). Consequently, manually adding the started counterpart to the *working-set*, before executing the matching algorithm on a situation's end ensures uniqueness of the produced results.

The details are presented in Algorithm 4. After purging outdated situations from the buffers (Line 1), each started situation (s) is added to the additional buffer and if $s.ts \in t_d(\mathcal{P})$, the algorithm performs a regular match with s being the only constant in the *working-set* (Lines 2-5). Furthermore, if there are *started and unrelated* situations, we perform matches with s and each combination of them (Lines 6-8). This accounts for configurations as seen in Figure 4.2. All finished situations are migrated from the separate to the regular buffer (Lines 9-11) and if $s.te \in t_d(\mathcal{P})$, the matching process is triggered. This time with combinations of s and all *started and related* situations (Lines 15-16), further combined with all *started and unrelated* situations (Lines 17-18), which fuses the avoidance of duplicate results and false positives. An example for this case is shown in Figure 4.3. Note that, the actual constraint-checking among the created combinations is performed by the call to *performMatch* (Algorithm 3), since it is aware of pre-set situations in the *working-set*. As we will show

Algorithm 4: Low-Latency MatcherUpdate

Input: S_f, S_s: sets of finished/started, t: current time

```
1  purgeBuffers(t);
2  foreach s ∈ S_s do
3      startedBuffer.add(s);
4      if matchOnStart(s) then
5          performMatch( {s}, 0 );
6          U ← getUnrelatedStarted(s);
7          foreach u ∈ powerset(U)\∅ do
8              performMatch( u ∪ {s}, 0 );
9  foreach s ∈ S_f do
10     startedBuffer.remove(s);
11     addToBuffer(s);
12     if matchOnEnd(s) then
13         R ← getRelatedStarted(s);
14         U ← getUnrelatedStarted(s);
15         foreach r ∈ powerset(R)\∅ do
16             performMatch( r ∪ {s}, 0 );
17             foreach u ∈ powerset(U)\∅ do
18                 performMatch( r ∪ u ∪ {s}, 0 );
```

in section 6, the extensive building of combinations has only minimal impact on the runtime-performance, because it shifts load from joining to the update algorithm and does not introduce additional computation steps.

Duration constraints on situations are incorporated into low latency-matching with only a few modifications: First, if a maximum duration constraint is defined (regardless of a possibly specified minimum duration), the corresponding situation must not be included in the matching process until its end is known – and the constraint is fulfilled. Hence, these situations are excluded from the set of started situations (S_s) and if their start timestamp is in $t_d(\mathcal{P})$, matching is deferred to their end timestamp. Second, if a minimum but no maximum duration is defined, the inclusion into the set of started situations is deferred until the constraint is satisfied. This possibly implies the inclusion of its deferred start timestamp ($\overline{\texttt{ts}}$) into $t_d(\mathcal{P})$. As an example, consider the pattern `A during B` and the following order of timestamps: `B.ts` $<$ `A.ts` $<$ `A.te` $<$ `B.`$\overline{\texttt{ts}}$ $<$ `B.te`. This match can not be detected at `A.te`, because `B`'s duration does not exceed the lower bound at this point. Hence `B.`$\overline{\texttt{ts}}$ requires a matcher invocation.

To handle *prefix groups* the restriction that two *started and explicitly related* situations must not be matched is relaxed. That is, matching is performed if the corresponding *temporal constraint* contains one or more *prefix groups*. However, for still being able to omit false positives, the matcher must distinguish between *prefix group* and regular detection. Technically this means splitting the *temporal constraint* into two disjoint sets (one containing all *temporal relations* forming a *prefix group* and another one holding the remaining relations) and use the first set, when matching on a situation's start and the second one on its end.

5.4 Computing the Evaluation Order

The *matcher component* maps the problem of temporal pattern matching to a multi-way join between situation buffers. Like multi-join processing in traditional relational database systems,

Relation	before	during	overlaps	starts, finishes, meets	equal
Selectivity	*0.445*	*0.03*	*0.01*	*0.0049*	*0.0006*

Table 3: Initial estimates for the selectivity of *temporal relations*. Mirror relations are equivalent.

the performance of joining heavily depends on the order in which the join operations are executed. In this section, we discuss how the *matcher's* evaluation order is computed and present the cost-model used during this process.

Analogous to classical join processing we implemented an optimizer that enumerates possible execution plans, computes the expected computational costs for each of them and suggests the most efficient plan for execution. We do not provide multiple implementations of the join operator, so that enumerating possible plans reduces to the enumeration of possible evaluation orders. To further reduce the number of plans to consider, we exclude orderings joining a situation buffer without an applicable *temporal constraint*. In other words: Plans involving the calculation of a cross product are omitted.

According to Equation 2, estimating the costs for a given plan boils down to estimating the size of intermediate results:

$$|R_i| := \begin{cases} |B_1| & \text{if } i = 1 \\ |R_{i-1}| \cdot |B_i| \cdot s_i & \text{otherwise} \end{cases} \quad (3)$$

s_i denotes the selectivity of the applicable *temporal constraints* in step i (C_i), which can be composed from the selectivities of the contained *temporal relations* as follows:

$$s_i := \prod_{C \in C_i} \left(\sum_{R \in C} s_R \right) \quad (4)$$

When a query is initially deployed into the system, the situation buffers are empty and we have no estimation on the selectivity of the *temporal constraints*. Hence, we initially assume the selectivities depicted in Table 3. These values are backed by the following back-of-the-envelope calculation: The combined selectivity of all possible relations should be 100%. Assuming equal sized buffers and an equal temporal distribution of the situations, the selectivity of a `before` relation will be around 50%. For `during`, the number of results is limited by the maximum of both buffer sizes, because a situation `A` can happen during at most one other situation (`B`), but `B` may contain multiple `A` situations. All other *temporal relations* define a 1:1 relationship, which limits the worst case to the minimum of both buffer sizes. As seen in Table 3, we additionally separate the last case by the number of stated equalities. Note that even though this is an initial estimate, the resulting plans prove to work well in most cases (cf. section 6.4.2).

5.4.1 Adaptivity. Once a query is deployed in a CEP-system, it is typically active for a long time. Hence, more important than the quality of an initial execution plan is the ability to tune this plan and adapt it to changing workloads. To do so, we keep track of the buffer sizes and selectivities imposed by *temporal constraints* during execution. The buffer-sizes are available at any point in time and at no cost, since they are tracked by the underlying data structure. However, to smooth out (potential) spikes, we monitor the buffer size using an *exponential moving average*, which is adjusted after each call to the *matcher's* update method as follows:

$$EMA_i = \alpha * |B_i| + (1 - \alpha) * EMA_{i-1}$$

EMA_i holds after the i-th update. $|B_i|$ denotes the size of the considered buffer at update i and the *smoothing factor* $\alpha \in (0, 1)$ determines how much weight is given to previous values. For example, a value close to 1 assigns almost no weight to older values, while a value close to 0 decreases the influence of new values. The selectivities of the *temporal constraints* are also managed with EMAs using one EMA-value per constraint.

To check if a re-computation of the evaluation order is required, the active plan stores a snapshot of the statistics it is based on. After each update, we compare them to the current values and if any of them differs by more than the defined threshold (t), we trigger a re-computation.

Finally, if a migration is required, we are able to migrate to the new plan between any two invocations of the matcher component. Because the matcher does not store any intermediate results, but solely relies on the situation buffers this switch comes without any additional migration costs. As we will show in section 6.4.2, the total costs for adaptivity are negligible.

6 EXPERIMENTAL EVALUATION

In this section we present the results from our experimental evaluation of *TPStream*[2]. First, we study *TPStream*'s evaluation performance in comparison to *ISEQ* and point based CEP systems. Then, we analyze the latency improvement of our approach in comparison to *ISEQ*. Finally, we prove the validity of our optimization techniques.

6.1 Setup

All experiments were conducted on a workstation equipped with an Intel i7-2600 3.4 GHz processor and 8GB of memory, running a Debian Linux (kernel version 4.11.11-1). The results presented for each experiment are averaged values from a total of 10 runs, whereby every run was preceded by a warm-up phase of evaluating at least 100,000 events before the measurement was started.

The main goal of this section is to compare *TPStream*'s processing performance and our low latency approach to the state-of-the art solution for temporal pattern matching (*ISEQ*). There is no publicly available implementation of *ISEQ*, so we implemented it based on the available description in [20]. As required by the design of *ISEQ*, the input consists of interval streams ordered by endpoint. These streams are again generated with our *deriver* component.

In order to provide a comparison with point based systems, we also included CEP-solutions from the open-source community (Esper[3] 6.0.1) and academia (SASE+ [4]), when applicable. While Esper is a production ready CEP system, highly optimized for efficient query execution, SASE+ is one of the most popular CEP languages in the research community and served as foundation for the *ISEQ* operator. The rich query language of Esper allowed us to express both straw man's approaches as sketched in the introduction. We refer to the first approach (2 phase pattern matching) with *Esper-1* and the low latency approach is denoted as *Esper-2*. Because the SASE+ implementation does not feature chaining of queries, we only implemented the low-latency approach. *TPStream* and all its competitors are implemented in the JAVA programming language, whereby *TPStream* and *ISEQ* are based on JEPC [19] – an event processing middleware. We used

[2]Datasets and source code available at http://uni-marburg.de/oaCPk
[3]http://www.espertech.com
[4]https://github.com/haopeng/sase

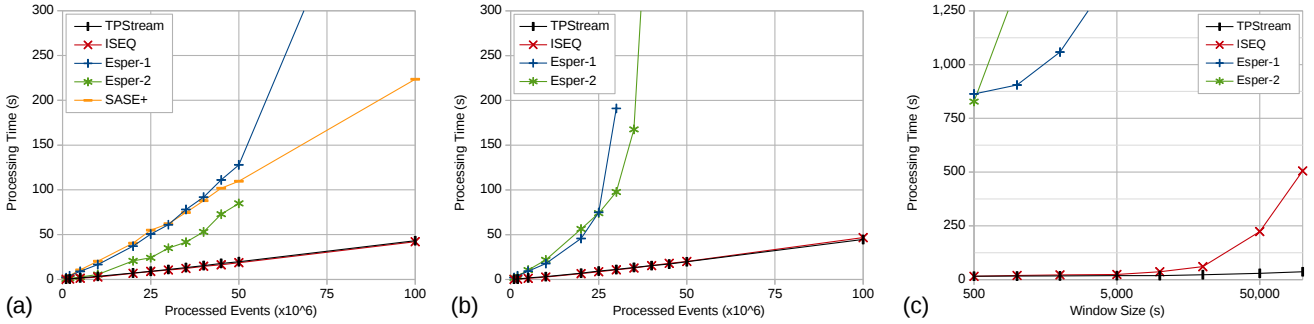

Figure 5: Processing time for aggressive driver detection as a function of the input size: (a) simplified pattern, (b) full pattern and processing time for disconnected pattern detection as a function of the window size (c)

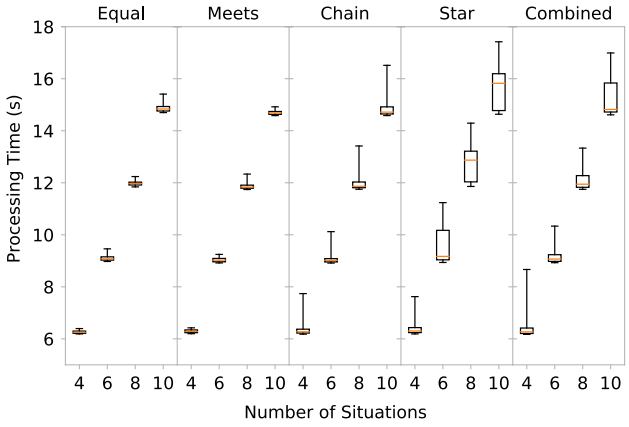

Figure 6: Processing time for various query patterns

Oracle JDK 1.8.0.144 compile the systems and ran all experiments on that JVM with 6GB of heap space.

During the evaluation two data sources were used. The first source comprises trip data generated with the Linear Road Benchmark [4]. Besides other attributes, each event consists of a unique car id, its location, the current speed and acceleration. We generated data simulating 5 hours of traffic on a single expressway with 1000 active cars per hour. Each active car reports its state every second, leading to 887 million events (36 GB of data). The second source is a random event generator, tuned to pose high load on the system. It generates event streams with a configurable number of boolean attributes, each representing a single situation stream. The generated situations last between 10 and 100 seconds, while the gaps between two consecutive situations span 10 to 50 seconds (both uniformly distributed). Events are generated with a frequency of 1Hz, so that for a situation lasting n seconds, the corresponding attribute's value is `true` for exactly n consecutive events.

Independent of the data-source, we used a single thread for both, reading/generating the data and evaluating the query. For each experiment, we measured the reading/generation time upfront and removed it from the presented results. The most important parameters throughout all experiments are as follows:

Event Rate The rate (events/s) with which events are pushed into the systems.

Window Size The size of the time window (s) during which a pattern must occur completely.

Event Count The total number of events to process.

6.2 Processing Time

This set of experiments compares the processing performance of *TPStream* with its competitors using various queries and parameter settings. The events were pushed into the system at the maximum possible rate and we used the processing time as main measure.

6.2.1 Aggressive Drivers. We injected different fractions (1M to 100M events) of the Linear Road dataset into the system and executed the example query of Listing 1 (without duration constraints). The thresholds for speeding, acceleration and deceleration were the 99th, 90th and 90th percentiles for the speed and positive/negative acceleration values of a 50M event sample. Besides chaining of queries, the SASE+ implementation also lacks support for disjunctions. Nevertheless, to include SASE+ in this experiment, we also evaluated a simplified query version which restricts the used *temporal relations* to `meets` and `overlaps`.

The results of this experiment are shown in Figure 5 (a – simplified pattern, b – full pattern). The x-axis shows the number of processed events, the processing time is shown on the y-axis. *TPStream* and *ISEQ* are head to head and their processing times increase linearly with the number of processed events. Further, they are insensitive to alternatives, resulting in almost identical processing times for both query variants. *TPStream* was not able to outperform *ISEQ* in this experiment, because in the given pattern all situations overlap which in turn allows to break the buffer scan early. Esper benefits from the simplified version of the pattern, but its evaluation performance is inferior to *TPStream* and *ISEQ* (up to 30x for the full query and 15x for the simplified version). When evaluating the full query, Esper hit the memory limit of 6GB and the system crashed if more than 30M (Esper-1) and 40M (Esper-2) events were processed. For the simplified version, the processing time of Esper-1 increases drastically when processing more than 50M events – Esper-2 runs out of memory and crashes. SASE+ managed the evaluation, but was clearly outperformed by *TPStream* and *ISEQ*.

6.2.2 Disconnected Pattern. The second experiment compares processing time and memory consumption of the systems using a pattern with high selectivity: `A before B overlaps C`. The difference to the first experiment is, that each `A` situation may be related to many `B overlaps C` sub-matches instead of contributing to at most one match. Hence, we expected the number of results and consequently the processing time/memory consumption to depend on the size of the configured time window. We injected 300M synthetic events into the systems and executed

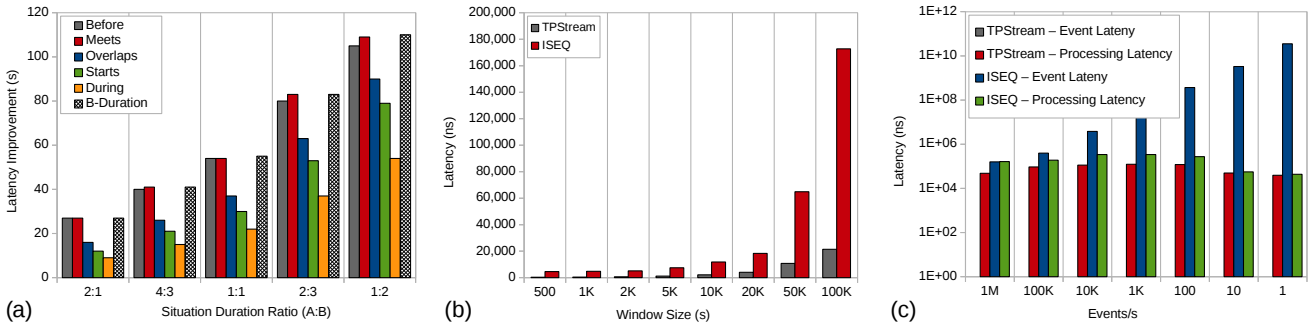

Figure 7: (a) application time latency gain per *temporal relation*and comparison of result latency (b) under maximum possible throughput as a function of the window size, (c) under varying event rates with a fixed size window

the query with window sizes varying from 500s (8:20 minutes) to 100,000s (slightly more than one day).

Figure 5 (c) shows the processing time of all systems as a function of the window size (note the log-scale). In this experiment, *TPStream* is able to outperform *ISEQ* by a factor of 14 using a window of 100,000s. This is because *ISEQ* does not make use of the order on the situations' start timestamp and requires additional computational steps during result construction and buffer pruning. SASE+ did not finish this experiment in a reasonable time for none of the window sizes and Esper barely managed window sizes up to 20,000s. To measure the average memory consumption, we monitored the used heap space with a frequency of 20Hz during each run and averaged these values. Both, *TPStream* and *ISEQ* require only very little additional memory for increased window sizes: *TPStream* 911 - 1018 MB, *ISEQ* 903 - 1027 MB. Esper stays stable at 1 GB up to a window size of 10,000s but afterwards suffers from buffering single events rather than a compact representation like situations. For the last evaluable query (20,000s window) Esper already consumed 1,7 GB of memory.

6.2.3 Query Patterns. To give a comprehensive overview of *TPStream*'s processing performance, we evaluated 5 different query patterns and varied the number of situation streams from 4 to 10. Queries 1-3 (**Equal, Meets, Chain**) are of the form $S_1 \oplus_1 S_2 \oplus_2 \ldots \oplus_{n-1} S_n$, with $\oplus_i$ = `equals`, $\oplus_i$ = `meets` and $\oplus_i$ a randomly drawn *temporal relation*, respectively. In query 4 (**Star**), S_1 is connected with every other situation, via a random *temporal relation* (i.e. $S_1 \oplus_1 S_2, S_1 \oplus_2 S_3, \ldots, S_1 \oplus_{n-1} S_n$). Query 5 (**Combined**) combines the two generic patterns by connecting the first $n/2$ situations via the **Chain** pattern and the remaining situations according to the **Star** pattern. Each query-type was executed 100 times, using 50M synthetic events and a window size of 2,000s.

The box plots in Figure 6 provide the median as well as the 25th and 75th percentiles of the processing time. For all query types, the median processing time increases linearly with the number of situations. The generic **Chain** pattern incurs higher maximum values than **Equal** and **Meets**, because the possible *temporal relations* include `before`, which is highly selective. This forces the matcher to build many partial results – especially if three or more consecutive situations are in a `before` relationship. **Star** queries are more sensitive to the concrete pattern instance, because in the worst case every situation triggers the matching process. This effect can also be observed for the **Combined** pattern, but to a smaller degree, because only half of the situations are connected via a **Star** pattern.

6.3 Low Latency

This set of experiments compares the result latency of our approach with the state-of-the art solution for temporal pattern matching, *ISEQ*.

6.3.1 Application Time. At first, we measure the latency improvement of *TPStream* compared to *ISEQ* in terms of application time. That is, we compare the timestamps of the events that produced a result in both approaches and calculate their difference. We evaluated each *temporal relation* independently using two synthetic situation streams (`A`,`B`). We varied the average duration ratio from 2:1 to 1:2, keeping `A`'s average duration fixed at 55 seconds. Note that the window size has no impact here (as long as it is not too small to hold a match), so it was set to 1,000s.

Figure 7 (a) shows the average latency improvements per *temporal relation*. For sequential relations (`before, meets`), the gain in latency is equal to the average duration of `B` situations, because matches are detected at `B.ts`. For the remaining relations, the detection time is `A.te` and the average improvement depends on the concrete *temporal relation*. In the worst case (`during`) this is `B.duration`/2. Note that, `equals` and `finishes` were not included, because no latency improvements can be achieved.

6.3.2 Wall Clock Latency. We conducted two experiments, showing that *TPStream*'s processing techniques significantly reduce the result latency in terms of wall clock time which is a critical aspect in a streaming scenario. Therefore, we repeat the experiment from section 6.2.2 twice: first, we measure the time passed between the arrival of the first event that could produce a result and the receipt of that result. We varied the window size and pushed events with the maximum possible rate. For the second experiment we fixed the window size at 100,000s and varied the event rate from 1M to 1 events/s. This time, we split the measured latency in (i) processing latency: the time passed between arrival of the event that triggered the result and the actual receipt of that result and (ii) event latency: the time passed between arrival of the first event that *could* trigger the result and the arrival of the event that *actually* triggered that result.

The results are shown in Figure 7 (b,c). Both figures show the average latency per result (y-axis, note the log-scale for c). While (b) shows, that *TPStream*'s evaluation techniques provide latency savings through reduced processing time, (c) highlights the savings achieved with our low-latency matcher. Especially when the rate is in sync with application time (1 event/s), the event latency of *ISEQ* dominates the processing latency and almost reaches the application time savings (~35s, cf. Figure 7 a, 1:1, `overlaps`), while *TPStream* introduces no event latency at all.

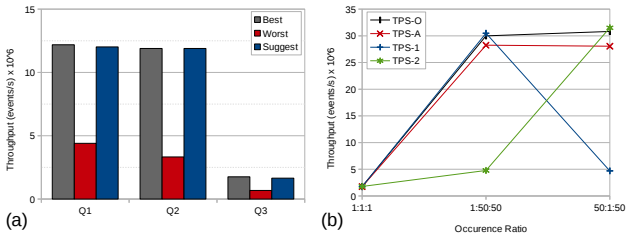

Figure 8: (a) Quality of the initial plans for Q1 – Q3, (b) Throughput comparison: dynamic plan adaption vs. best initial plans

6.4 Plan Quality & Adaption

Finally, we evaluate the optimization techniques presented in section 5.4. Like in section 6.2, events were pushed with the maximum possible rate.

6.4.1 Initial Plan Quality. To evaluate the quality of the generated initial plans, we used the following queries on three situation streams: **Q1**: `A overlaps B AND A overlaps C AND B starts C`, **Q2**: `A overlaps B AND A before C AND B overlaps C` and **Q3**: `A before B AND A before C AND B before C`. For each query, we generated all 6 valid plans and measured the throughput (processed events/s) by evaluating synthetic events with a window size of 5,000s.

Figure 8 (a) shows the results for the best, worst and suggested plans and clearly confirms our approach. For queries **Q1** and **Q2** the best plan was suggested. The initial plan for **Q3** was `C` → `B` → `A` even though the estimated costs for `C` → `A` → `B` are the same. The experiments show, that `C` → `A` → `B` would have been a slightly better choice, but the difference is negligible.

6.4.2 Dynamic Plan Adaption. To analyze the plan adaption capabilities of *TPStream*, we executed **Q3** again and processed 300M events. The occurrence ratio of situations `A`, `B` and `C` changed from 1:1:1 to 1:50:50 after 100M events and finally to 50:1:50 after 200M events. The window size, smoothing-factor (α) and threshold for plan migration (t) were set to 10,000s, 0.01 and 0.2, respectively. Besides the adaptive implementation (TPS-A), we ran the experiment with both best initial plans `C` → `B` → `A` (TPS-1), `C` → `A` → `B` (TPS-2), and an implementation, doing a hard coded switch to the best plan exactly when the characteristics of the stream changes (TPS-O).

Figure 8 (b) shows the throughput for all four configurations and the three different stream-characteristics: TPS-1 and TPS-2 both have drawbacks in either one of the skewed phases, while our adaptive approach is very close to the optimal solution TPS-O (suffering slightly from dynamic adaption). However, the total runtime of TPS-O (63,523ms) compared to TPS-A (64,612ms) reveals only a negligible overhead of 1,089ms (less than 2%) for plan adaption.

7 CONCLUSION

We presented *TPStream*, a novel event processing operator for detecting complex *temporal patterns* among event streams. We enabled *TPStream* to derive lasting situations directly from streams of events and developed new techniques for detecting *temporal patterns* at the earliest possible point in time. Furthermore, we demonstrated low-cost adaptive approaches suitable for a streaming scenario. We proved the potential of *TPStream* by comparing it to industrial and academic solutions for CEP in experiments.

Since research on situations in CEP is scarce, we focused our efforts on presenting a fundamental solution suited for this scenario and equipped it with the capabilities to handle the adaptive, low-latency nature of stream processing. For future work, we intend to extend *TPStream* to tackle out-of-order arrivals [7, 21] and parallel processing [24, 26].

ACKNOWLEDGMENTS

This work has been supported by the German Research Foundation (DFG) under grant no. SE 553/9-1.

REFERENCES

[1] Mohamed H. Ali and others. 2009. Microsoft CEP Server and Online Behavioral Targeting. *Proc. of the VLDB Endowment* 2, 2 (2009), 1558–1561.
[2] James F. Allen. 1983. Maintaining knowledge about temporal intervals. *Comm. of the ACM* 26, 11 (1983), 832–843.
[3] H.-Jürgen Appelrath and others. 2012. Odysseus: a highly customizable framework for creating efficient event stream management systems. In *DEBS'12*. 367–368.
[4] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, and Anurag Maskey. 2004. Linear road: a stream data management benchmark. In *VLDB'04*. 480 – 491.
[5] Ron Avnur and Joseph M. Hellerstein. 2000. Eddies: Continuously Adaptive Query Processing. In *SIGMOD'00*. 261–272.
[6] Shivnath Babu, Rajeev Motwani, Kamesh Munagala, Itaru Nishizawa, and Jennifer Widom. 2004. Adaptive ordering of pipelined stream filters. In *SIGMOD'04*. ACM, 407–418.
[7] Badrish Chandramouli, Jonathan Goldstein, and David Maier. 2010. High-performance dynamic pattern matching over disordered streams. *Proc. of the VLDB Endowment* 3, 1 (2010), 220–231.
[8] Alan J. Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, and Walker M. White. 2007. Cayuga: A General Purpose Event Monitoring System. In *CIDR'07*. 412–422.
[9] Yanlei Diao, Neil Immerman, and Daniel Gyllstrom. 2007. *Sase+: An agile language for kleene closure over event streams.* Technical Report. University of Massachusetts.
[10] Martin Erwig. 2004. Toward Spatio-Temporal Patterns. In *Spatio-Temporal Databases: Flexible Querying and Reasoning*. Springer Berlin Heidelberg, 29–53.
[11] Opher Etzion, Fabiana Fournier, Inna Skarbovsky, and Barbara von Halle. 2016. A model driven approach for event processing applications. In *DEBS'16*. 81–92.
[12] Ioannis Flouris, Nikos Giatrakos, Antonios Deligiannakis, Minos Garofalakis, Michael Kamp, and Michael Mock. 2016. Issues in complex event processing: Status and prospects in the Big Data era. *Journal of Systems and Software* (2016).
[13] Dengfeng Gao, Christian S. Jensen, Richard T. Snodgrass, and Michael D. Soo. 2005. Join operations in temporal databases. *VLDB Journal* 14, 1 (2005), 2–29.
[14] Thanaa M. Ghanem, Walid G. Aref, and Ahmed K. Elmagarmid. 2006. Exploiting predicate-window semantics over data streams. *SIGMOD Record* 35, 1 (2006), 3–8.
[15] Lukasz Golab and M Tamer Özsu. 2003. Processing sliding window multi-joins in continuous queries over data streams. In *VLDB'03*. 500–511.
[16] Lukasz Golab and M. Tamer Özsu. 2005. Update-pattern-aware modeling and processing of continuous queries. In *SIGMOD'05*. 658–669.
[17] Michael Grossniklaus, David Maier, James Miller, Sharmadha Moorthy, and Kristin Tufte. 2016. Frames: Data-driven windows. In *DEBS'16*. 13–24.
[18] Annika Hinze and Agnès Voisard. 2015. EVA: An event algebra supporting complex event specification. *Information Systems* 48 (2015), 1 – 25.
[19] Bastian Hoßbach, Nikolaus Glombiewski, Andreas Morgen, Franz Ritter, and Bernhard Seeger. 2013. JEPC: The Java Event Processing Connectivity. *Datenbank-Spektrum* 13, 3 (2013), 167–178.
[20] Ming Li, Murali Mani, Elke A. Rundensteiner, and Tao Lin. 2011. Complex event pattern detection over streams with interval-based temporal semantics. In *DEBS'11*. 291–302.
[21] Mo Liu, Ming Li, Denis Golovnya, Elke A. Rundensteiner, and Kajal Claypool. 2009. Sequence Pattern Query Processing over Out-of-Order Event Streams. In *ICDE'09*. 784–795.
[22] Yuan Mei and Samuel Madden. 2009. ZStream: a cost-based query processor for adaptively detecting composite events. In *SIGMOD'09*. 193–206.
[23] Olga Poppe, Chuan Lei, Elke A. Rundensteiner, and Daniel J. Dougherty. 2016. Context-Aware Event Stream Analytics. In *EDBT 2016*. 413–424.
[24] Medhabi Ray, Chuan Lei, and Elke A Rundensteiner. 2016. Scalable pattern sharing on event streams. In *SIGMOD'16*. 495–510.
[25] Mahmoud Attia Sakr and Ralf Hartmut Güting. 2011. Spatiotemporal pattern queries. *GeoInformatica* 15, 3 (2011), 497–540.
[26] Nicholas Poul Schultz-Møller, Matteo Migliavacca, and Peter Pietzuch. 2009. Distributed complex event processing with query rewriting. In *DEBS'09*. 1–12.
[27] Haopeng Zhang, Yanlei Diao, and Neil Immerman. 2014. On complexity and optimization of expensive queries in complex event processing. In *SIGMOD'14*. ACM Press, 217–228.