Distributed and Parallel Databases (2021) 39:361–412
https://doi.org/10.1007/s10619-019-07272-z

TPStream: low-latency and high-throughput temporal pattern matching on event streams

Michael Körber[1] · Nikolaus Glombiewski[1] · Andreas Morgen[1] · Bernhard Seeger[1]

Published online: 5 July 2019

Abstract

Sequential pattern matching to detect a user-defined sequence of conditions on event streams is a key feature in modern event processing systems. However, the sequential nature of event based pattern matching has two major deficiencies. First, it is hardly possible to express complex temporal relationships between *situations* lasting for periods of time. Because events are equipped with a single timestamp only, the expressible temporal relations are limited to before/after/at the same time. Second, a sequential pattern is mapped to a continuous subsequence of the input stream starting with an arbitrary event, making efficient parallelization a hard problem. In this paper we present *TPStream*, a novel event processing operator for complex temporal pattern matching on event streams. *TPStream* first summarizes incoming events to *situations* lasting for periods of time, before it matches temporal patterns. With *situations*, temporal patterns can easily be defined based on Allen's interval algebra. We also show that situation based temporal pattern matching can be efficiently executed in parallel using multiple threads on a single machine or multiple machines in a cluster. Finally, we present adaptive optimization components continuously tuning the execution strategy of *TPStream* towards the lowest possible result latency with respect to the overall system load. The results of our experimental evaluation show that *TPStream* is capable of processing high-volume event streams with both low latency and high throughput while outperforming applicable CEP solutions from academia and industry.

✉ Michael Körber
koerberm@mathematik.uni-marburg.de

Nikolaus Glombiewski
glombien@mathematik.uni-marburg.de

Andreas Morgen
morgen@mathematik.uni-marburg.de

Bernhard Seeger
seeger@mathematik.uni-marburg.de

[1] Database Systems Group, University of Marburg, Marburg, Germany

Keywords Event processing · Pattern matching · Query optimization · Parallel event processing · Distributed event processing

1 Introduction

Analyzing massive streams of data in near real-time is a mission-critical task in various application domains, including infrastructure monitoring, traffic monitoring, health care and financial applications. A special form of data stream processing is (complex) event processing. From a user's perspective, an event can be seen as a notification that something happened at a specific point in time. When ordered by time, a sequence of these notifications form an event stream. For example, a temperature sensor naturally generates an event stream by periodically measuring the temperature and reporting it with the time of measurement. Technically, an event is composed of some structured information (e.g., the temperature value) and a timestamp, and an event stream is a potentially unbounded sequence of events, ordered according to this timestamp.

Besides filtering, aggregating and joining event streams, a key feature in event processing systems is *sequential pattern matching*. Typically, a pattern is expressed as a regular expression over a set of symbols, whereby every symbol is associated with a user-defined condition on the input data. A match of the pattern is detected if the event stream contains a sub-sequence of events fulfilling the conditions in the specified order.

However, the sequential nature of regular expression based patterns has two major deficiencies. First, the expressible temporal relationships are limited to before/after/at the same time relationships. Conditions lasting for periods of time and their temporal relationships (e.g., A happens during B) can not or only hardly be expressed in this approach. Second, due to the sequential nature of this process efficient parallel execution strategies are scarce. Nevertheless, efficient parallel and distributed execution is a crucial aspect to deal with ever increasing data rates.

To overcome these deficiencies, we introduce the concept of *situations*. A *situation* is a period of time for which a set of conditions holds true. *Situations* can be derived from event streams on-the-fly and a *temporal pattern* can be matched using the *situations'* time intervals. This reduces the input data by (i) summarizing relevant sub-sequences of the stream and (ii) filtering out events not relevant for matching the pattern, which, as we will show, allows for efficient parallel and distributed query processing.

We illustrate this idea with the following example. A traffic monitoring system is continuously receiving sensor data from connected cars (i.e., position, speed, acceleration). One of the system goals is to notify drivers about potential threats 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 aggres-

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

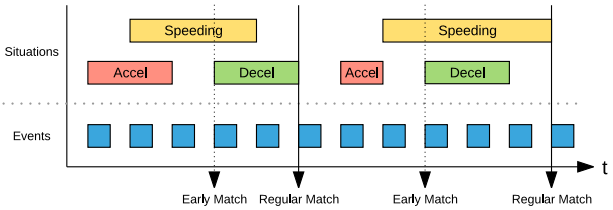

Fig. 1 Detecting aggressive driving with situations

sive drivers could be stated as: "A sharp acceleration followed by hard braking, both accompanied by a period of speeding."

Figure 1 illustrates this example. The stream of raw sensor readings is transformed into three situation streams, one for each component of the pattern. Each situation consists of a time interval describing the temporal validity and a meaningful summarization of the event sequence it was derived from (e.g., the average speed during the phase of speeding). The *temporal pattern* can then be matched by joining streams of situations with appropriate conditions. Figure 1 also showcases two more desirable features for temporal pattern matching. First, the way the situations are temporally related to each other differs slightly among the two sketched matches. In the first match they *overlap*, while in the second match deceleration happens *during* the speeding situation. The query language should be flexible enough to cover these cases within a single query. Second, the pattern should be detected with the *lowest possible latency*. As depicted above, both matches may be concluded at the beginning of the deceleration situation, since speeding holds true at this point in time 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.

While the problems sketched above are described based on the assumptions of incoming point events, there is some research on native support for complex temporal pattern matching [37]. However, the major drawback of these approaches is that they rely on externally created interval events as input, and hence are not able to conclude matches before the end of all situations is known.

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

- *TPStream* is the first event processing operator to closely couple derivation of situations with pattern matching, enabling match detection at the *earliest possible point in time*.
- 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 an easy-to-read query-language for temporal pattern matching.
- Unlike previous research, 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.
- In this extended version of a previous paper [36], we show that temporal pattern matching via *situations* can be efficiently parallelized—on a single machine as well as in a shared-nothing cluster environment.
- Additionally, we present an auto-tuning component capable of tuning the parallelization degree towards minimizing both, result latency and resource consumption.

The rest of the paper is organized as follows. The next section discusses alternative solutions to temporal pattern matching in detail. Section 3 reviews related work, before we introduce *TPStream*'s query language in Sect. 4. In Sect. 5 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 Sect. 6. Solutions for parallel and distributed execution of *TPStream* are introduced in Sect. 7. We evaluate the performance of *TPStream* in Sect. 8 and conclude this paper in Sect. 9.

2 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 [37]. However, *ISEQ* has several shortcomings concerning the desired features: first, the operator requires interval-events (i.e., situations) as input, leaving all aspects of deriving situations to an unspecified external entity. Being unaware of the origin of interval-events severely limits the operator in processing power (in terms of plan optimization) and most importantly renders a detection with the lowest possible latency impossible since there is no way to directly access an incomplete situation or indirectly manipulate the building of a situation through constraints. 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 $\texttt{A.ts} < \texttt{B.ts} < \texttt{A.te} \leq \texttt{B.te} \vee \texttt{A.ts} < \texttt{B.ts} < \texttt{B.te} < \texttt{A.te}$ on two situations `A` and `B` is expressed as $\texttt{A.ts} < \texttt{B.ts} < \texttt{A.te}$. Hence, disjunctions like $\texttt{A.ts} < \texttt{B.ts} < \texttt{A.te} < \texttt{B.te} \vee \texttt{B.ts} < \texttt{A.ts} < \texttt{B.te} < \texttt{A.te}$ 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. Third, *ISEQ* does not provide any solution to parallelizing the process of matching complex *temporal relations*. Finally, *ISEQ* relies on auxiliary index structures and punctuation mechanisms for efficient query execution, complicating the integration into existing systems.

2.1 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 summarizations. Technically, this means matching patterns of the form `!S S+ !S` with `S` being the situation's condition (e.g., `speed > 70 mph`) which identifies the longest contiguous subsequence of events fulfilling `S` by surrounding this sequence (`S+`) with events not fulfilling the condition (`!S`). 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. While this approach is able to derive situations including the desired summarizations, it fails to produce early results 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: $\texttt{A} : \texttt{accel} > 8\ \texttt{m/s}^2$, $\texttt{C} : \texttt{speed} > 70\ \texttt{mph}$ and $\texttt{B} : \texttt{A} \wedge \texttt{C}$. Since patterns are expressed on the granularity of events, early results are achieved by simply omitting the last portion of the pattern. At the same time summarizations of single situations are left to a post-processing step, since situations are disassembled to express temporal overlaps.

3 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 five headlines: Event Pattern Matching, Context/State in CEP, Stream Reasoning, Spatio-Temporal Database Systems and Parallel Pattern Matching.

Event pattern matching Systems capable of CEP (e.g. [5,17]) are generally closely associated with a pattern matching operator. Zhang et al. [49] features a discussion on the several different semantics of the operator and a recent survey [21] 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 Allen [3]—a

crucial aspect for dealing with long-lasting situations. Cayuga [16], ZStream [40] and Microsoft StreamInsight [2] 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 Cayuga). As we will show, this approach results in significant performance deficiencies.

Context/state in CEP There has been recent work on introducing *contexts* into a CEP environment. CAESAR [43] 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 described by Etzion et al. [20] are used to group up event types to process them together. While contexts and 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 dealing with periods of times such as states [30] or aggregating windows [25,28], cover the deriving aspect of situations, but lack interval relations [3] or pattern matching.

Stream reasoning The semantic web community has worked on extending RDF triples with a time dimension, introducing the concept of temporally limited information, and thus allowing for a variety of graph-based temporal queries such as temporal joins [24]. C-SPARQL [11] deals with streams in particular, introduces window semantics and corresponding aggregation capabilities, but does not feature temporal relationships necessary for formulating CEP-style pattern queries. EP-SPARQL [4] and its implementation ETALIS assume that each data item is associated with a fixed time interval and support the usage of Allens temporal operators when formulating queries. Similarly, TEF-SPARQL [35] supports all temporal relationships in a language and algebra build around always valid triples and so-called *facts*. Analogous to situations, facts feature a start time and a potentially undetermined expiration time which may be set in the future. Like *TPStream*, EP-SPARQL and TEF-SPARQL aim to solve the combination of fleeting events with longer lasting situations, but are ultimately complementary to the work presented here due to a variety of reasons. First, the approaches aim to empower SPARQL with event semantics while we look at the problem from the perspective of event processing in order to facilitate easy integration into existing CEP languages and systems. Second, both works focus on establishing the language in particular while *TPStream* also deals with the implementation side in order to improve latency and parallelization aspects. Finally, a more recent study [23] showed performance benefits of TEF-SPARQL's algebraic notion of ongoing facts by compiling them into Esper queries. We go a step further by not only supporting ongoing situations, but also developing optimization mechanisms unique to the pattern matching process.

Spatio-temporal database systems The spatial databases community studied the problem of *spatio-temporal pattern queries* (STPQ) in trajectory databases, e.g. in Erwig [19]. 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 Sakr and Güting [45] in particular served as a foundation for our proposed *TPStream* operator as *TPStream* adapts similar concepts of temporal predicates and constraints. Similar to the spatial community, Helmer and Persia [29] introduce an event query language for high-level event detection for *temporal* databases. The approach is based on introducing Allen's interval algebra into PostgreSQL and can also be applied for video stream surveillance [41]. While those efforts focus on language and databases, *TPStream* targets data streams while highlighting efficient methods for both low latency and parallel processing.

Due to relating multiple types of situation to each other based on their point of occurrence, our evaluation method is also related to temporal joins (see [22] for an excellent survey). Even though most research is not based in stream processing, Piatov et al. [42] present a sweep-line algorithm for joins based on temporal *overlap* predicates. To evaluate interval relations, a specialized hash-index on intervals' start and end points is used. Dignös et al. [18] also target joins with *overlap* predicates, but focus on long-lasting intervals. To speed up processing while dealing with intervals of varying sizes, the authors adaptively divide intervals into temporal partitions of different granularities while reducing the join to temporally overlapping partitions. Both of those works are orthogonal to *TPStream*, since they optimize for overlap predicates on stored interval data, while *TPStream* derives intervals on the fly, allows for multiple arbitrary temporal relations and combines both aspects for low latency pattern detection. A more general approach called the Timeline Index [34] was presented for main memory database systems. The Timeline Index is an append-only data structure supporting efficient methods for time travel, temporal aggregation and temporal join queries. While the Timeline Index is designed to support a wide variety of queries in multi-version workloads (i.e., including records updates and queries covering the whole history), data structures used in *TPStream* are optimized for streaming workloads (i.e., insert-heavy with windowed queries for recent items). In comparison to join algorithms on streams [8,26] as well as adaptive approaches [7], *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.

Parallel pattern matching There is a wide variety of Frameworks (e.g. Apache Spark [48], Apache Flink [12]) and research (e.g. [1,38]) on parallel and distributed execution of streaming applications. However, since pattern matching is generally sequential in nature, work on parallel approaches is much more limited. To categorize the different approaches they can generally be grouped into either *task*, *data* or *pipeline* parallelism [15].

Task parallelism naturally occurs when executing multiple non-chained streaming operators concurrently. Ray et al. [44] propose a technique to find shared sub-patterns among multiple operators and compute them only once. Zhang et al. [49] propose a similar technique that finds a generalization of a pattern and delays more detailed evaluation that could be handled in parallel. Both works do not consider situations

and feature techniques best suited for multi-query optimization. We focus on the orthogonal approach of intra-query parallelism. Nevertheless, we design our solution in a way that allows applying above techniques and highlight how crucial points of dealing with situations effect the design of tasks.

In *data parallelism*, data blocks can be processed independently, e.g. in a query with a group-by clause. Hirzel [31] discusses how to semantically introduce the partition-by keyword into a CEP language based on a automaton evaluation technique. One of our techniques also handles partitioned data, but is a join-based approach and focuses on handling situations. Schneider et al. [46] developed a system that automatically applies safe data parallelism techniques (e.g. partitioning) for parts of an operator graph. The paper does not discuss pattern matching. Furthermore, most techniques cannot be directly applied to achieve intra-parallelism of the *TPStream*'s components, because the matcher features multiple predecessors and a non-partitioned state which the system considers unsafe for parallelism. Regardless, we also apply the technique of sequence numbers to deal with synchronization issues for our non-partitioned data strategy. Most similar to our work is RIP [10]. Instead of partitioning data by key, the authors propose to partition data by run, i.e. distributing consecutive batches of events. To ensure correct results for patterns spanning multiple batches, RIP replicates a certain number of events based on the pattern length. We use a similar technique of run-based partitioning for non-partitioned data, but cannot rely on pattern length and have to adapt the idea to handling situations. In addition, we develop an adaptive strategy for deployed threads and batch size.

Finally, *pipeline parallelism* can be applied to queries featuring chained operators that are processed in sequence. This chain is parallelized by assigning computing resources to each operator. Schultz-Møller et al. [47] propose to rewrite the pattern matching operator into several smaller chained tasks that are distributed and reused over multiple queries. Jayasekara et al. [33] design a framework for distributing streaming operator graphs including task, data and pipeline partitioning method in a cluster environment. Pattern matching is handled in the pipeline sense by distributing filter operations and joining their result on another node. Cugola and Margara [14] use the CPU to evaluate whether an event matches a state in the pattern matching automaton and the GPU to produce result sequences and aggregates. All those works aim to use external resources to speed up parts of the pipeline. For a single machine, we decided not to rely on pipeline parallelism, since we observed that switching threads between deriving and matching causes unnecessary overhead. However, we do discuss how our techniques can be converted to a pipeline parallel model for a cluster environment.

4 Query language

In this section, we present *TPStream*'s query language for specifying temporal pattern queries over point event streams. Therefore, we first present the basic structure of a *TPStream*-query. Then, we discuss every component in detail, before we formulate the aggressive drivers query from the introduction as an example. Finally this section is closed with a discussion on the expressiveness of the proposed language.

4.1 Syntax

The basic structure of the language is based on *match recognize*, a language for sequential pattern matching in the SQL:2016 standard (items in square brackets are optional):

```
FROM          <input stream>
[PARTITION BY <attributes>]
DEFINE        <situation definitions>
PATTERN       <pattern definition>
WITHIN        <duration>
RETURN        <output definition>
```

The `FROM` clause selects the input stream for the query. If the physical input stream carries multiple logical partitions, where each of them should be evaluated separately, this can be specified in the optional `PARTITION BY` clause. The partitioning scheme is specified via one or more `<attributes>` of the input stream, whereby each unique combination of attribute values defines a logical partition. To limit the search space for matches of the *temporal pattern*, a time window in which the pattern must occur completely is defined via the `WITHIN` clause. The duration is specified as `<number> <time unit>`, with time unit being one of second(s), minute(s), hour(s).

The `DEFINE` clause specifies which situation streams should be derived from the raw events and how to derive them:

```
<situation definitions> := <situation definition> |
                      <situation definition>, <situation definitions>
<situation definition>  := <name> AS <condition>
                        [<duration constraint>]
```

A situation stream definition consists of a unique `name` and a `condition`. Conditions are constructed using boolean expressions (`true`, `false`, `and`, `or`, `not`), predicates ($<$, $>$, $\leq$, $\geq$, $=$, $\neq$) and arithmetic expressions composed of constant values and references to attributes of the input stream. The situations (time-intervals) composing a situation stream are derived by applying those conditions to every incoming event. Optionally, a `duration constraint` can be set, allowing to restrict the length of derived situations:

```
<duration constraint> := AT LEAST <duration> |
                         AT MOST <duration>  |
                         BETWEEN <duration> AND <duration>
```

This means that only situations fulfilling this constraint are considered for matching the *temporal pattern*.

The `PATTERN` clause specifies the *temporal pattern* as a set of *temporal constraints* between two situation types:

```
<pattern definition> := <constraint> |
                        <constraint> AND <pattern definition>
<constraint>         := <name> <relations> <name>
<relations>          := <relation> |
                        <relation>;<relations>
```

The meaning here is as follows. A set of situations matches the *temporal pattern* if all defined constraints are satisfied. A `constraint` between situations from two

Table 1 Allen's interval algebra

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

streams is satisfied if their *temporal relation* is among the specified `relations`. In other words, a constraint is a disjunction of *temporal relations* (delimited by `;` in the query language). In order to express *temporal relations* between situations, we adopt Allen's Interval Algebra (see [3]) depicted in Table 1 for two 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*. *Relation* (*R*) between `A` and `B` is represented through the relation between these four points as given by *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 both within the interval `B`.

The last part of a *TPStream* query is the `RETURN` clause defining the shape of result events:

```
<output definition> := <output variable> |
                       <output variable>, <output definition>
<output variable>   := <aggregate> as <name>
<aggregate>         := FIRST(<ref>) | LAST(<ref>) |
                       COUNT(<ref>) | SUM(<ref>) | ...
```

Every attribute of the output event stream is an aggregation over an `attribute` of the input-stream. It summarizes all events that occurred during the time interval of a situation. The attribute reference (`<ref>`) is composed of the name of the situation stream to refer to and the name of an attribute of the input stream. Besides the standard aggregation functions like `COUNT` or `SUM`, *TPStream* allows to refer to values of the `FIRST` and `LAST` event participating the specified situation.

Listing 1 shows the query definition for detecting aggressively driving cars from the introductory example. The query processes point events from the `CarSensors` stream which is partitioned by the `car_id` to evaluate each driver individually. We define the three situations for sharp acceleration `(A)`, speeding `(B)` and hard breaking `(C)`, by referring to the acceleration and speed attributes of the input stream. All three definitions make use of duration constraints. The pattern allows for different combinations among the derived situation types. For instance, acceleration may `meet`, `overlap`, `start` or occur `during` a phase of speeding. The total length of a match is restricted to 5 minutes and result events are composed of the unique `car_id` and the average speed during the speeding situation.

```
FROM     CarSensors PARTITION BY car_id
DEFINE   A AS accel >  8m/s^2 AT LEAST 5s,
         B AS speed > 70 mph BETWEEN 4s AND 30s,
         C AS accel < -9m/s^2 AT LEAST 3s
PATTERN  A meets;overlaps;starts;during B
     AND B contains;finishes;overlaps;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

4.2 Expressiveness

Most common CEP systems define patterns based on symbols connected via regular expressions. Specific extensions, like aggregation, put the expressiveness of those languages between regular and context-free grammars [49]. 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 Sect. 2, 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 [49] 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, in some languages [17] symbols can 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 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 [32] fea-

tures 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.

5 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 formally model those aspects (streams, data, deriving situations and temporal pattern matching) in an algebra.

5.1 Stream model

Definition 1 (*Data stream*) A data stream D is a potentially unbounded sequence of data items $\langle d_1, d_2, \ldots \rangle$ totally ordered by a relation $<_D$. $d_i \in D$ refers to the ith 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 readability, we will generally assume that each item in a data stream is unique (i.e., $\forall d_j^i, d_k^i \in D^i : j \neq k$) and refer to previous work on the matter of handling potentially equal elements (see [16]). $\langle\rangle$ 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 $<_D$ and no contemporary items between D^1 and D^2, 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 $\biguplus_{i=1}^{n} D^i$.

5.2 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 (e.g. [13,39]) for future work.

5.3 Derivation

Situations are derived from event streams through aggregation and predicate evaluation. First, we 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:

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

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 $\tau := [d_{min}, d_{max}]$:

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

Example Assume the query in Listing 1 derives a speeding situation for a car with the time interval [2, 10). This means `speed` $\leq$ `70 mph` at $t = 1$ and $t = 10$ and in between those timestamps `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 important property means 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 as stated by Golab and Özsu [27]. Due to space limitations, we omit a formal proof here.

5.4 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.

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 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 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* $\{C^{i,j} | 1 \leq i < j \leq m\}$. 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*:

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

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):

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

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:

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

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

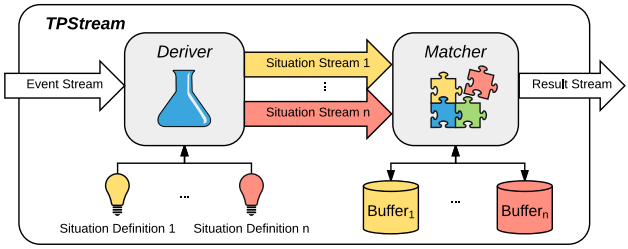

Fig. 2 *TPStream* architecture

Algorithm 1: DeriveSituations

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

6 Algorithms and 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 Fig. 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 the discussion on low latency detection to Sect. 6.3 and 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.

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     (C, B, nextStep) ← getEvaluationOrder(s);
5     performMatch({s}, (C, B, nextStep));
```

6.1 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, three 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). Otherwise, if the current event fulfills the predicate, the temporal validity of a started situation is prolonged. 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).

6.2 Matching the pattern

The *matcher* implements an incremental version of $TPStream_{w,\hat{\gamma}}$ (Definition 13). 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) by removing all situations `s` with $\texttt{s.ts} < \texttt{t} - \texttt{window}$. Because of the mentioned ordering, this effectively means, finding the first situation $\texttt{s}'$ with $\texttt{s}'\texttt{.ts} \geq \texttt{t} - \texttt{window}$ and discarding all previous situations. The buffers are implemented via array-backed ring buffers, which efficiently support these operations. After purging outdated situations, the following steps are executed for every received situation. First, it is added to the corresponding buffer (Line 3). Then, we obtain the evaluation order for the given situation (Line 4).

Algorithm 3: PerformMatch

```
   Input: ws : working-set, step : (C : Temporal Constraints, B : Situation Buffer, nextStep)
1  if step = ∅ then
2      publishResult(ws);
3  else if containsSituationForStep(ws,step) ∧ checkConstraints(ws, C) then
4      performMatch(ws, nextStep);
5  else if ¬containsSituationForStep(ws,step) then
6      foreach (p, t_s, t_e) ∈ findMatches(C, B, ws) do
7          performMatch(ws ∪ {(p, t_s, t_e)}, nextStep);
```

That is the order how situation buffers are joined. For every join-step, we store the set of *temporal constraints* to be fulfilled ($\mathcal{C}$), the buffer to join (B) and a reference to the next step. Finally, the recursive matching algorithm (*performMatch*) is called. In order to ensure incremental and unique results, the currently processed situation serves as a parameter and is part of every successful match.

PerformMatch (Algorithm 3) matches the pattern as follows. In each recursive step, one situation buffer is joined with the current partial result (*working-set*). Here, two cases must be distinguished. The first case handles situations that were pre-set in the *working-set*, which accounts for the situation passed as a parameter from *updateMatcher*. In this case, we omit joining the whole buffer, check the *temporal constraints* of the current step ($\mathcal{C}$), and proceed recursively, if the constraints are fulfilled (Lines 3, 4). The second case regularly joins the current *working-set* with the buffer (B) of the currently processed evaluation step and proceeds recursively to the next step for each matching situation (Lines 5–7). If all steps are processed, the *working-set* contains a situation from every buffer and all *temporal constraints* are satisfied for this set of situations. Hence, the algorithm calls *publishResult* to materialize a result event and push it into the output stream (Lines 1, 2).

Obviously, the evaluation performance of *performMatch* mainly depends on the efficiency of the function *findMatches*. A naïve approach would be, to scan the entire buffer and check the *temporal constraints* for each situation separately. With R_i denoting the ith 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. 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 obtain matching instances of `B` by (i) issuing two range-queries

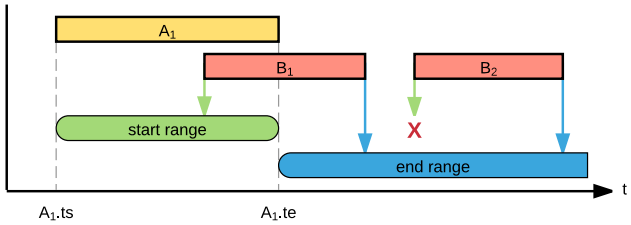

Fig. 3 Temporal matching via range queries

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. $\texttt{A.ts} < \texttt{ts} < \texttt{A.te}$ for the start-timestamp and
2. $\texttt{A.te} < \texttt{te} < \infty$ 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 $\texttt{A}_1$ in combination with the *temporal relation* is used to build the two search ranges. After intersecting the results ($\{\texttt{B}_1\}$ for the start range and $\{\texttt{B}_1, \texttt{B}_2\}$ for the end range), we receive our final result $\texttt{B}_1$. 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.

In general, 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:

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

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.

6.3 Low-latency matching

In this section, we will determine the earliest points in time $tr_{min}(R)$, $tc_{min}(C)$, $tp_{min}(P)$ to detect a *temporal relation* R, *temporal constraint* C, and *temporal pattern* P, respectively. Then, we illustrate cases in which the algorithms of Sect. 6.2 in combination with low-latency matching fail to deliver correct results. In the last part of this section, we present adjusted algorithms for low-latency matching.

Table 2 Temporal relations R and their prefix groups G with their earliest detection times $tr_{min}(R)$ and $tg_{min}(G)$

Relation (R)	Definition (δ_R)	tr_{min}(R)	Prefix-Group (G)	tg_{min}(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		

6.3.1 Analysis

Two situations `A,B` can only be related once we know they exist, making $\max(\texttt{A.ts}, \texttt{B.ts}) \leq tr_{min}(R)$ a trivial lower bound for *all* relations R. For an exact computation of $tr_{min}(R)$ we consider the definition δ_R of relations given in Table 2. Let $\texttt{t}_1 \leq \texttt{t}_2 \leq \texttt{t}_3 \leq \texttt{t}_4$ be the timestamps in the order they appear in δ_R. It is easy to see that the ordering of $\texttt{t}_4$ is already available at $\texttt{t}_3$, because $\texttt{t}_3 \leq \texttt{t}_4$ and there are no timestamps after t_4. Furthermore, there are multiple relations sharing the same definitions up to $\texttt{t}_2$, i.e., it is not possible to distinguish those relations from each other. We group these relations with a common prefix of two timestamps into so-called *prefix groups* as shown in Table 2 for relations starting at situation A (an analogous definition exists for relations starting with situation `B`). Thus, we conclude that $tr_{min} = \texttt{t}_3$ holds.

A *temporal constraint* $C = (R_1, \ldots, R_n)$ for `A,B` matches if one of the contained relations matches. Therefore, C has multiple earliest detection times given by a set $tc_{min}(C) = \{tr_{min}(R_1), \ldots, tr_{min}(R_n)\}$. Further, if C contains *all* relations of a *prefix group* G (cf. Table 2), the detection time of these relations is shifted to the trivial lower bound of that group (denoted $tg_{min}(G)$).

Finally, for a pattern $\mathcal{P} = (C_1, ..., C_m)$, each constraint $C_i = (R^i_1, \ldots, R^i_{n_i})$ must be matched. However, a single *temporal configuration* matching $\mathcal{P}$ fulfills exactly one *temporal relation* $(R^i_{j_i} \in C_i, i = 1, \ldots, m)$ from each constraint, making $tp_{min}(\mathcal{P}) = \left\{ max(tr_{min}(R^1_{j_1}), \ldots, tr_{min}(R^m_{j_m})) \mid 1 \leq j_i \leq n_i, 1 \leq i \leq m \right\}$. Thus, $tp_{min}(\mathcal{P})$ is among all the constraint detection points, i.e., $tp_{min}(\mathcal{P}) \subseteq \bigcup_{i=1\ldots m} tc_{min}(C_i)$.

6.3.2 Problem statement

For the ease of presentation, we first postpone the discussion of optional duration constraints on situations as well as *prefix groups* to the end of this section. In general, our low-latency analysis provides two insights for the matching algorithm. First, new matches only occur if a new situation starts or a situation ends. Second, only a subset of the situations in a pattern $\mathcal{P}$ can produce a match at a point in $tp_{min}(\mathcal{P})$. Thus, the matching process is delayed until a situation with at least one endpoint in $tp_{min}(\mathcal{P})$

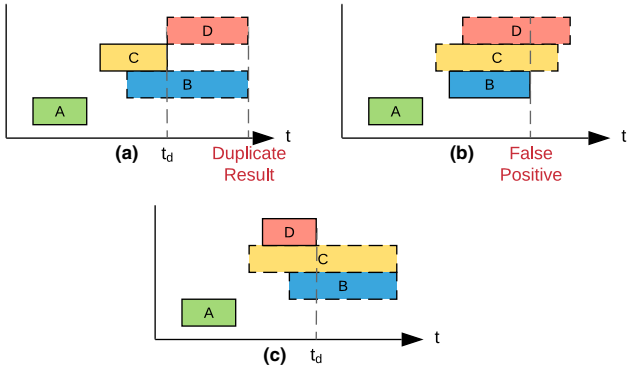

Fig. 4 Earliest detection time ($tp_{min}(\mathcal{P})$) of different *temporal configurations* for the sample pattern $\mathcal{P}$

occurs without affecting the latency. We call those situations *trigger situations* since only they *trigger* a *performMatch* call. These insights affect our algorithms in the following ways. Situations must be available for matching from their start. This requires an adjustment of the *deriver* component. Additionally, we need to determine for each situation stream if the derived situations are *trigger* situations. For *trigger* situations the algorithm computes the point in time to execute *performMatch* (at its start, end or both).

In contrast to our algorithms so far, the following two cases must be considered during the matching process in order ensure the correctness of the produced results while delivering them as early as possible. First, unique results were ensured by examining a new situation on every invocation of *performMatch* (Algorithm 3). However, due to the fact that for low latency matching both endpoints of a situation must be considered, *performMatch* might be called twice with the same pre-set situation. Hence, additional steps are required to guarantee unique results. Second, situations whose endpoint is unknown always carry the current time as a temporary end timestamp. This affects the matching of *temporal relations* that require both situations to end at the same point in time (i.e. `finishes`, `equals`). If two ongoing situation are inspected by the matcher they might fulfill the respective *temporal relation* even though their true end timestamps differ, which in turn would lead to false positive matches. Note that other relations are not affected by this phenomenon, since they require their end timestamps to differ, which always evaluates to false for two ongoing situations.

We illustrate both cases using the following example pattern $\mathcal{P}$ on four situations: (`A, B, C, D`):

```
A before B AND A before C AND A before D AND
C contains;finishes;meets D
```

It defines situation `A` as the starting point of every match. `B` is not explicitly related to either `C` or `D` and is required to happen after `A`. Consequently, `B` is a *trigger* situation and `B.ts` is in $tp_{min}(\mathcal{P})$. For `D`, both `D.ts` (via `meets`) and `D.te` (via `contains,finishes`) are in $tp_{min}(\mathcal{P})$.

For temporal configurations as shown in Fig. 4a, the earliest point of detection for $\mathcal{P}$ is `D.ts`. However, since `D.te` is in $tp_{min}(\mathcal{P})$, the same match would be detected again

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      (C, B, nextStep) ← getEvaluationOrder(s);
5      if matchOnStart(s) then
6          performMatch( {s}, 0 );
7          U ← getUnrelatedStarted(s);
8          foreach u ∈ powerset(U) \ ∅ do
9              performMatch( u ∪ {s} , (C, B, nextStep) );
10 foreach s ∈ S_f do
11     startedBuffer.remove(s);
12     addToBuffer(s);
13     if matchOnEnd(s) then
14         (C, B, nextStep) ← getEvaluationOrder(s);
15         R ← getRelatedStarted(s);
16         foreach r ∈ powerset(R) \ ∅ do
17             performMatch( r ∪ {s} , (C, B, nextStep) );
18             U ← getUnrelatedStarted(s);
19             foreach u ∈ powerset(U) \ ∅ do
20                 performMatch( r ∪ u ∪ {s} , (C, B, nextStep) );
```

at `D.te`. Figure 4b showcases the false positive detection if validating `C finishes D` would succeed at `B.te`. Finally, Fig. 4c illustrates that two ongoing situations like B and C contribute to the match even if they are not explicitly related via a *temporal constraint*.

6.3.3 Low-latency matching algorithm

Instead of handling the above 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 the second situation to be finished in the past. On the other hand, *temporal relations* triggering matching on a situation's end require the second situation 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 an additional buffer and if $s.ts \in tp_{min}(\mathcal{P})$, 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 7–9). This accounts for configurations as seen in Fig. 4a. All finished situations are migrated from the separate to the regular buffer (Lines 10–12) and if $s.te \in tp_{min}(\mathcal{P})$, the matching process is triggered (Lines 13, 14). This time with combinations of s and all *started and related* situations (Lines 15–17), further combined with all *started and unrelated* situations (Lines 18–20), which fuses the avoidance of duplicate results and false positives. An example for this case is shown in Fig. 4c. 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 in Sect. 8, 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 Only a few modifications are required to incorporate *duration constraints* on situations into low latency-matching. 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 $tp_{min}(\mathcal{P})$, 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 $tp_{min}(\mathcal{P})$. As an example, consider the pattern `A during B` and the following order of timestamps: $\texttt{B.ts} < \texttt{A.ts} < \texttt{A.te} < \texttt{B.}\overline{\texttt{ts}} < \texttt{B.te}$. This match can not be detected at `A.te`, because `B`'s duration does not exceed the lower bound at this point. Hence $\texttt{B.}\overline{\texttt{ts}}$ requires a matcher invocation.

Prefix groups 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. In our algorithm this means splitting the *temporal constraint* into two disjoint sets, one containing all *temporal relations* forming a *prefix group* and another one for the remaining relations. The first set is used when matching on a situation's start while the second is used on its end.

6.4 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, 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 our optimizer enumerates possible execution plans, computes their expected computational costs and determines the plan with minimum cost. We do not provide multiple implementations of the join operator,

Table 3 Initial estimates for the selectivity of *temporal relations*

Relation	Before	During	Overlaps	Starts, finishes, meets	Equal
Selectivity	0.445	0.03	0.01	0.0049	0.0006

Mirror relations are equivalent

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

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

s_i denotes the selectivity of the applicable *temporal constraints* in step i ($\mathcal{C}_i$), which can be composed from the selectivities of the contained *temporal relations* as follows:

$$s_i := \prod_{C \in \mathcal{C}_i} \left(\sum_{R \in C} s_R \right) \tag{4}$$

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. Sect. 8.4.2).

6.4.1 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 ith 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 Sect. 8.4.2, the total costs for adaptivity are negligible.

7 Parallel *TPStream*

The goal of this section is to present a parallel version of *TPStream* to improve throughput by leveraging the parallel processing power of today's multicore systems. This goal should be achieved without any latency degradation. The nature of *TPStream* makes its parallelization a non-trivial task for the following two reasons: First, *TPStream* works with continuous sequences of situations derived from continuously incoming events. To identify a pattern P, either a thread must process each event in a sequence that leads to the detection of P, or this knowledge must be exchanged among the threads involved. Second, a key component of *TPStream* to achieve low latency is its situation buffer. However, in a parallel version, the effort involved in synchronizing the accesses of different threads to the buffer can become a serious performance bottleneck.

In the following, the most important concepts for introducing parallelism for *TPStream* are presented. We start with the static integration of *TPStream* into the application context from which the event streams come and to which the results of continuous pattern queries are sent. We then develop two multi-threaded versions of the *TPStream* query processing algorithm. The first version is applicable when the input stream is partitioned with one or more attributes of the incoming events, while the second version applies to non-partitioned input streams. Parallelizing the latter case is more difficult and requires a finely tuned processing pipeline, which we describe in detail in separate subsections. Next, we address the problem of the fluctuating behavior of event streams (e.g., changes in data rates) and present an auto-tuning component that overcomes the limitations of our static approaches presented so far. At the end of this section we present an extension of parallel *TPStream* for a distributed streaming

environment and show how *TPStream* can be integrated into a distributed streaming platform.

7.1 Integration with application context

Before we go into the details of parallel processing, we discuss the import of event streams and the export of result streams. To exploit parallelism, the processing threads must be decoupled from both the (external) event producers and the result consumers.

For the decoupling from event producers, we deploy queues on the thread boundaries in which the producer is writing events while processors are reading from the queues. Because the synchronization of producer and processors introduces a reasonable overhead, we decided to access the queues in event-batches of fixed size rather than single events. The batch size is initially constant, but the auto-tuning component (see details in Sect. 7.4) can adjust it if changes occur in the application context and system load. Furthermore, the size of a queue is constant per active thread, resulting in a stable and predictable memory footprint. In our experiments we found that the capacity of a queue should be 2^{16} events per active thread. An overload is propagated upstream via backpressure allowing for countermeasures on the producer side (e.g., writing events into logs). In general, the goal is to set the batch size in a way that minimizes the waiting time on both, the producer and consumer side. We experimentally examined various settings and found that a batch should be a power of two, at least 32 and at most 2^{15} (i.e. half of the number of events in a queue). In the extreme case of two batches, the producer is writing in its batch, while a processor is reading from the other batch. Depending on the parallelization approach, we either use a shared queue for all threads or a dedicated queue per thread. In the second case, each queue stores up to $2^{16}/batchsize$ batches.

Additionally, the processing threads need to be decoupled from the result consumers. In particular, we have to treat out-of-order results which are likely to occur during parallel processing. Consider for example the following (simplified) scenario of two processing threads $(t_1; t_2)$ that process two consecutive event-batches $(b_1; b_2)$ in parallel. Because all events in b_1 have a smaller timestamp than events in b_2, the results of t_2 will have a timestamp greater than those of t_1. However, if t_2 finishes before t_1 and the results of t_2 are sent downstream before the results of t_1, the (merged) result stream does not have a monotonically increasing time order. A naïve solution to this problem is to wait for a result from every processing thread and only forward the result with the smallest timestamp at a time. However, this may lead to unpredictable delays and block all processing threads if one of them is not producing any results. Therefore, we use a K-slack [9] working as follows: Results are collected in a min-heap of fixed size (K events). The first K events are simply stored in the heap. From event $K + 1$ on, the new result is put into the heap and the top of the heap (i.e., the event with the smallest timestamp) is removed and sent downstream. This method ensures that threads are not blocked during result propagation at the expense of potential out-of-order results. If a result with a timestamp smaller than the top of the heap arrives, we are unable to decide whether it is out-of-order or not. However, depending on the downstream operators we can decide to either publish it because it can be handled or discard it and report

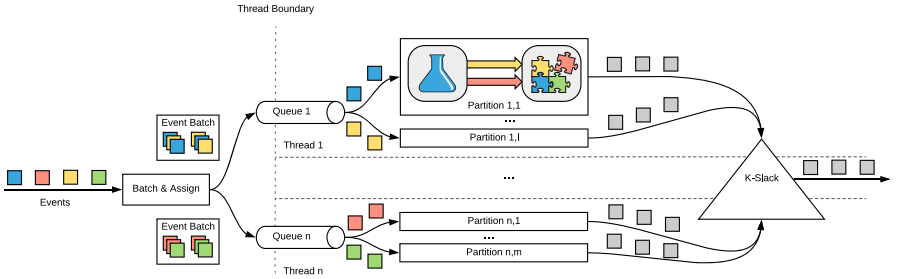

Fig. 5 Overview partition parallel *TPStream*

a warning. The probability of this case to occur depends on the parameter K which needs to be chosen depending on the (latency) demands of the concrete application scenario.

7.2 Partitioned data

TPStream's query language features a `PARTITION BY` clause (cf. Sect. 4), which allows to divide the input stream into several logical partitions. Those partitions are evaluated independently by applying the *TPStream* operator on each of the partitions. For example, in the aggressive drivers query, each car has to be analyzed separately. This means the input stream has to be partitioned by `car_id`. Therefore, even in the single threaded implementation, *TPStream* maintains a processing pipeline including a dedicated matcher and deriver for every partition. Given this, parallel processing of a partitioned stream is straightforward (cf. Fig. 5): Partitions are assigned to processing threads in a round-robin fashion. Every processing thread has its own input queue, which is fed with batches containing only the relevant events for this particular thread. Therefore, the producer thread maintains a *working-batch* for each of the n processors and assigns incoming events accordingly. Once the *working-batch* reaches the configured batch size, it is put into the corresponding queue and a new *working-batch* is started by the producer. The working threads consume the batches from their assigned queues and process them event-by-event analogous to the single-threaded case. The only difference here is the output-handling: Instead of sending results downstream directly, they go through the K-slack deployed at the end of the pipeline.

To illustrate partition parallel processing, consider the following event trace:

$$< e_1[t = 1, p = 1], e_2[t = 2, p = 2], e_3[t = 3, p = 3], e_4[t = 4, p = 2], \\ e_5[t = 5, p = 4], e_6[t = 6, p = 1], e_7[t = 7, p = 3], e_8[t = 8, p = 4] >$$

It consists of eight events, each with a timestamp t and a partition attribute p. The four logical partitions are handled by two processing threads and a batch-size of four events, resulting in two batches created by the producer: $b_1 = (e_1, e_3, e_6, e_7)$ and $b_2 = (e_2, e_4, e_5, e_8)$. b_1 is assigned to the first and b_2 to the second processing thread. In turn, each of them maintain two *TPStream* instances, one per assigned partition.

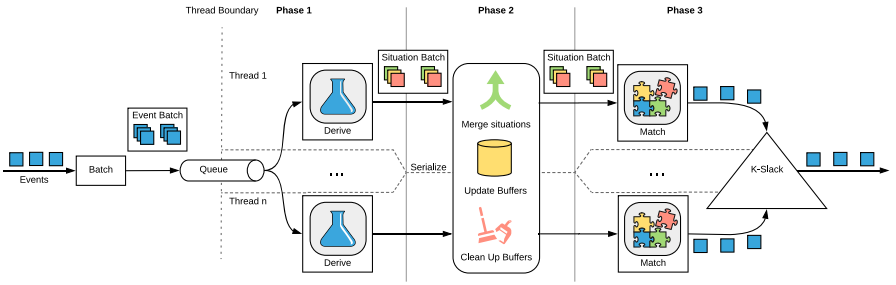

Fig. 6 Overview: parallel processing of unpartitioned data

The processing threads then route the events of their batches to the corresponding partition and process them like in the single threaded case.

7.3 Unpartitioned data

In the following, we consider the case that all processing threads need to work cooperatively on all of the input data. This occurs if there is no `PARTITION BY` clause. Figure 6 gives a brief overview of how unpartitioned data is processed by multiple threads. Similar to partition-based processing, the incoming events are collected into batches of fixed size. The difference here is that there exists one *shared* queue serving all processing threads. Thus, the producer simply has to slice the incoming event stream into fixed size batches by maintaining a single *working-batch* instead of multiple ones. Every processing thread handles one event batch at a time, each passing three phases: deriving, synchronization and matching, whereby the second phase is the only synchronization point between the processor threads.

In the first phase, situations are generated by applying the derivers to the current event batch. Phase two first analyzes the situations derived from the previous batch and merges situations spanning the batch boundary. Then, the situation buffers are updated by inserting new and purging outdated situations. Finally, phase three performs the actual pattern matching. In order to avoid duplicate results across threads, we require that at least one situation of the current batch is part of a successful match (c.f. Sect. 6.2). As in the partitioned case, we employ a K-slack to produce an ordered result stream.

Before detailing each of these phases, we illustrate this method with another example. Consider *TPStream* with two situation definitions (A,B), a batch size of four events, two processing threads (t_1, t_2) and the following event trace of eight events:

$$< e_1[t = 1], \ldots, e_8[t = 8] >$$

The producer generates two batches $b_1 = (e_1, \ldots, e_4)$ and $b_2 = (e_5, \ldots, e_8)$. Let thread t_1 processes b_1 and t_2 processes b_2 in parallel. Let us assume that both of them finish the deriving phase at the same time returning the following situation batches:

$$sb_1 = \{A_1 = [2, 4), B_1 = [4, 5)\}$$

$$sb_2 = \{A_2 = [7, 8), B_2 = [5, 6)\}$$

For the ease of presentation, derived situations are represented by their validity intervals only. At phase two, execution is synchronized, because *TPStream* relies on the temporal ordering in the situation buffers. Hence, t_1 is permitted first because all situations in sb_1 happen before the situations in sb_2. While A_1 can be added to the buffer directly, B_1 is a cross-batch candidate: the last valid timestamp of B_1 is equal the timestamp of e_4—the last event in b_1. Thus, it is possible that the situation continues with the next batch and is put aside for further checking. Then, outdated situations are removed from the buffers and t_1 moves on to the third phase. Next, t_2 enters phase two and adds A_2 to the buffer. It then checks the set-aside B_1 and merges it with B_2 because B_2 starts with the first event of b_2, and hence the situation would span e_4 and e_5 in sequential processing. After buffering the merged situation $B_{1,2} = [4, 6)$ and purging outdated situations, t_2 continues with phase three.

Note that when removing outdated situations from the buffers, care must be taken not to remove situations which might be used by other threads in phase three. We detail how this is done in Sect. 7.3.2. In the following, we describe the three phases in detail and show how the synchronization overhead between processing threads can be kept at a minimum.

7.3.1 Deriving

In the deriving phase, a processing thread fetches an event batch from the shared queue and applies all derivers to the events of that batch (cf. Algorithm 1). The resulting situations are stored in a so-called situation batch. Besides derived situations, a situation batch carries a sequence number, a head and a tail information. The sequence number is required to update the situation buffers (i.e., maintain the temporal ordering of situation streams). It is assigned to every batch during creation. The head and tail information is required to merge situations spanning more than one batch. They contain all ongoing/partial situations after evaluating the first and the last event of the current batch respectively.

7.3.2 Synchronization

After a processing thread creates a situation batch, it enters the synchronization phase. As mentioned above, the execution gets serialized at this point to guarantee the temporal ordering among situation buffers. To enforce the correct execution order among all threads, we use the batch's sequence number: A processing thread holding a batch s may only enter this phase, if the thread responsible for batch $(s - 1)$ moved on to phase three.

Once entered, the following steps are executed in this phase. At first, batch-crossing situations from the previous batch are merged. Algorithm 5 shows the details of situation merging. We loop over all situation definitions and have to consider three different cases: (i) If the tail of the previous batch does not contain a partial situation, nothing

Algorithm 5: MergeSituations

```
   Input: previous:SituationBatch, current:SituationBatch
   Data: D := [(φ, γ, τ)_i]: situation definitions
1  newUpdate ← ∅;
2  foreach i ∈ |D| do
3      if previous.tail[i] = ∅ then
4          continue;
5      else if current.head[i] = ∅ then
6          previous.tail[i].end = current.head.timestamp;
7          newUpdate ← newUpdate ∪ {previous.tail[i]};
8      else
9          merged ← false;
10         foreach update ∈ current.updates do
11             if update[i] ≠ ∅ then
12                 merge(update[i], previous.tail[i]);
13                 merged ← true;
14         if ¬merged then
15             merge(current.tail[i], previous.tail[i]);
16 if newUpdate ≠ ∅ then
17     current.addUpdate(newUpdate);
```

needs to be done (lines 2–4). (ii) If there was a situation ongoing in the previous batch, but we have no information in the head of the current batch, this situation ended with the first event of the current batch. In this case, we only set the correct end timestamp and add it to a new situation update (lines 5–7). At the end of merging, this new update is added to the batch (lines 16, 17). (iii) If we have information in the previous tail *and* the current head, this situation ends either within the current batch or in a future batch. To handle this case, we loop the situation updates of this batch and if we find a matching situation, they are merged, i.e., the aggregates of both situations are merged and the timestamps are adjusted. If no appropriate update is found, the situation covers the whole batch and ends within a future batch. In this case, we transfer the information from the previous tail to the current tail. After the merging is completed, the situations are added to the corresponding buffers in proper order.

Finally, outdated events are purged from the buffers as follows. We compute the minimum timestamp of situations that are relevant (i.e., may participate in a successful match of any active thread) by subtracting the defined window-size from the smallest timestamp found in the current situation batch. Every processing thread maintains an instance of this timestamp and the global minimum across all threads is used for removing outdated situations.

7.3.3 Matching

The matching process works similar to the single threaded version, but instead of processing a single situation update at a time, all updates of a batch are processed. However, there are two challenges here: First, threads must be able to execute the

matching phase while another thread is in the synchronization phase, resulting in concurrent buffer reads and writes. Second, in contrast to the single-threaded version the cleanup process for the buffer has to consider the state of all working threads.

We solve the first challenge with a specialized buffer implementation. As stated in Sect. 6.2, the situation buffers are array-backed ring buffers. These buffers store data in an array of fixed size and manage two indexes. One points to the next write slot (`w_idx`) and the other one to the first element of the buffer (`r_idx`). Instead of relying on the physical addresses in an array that may shrink/grow/loop around, we introduce logical addresses such that add operations increase `w_idx` and clean-up operations also increase `r_idx`. Consequently, the following invariants hold at any time:

1. $\texttt{r\_idx} \leq \texttt{w\_idx}$
2. $\texttt{w\_idx} - \texttt{r\_idx} \leq \texttt{size}$

It is easy to see that all valid data can be found between `r_idx` and `w_idx`, that references remain valid even if the buffer is modified (as long as they still fall into this range) and that the physical array indexes for accessing the situations can be obtained by a simple modulo computation. Processing threads working on a situation batch can now receive a logical address range, thus alleviating problems with concurrent buffer reads and writes. In Fig. 7 we illustrate this with an example: The physical array has 10 addresses, while the logical addresses continue to grow to 28. Each thread has a range of logical addresses it currently works on.

The second challenge is solved by a slightly modified variant of the range-query on the buffers (cf. Sect. 6.2): During the synchronization phase, the processing thread retrieves two index values for every buffer. The first one is the upper bound of relevant situations. It starts at the current `w_idx` when no situation of the batch was added to it and is incremented with each situation of the current batch. The second one is the lower bound of relevant situations which corresponds the timestamp computed for purging outdated situations.

Those indexes are used to restrict the range-queries appropriately and prevent other threads from removing required data from the buffers. To minimize synchronization, each thread records this lower bound at the buffer through the range query at the start

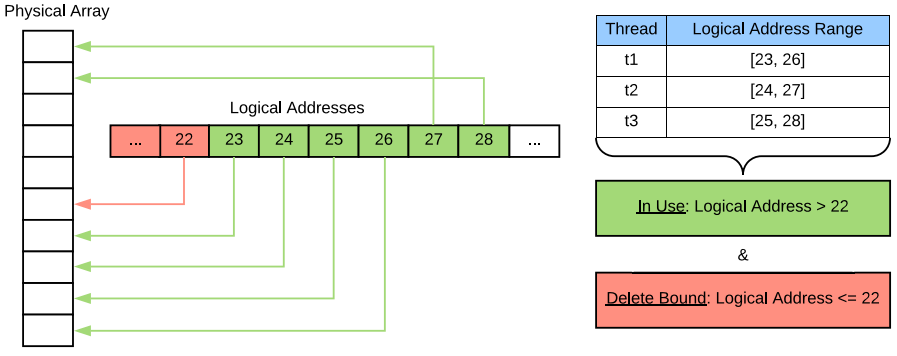

Fig. 7 Example of the buffer implementation

Table 4 Overview of parameters and sensors for the auto-tune component

Name	Description
Parameters	
R_{sched}	The scheduling rate of the auto-tune component
R_{batch}	The target batch rate in batches/second
$Threads_{max}$	The maximum number of threads to use
β	Factor for over-provisioning the number of threads
Sensors	
R_{in}	The input event rate
B_C	The current batch size
$\mathcal{T}_{proc}$	The processing time of a single event batch (of size B_C) without the time required to deque event batches
$\mathcal{T}_{wait}$	The time spent on waiting to enter the synchronization phase (unpartitioned approach only)
$\mathcal{T}_{sync}$	The time spent in the synchronization-phase, without the wait time to enter it (unpartitioned approach only)
$\mathcal{T}_{buf}$	The time spent updating the situation buffers during the synchronization phase (unpartitioned approach only). We have: $\mathcal{T}_{buf} \subseteq \mathcal{T}_{sync}$
Results	
B_T	The target batch-size, i.e. the new batch-size after auto-tuning
$\mathcal{T}^*_{proc}$	The predicted processing time based on B_T
$\mathcal{T}^*_{sync}$	The predicted synchronization time based on B_T
$Threads$	The target number of threads, i.e. the number of threads after auto-tuning

of a batch once. The current processing thread can purge all situations older than the globally lowest bound. For example, in Fig. 7, the lowest address in use by threads t1, t2 and t3 is 23. Thus, everything below or equal to 22 can be purged.

7.4 Auto-tuning

In a streaming scenario with fluctuating data rates, a fixed batch size and a fixed number of processing threads is not an option. When using large values, high throughput can be achieved and peak loads are handled gracefully. However, latency suffers because event batches take some time to be filled. Furthermore, resources are wasted, because the processing threads are starving. On the other hand, small batch sizes and few processing threads generate low latency results in times of moderate event rates, but lead to congestion when the load increases. We developed an auto-tuning component, which continuously monitors the workload and automatically tunes the batch size and number of threads such that the current load can be handled with the lowest latency and lowest resource consumption possible. In the following, we first describe

the approach for partitioned data. Then, we discuss the necessary adjustments to the approach for the unpartitioned case.

Independent of the parallelization approach, the auto-tuning component uses a small set of parameters and sensors, which we briefly describe upfront (a full overview is given in Table 4). Note that all sensors exclude times for thread synchronization, as they are unpredictable and our model relies on predicting processing times. Instead, the model handles them implicitly.

Tuning is scheduled at a fixed rate (R_{sched}). The sensors maintain average values of their respective measures and are reset after every auto-tuning execution. R_{in} is the input event rate, measured in events per second, B_C the currently used batch size and $\mathcal{T}_{proc}$ is the time required to process a batch of B_C events. $Threads_{max}$ is the maximum number of processing threads that may be used for computation and R_{batch} describes the target batch rate. With R_{batch}, we control the number of batches that should be created per second, effectively limiting the synchronization overhead introduced by en- and dequeuing event batches into/out of the queues. With the input rate and the target batch rate, the system is able to compute the target batch size B_T as follows:

$$B_T = \arg\min_{b \in \mathcal{B}} \left(\left| R_{batch} - \frac{R_{in}}{b} \right| \right)$$

That is, we choose B_T from a set of batch sizes ($\mathcal{B}$), such that the resulting batch rate is as close as possible to R_{batch}. We define this set as $\mathcal{B} = \{32, 64, \ldots, 32768\}$. Powers of two have the following advantages: (i) The number of possible values is small, (ii) tiny to huge batches are possible and (iii) it allows fine grained adjustments for smaller batch sizes.

Then, we predict the processing time per batch when using B_T as the batch size and based on this computation the optimal number of threads to use. The predicted processing time ($\mathcal{T}^*_{proc}$) increases linearly with the batch size, because i) deriving situations causes constant costs per event and ii) when assuming that the number of derived situations is increasing linearly with the number of events, the number of matcher invocations also increases linearly. Hence, $\mathcal{T}^*_{proc}$ can be predicted as follows:

$$\mathcal{T}^*_{proc} = \mathcal{T}_{proc} \cdot \frac{B_T}{B_C}$$

We then use $\mathcal{T}^*_{proc}$ to compute the number of threads required:

$$Threads = \min \left(\left\lceil \frac{R_{in}}{B_T} \cdot \mathcal{T}^*_{proc} \cdot \beta \right\rceil, Threads_{max} \right)$$

Essentially, this is the estimated number of batches per second times the estimated processing time per batch, rounded up. We additionally use the parameter β to over-provision the number threads and leave some space for queue operations.

7.4.1 Handling of skewed input

Typically, the number of events to process varies among partitions. Assuming that the number of partitions is much larger than the number of processing threads, the proposed round-robin strategy achieves a good load-balancing between threads in most cases. However, in case of heavily skewed data, this might not be sufficient to balance the load between threads. Consider for example two threads (t_1, t_2) and four partitions with varying event rates ($p_1, \ldots, p_4$). p_1 and p_3 receive 10^6 events/s while p_2 and p_4 receive only 1000 event/s. According to the round-robin strategy, p_1 and p_3 are assigned to t_1, while p_2 and p_3 are assigned to t_2. Obviously, t_2 will be idle most of the time, while t_1 is overloaded.

To solve this problem, the auto-tuning component continuously monitors the current load per thread and migrates partitions from overloaded threads to idle threads. In the following, we describe how the load is monitored and when a migration is triggered. As stated in Sect. 7.2 the producer thread creates batches of events and assigns them to worker threads according to the current partitioning scheme. During this process, additional statistics are collected. Namely, the total number of batches created (n) and for each active thread (t_i) the batches assigned to it (n_{t_i}). With m being the number of active threads an optimally balanced system would yield $n_{t_i} = \frac{n}{m}, \forall i \in [1, m]$. In real world scenarios, an optimal distribution can barely be achieved. However, a distribution within a 10% range around the optimum is achievable leading to the following condition for partition migration:

$$\exists i \in [1, m] : \frac{n}{10 \cdot m} \leq \left| n_{t_i} - \frac{n}{m} \right|$$

Note that, migration is considered only, if the auto-tuning component did not decide to change the number of threads, since this changes the partition assignment anyway.

7.4.2 Adjustments of unpartitioned data

In case of unpartitioned data, the measured processing time $\mathcal{T}_{proc}$ contains the time spent waiting to enter the synchronization phase ($\mathcal{T}_{wait}$), which we need to ignore to obtain a reliable prediction of $\mathcal{T}^*_{proc}$. Furthermore, the computation time of the synchronization phase does not scale linearly with number of processed events for the following reason. While updating the situation buffers ($\mathcal{T}_{buf}$) is linear in the number of events, the time for merging partial situations and cleaning the buffers are constant. This is because their complexity depends on the defined number of situation streams and the window size respectively, and thus are independent of the batch size. That being said, we can compute $\mathcal{T}^*_{proc}$ for unpartitioned data as follows:

$$\mathcal{T}^*_{proc} = \Big(\mathcal{T}_{buf} + \underbrace{\mathcal{T}_{proc} - \left(\mathcal{T}_{wait} + \mathcal{T}_{sync} \right)}_{\text{Time phases 1\&2}} \Big) \cdot \frac{B_T}{B_C} + \underbrace{\left(\mathcal{T}_{sync} - \mathcal{T}_{buf} \right)}_{\text{Sync. phase constant part}}$$

Furthermore, we need to assure that the processing threads are not blocking each other when entering the synchronization phase. To achieve this, we compute the

expected processing time for the synchronization phase $\mathcal{T}^*_{sync}$:

$$\mathcal{T}^*_{sync} = \left(\mathcal{T}_{sync} - \mathcal{T}_{buf}\right) + \mathcal{T}_{buf} \cdot \frac{R_{in}}{B_T}$$

Since every thread needs to execute the synchronization phase once per event batch, $\mathcal{T}^*_{proc}$ should be larger than $(Threads - 1) \cdot \mathcal{T}^*_{sync}$, so that all remaining threads could theoretically execute the synchronization phase while the current thread is processing phases 1 and 3. We experimentally determined that if this ratio is greater than 2, we encounter almost no wait times. Consequently, we scale up the batch size until the following inequation holds:

$$\frac{\mathcal{T}^*_{proc}}{\mathcal{T}^*_{sync}} \geq 2 \cdot (Threads - 1)$$

Since upscaling the batch size does not change the required number of threads to manage the incoming load, the auto-tuning component can apply the computed values without touching the previously computed required number of threads.

7.5 Distributed TPStream

The techniques for partitioned and unpartitioned data can in general also be applied in a distributed computing environment. We base our discussion of these adjustments on the architecture of Apache Kafka,[2] because most streaming applications usually receive their data through some sort of a message queue and perform computation in a shared-nothing cluster. By integrating our work into a full fledged framework like Kafka, we can use its fault tolerance and load distribution features and adjust them towards optimizing *TPStream*.

At its core Apache Kafka is a distributed log allowing both producers to write into and consumers to read from the log. Messages can be sent to a topic and topics are divided into partitions. Each topic partition is stored in a separate commit log on a Kafka broker, which is essentially one process in a cluster designated for managing input and output of a partition. In general, Kafka stores an offset for each consumer, i.e., consumers work and commit their offset to the log independently of each other. To allow multiple consumers to work on a single task in parallel, they have to be configured into a *consumer group*. The unit for parallel computations in Kafka are the partitions of a topic. Therefore, each partition in a topic will be processed by exactly one consumer until a failure or load balancing issues reschedule the partition to another consumer of the same group.

Redistribution of data between consumers, such as exchanging situations between deriving and matching, are performed through additional topics, incurring overhead through writing to additional topic logs and through transporting data over the network. Therefore and due to not being able to rely on the highly optimized ring buffer

[2] https://kafka.apache.org/.

implementation of *TPStream* in all stages of the processing, some adjustments are necessary when applying the techniques described above in a Kafka cluster.

The partitioned data approach naturally translates into a Kafka architecture, because incoming events can be partitioned based on the `PARTITION BY` clause into p partitions. In a cluster with c consumer nodes, the input stream of *TPStream* is configured as a topic with $p_{Kafka} = \min(c, p)$ partitions. If there are more logical partitions then physical processing nodes, we choose the lower number since each partition is a separate log file, thus incurring additional file system overhead. A group of p_{Kafka} Kafka Streams consumer applications executes the *TPStream* operator utilizing exactly once processing semantic. Results of each partition are written into a single, separate result topic monitored by single consumer implementing the K-slack operation.

For unpartitioned data, we adjust the batching and merging phase. We divide the input topic into c topic partitions and distribute event batches to those partitions, effectively writing in a broker's log. This replaces the manual batching and queue logic of our single machine approach. Deriving situation batches in parallel is achieved through a Kafka Streams consumer group which writes the resulting situation batches into a single result topic of unsorted runs. Unlike in a single machine approach, matching is done by one Kafka consumer rather than the same deriving thread, because synchronization of sequence numbers and merging those batches produces a larger overhead in a network environment. Due to this, we can perform a K-slack operation on the sequence numbers before the matching process to further improve processing speed.

So far, our single machine solutions have been mostly data parallel. Due to the modifications for a cluster environment, the strategies can be plugged into a variety of existing research. The second approach in particular uses typical pipeline parallel patterns since the matcher is an independent process waiting for the input of the deriver. This opens the gate to adjust the work for GPU architectures like for traditional pattern matching as described by Cugola and Margara [14]. It is easy to see that due to persisting intermediate results into Kafka topics, it is also straightforward to adapt task parallel solutions of sharing sub-patterns [44], since multiple matchers with shared situation definitions can work with the same Kafka topic.

8 Experimental evaluation

In this section, we present the results from our experimental evaluation of *TPStream*.[3] 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*. After proving the validity of our optimization techniques, we finally evaluate our parallel approaches.

8.1 Setup

All single machine experiments were conducted on a workstation equipped with an AMD Ryzen7 2700X CPU (8 cores, 16 threads) and 16GB of memory, running an

[3] Datasets and source code available at http://uni-marburg.de/oaCPk.

Ubuntu Linux (18.04, kernel version 4.16). 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 start of the measurements.

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 Li et al. [37]. 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[4] 6.0.1) and academia (SASE+[5]), 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 (two 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 [32]—an event processing middleware. We used Oracle JDK 1.8.0_181 to compile the systems and ran all experiments on that JVM with 16 GB of heap space.

During the evaluation two data sources were used. The first source comprises trip data generated with the Linear Road Benchmark [6]. Besides other attributes, each event consists of a unique car id, its location, the current speed and acceleration. We generated data simulating 5 h 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 lasting between 10 and 100 s, while the gaps between two consecutive situations span 10 to 50 s (both uniformly distributed). Events are generated with a frequency of 1 Hz, 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 and except for the parallel section, 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 (seconds) during which a pattern must occur completely.
Event count:	The total number of events to process.

[4] http://www.espertech.com.

[5] https://github.com/haopeng/sase.

8.2 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.

8.2.1 Aggressive drivers

We injected different fractions (1 M to 100 M 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 respectively. 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 Fig. 8 (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. However, *TPStream* reduced the detection latency (time between the start of first situation and the detection time) up to 70% (13% on average) compared to *ISEQ*.

Esper benefits from the simplified version of the pattern, but its evaluation performance is inferior to *TPStream* and *ISEQ*. Esper-1 (first derive then match) performs worse than Esper-2 (low-latency, single pattern) in both cases. *TPStream* and *ISEQ* are about 8 (simplified pattern) and 20 (full pattern) times faster. However, Esper-2 performs well on the simplified pattern and requires only twice the processing time of

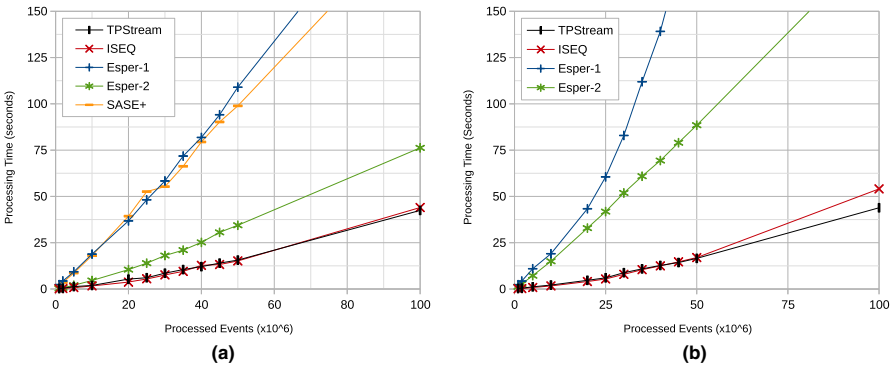

Fig. 8 Processing time for aggressive driver detection as a function of the input size: **a** simplified pattern, **b** full pattern

TPStream and *ISEQ*. When looking at the full pattern its performance drops (factor 4.3), since much more states must be maintained in the automaton. The performance of SASE+ was similar to that of Esper-2.

8.2.2 Disconnected pattern

The second experiment compares processing time and memory consumption of the systems using a more complex pattern : `A starts B before C overlaps D`. The difference to the first experiment is that each `A starts B` sub-match may be related to many `B overlaps C` sub-matches instead of contributing to at most one match. Hence, we expected the processing time/memory consumption to depend on the size of the configured time window. We injected 100 M synthetic events into the systems and executed the query with window sizes varying from 500 s (8:20 min) to 100,000 s (slightly more than 1 day).

Figure 9 shows the processing time of all systems as a function of the window size. In this experiment, *TPStream* is able to outperform *ISEQ* by more than a factor of 3 using a window of 100,000 s. 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+ is not able to manage the many intermediate automaton states efficiently and required approx. 5:20 h to finish this experiment with the largest window size. Esper behaves similarly to the previous experiment, except that Esper-2 was able to outperform *ISEQ* for 100,000 s windows.

To measure the average memory consumption, we monitored the used heap space with a frequency of 20 Hz during each run and averaged these values. Here, *ISEQ* requires by far the least memory 452–465 MB, independent of the window size. *TPStream*'s memory consumption behaves similar for window sizes up to 10,000 (466–507 MB), but increases for larger windows (up to 3.6 GB). This is because we create more objects during the buffer search, which are not immediately garbage collected. Esper uses around 4GB, independent of the window size and SASE+ consumes 3–11 GB of memory.

8.2.3 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. Query-Patterns 1–3 (**Equal, Meets, Chain**) are of the form:

$$Q_n = S_1 \oplus_1 S_2 \wedge S_2 \oplus_2 S_3 \wedge \cdots \wedge S_{n-1} \oplus_{n-1} S_n$$

Here, n denotes the number of situation streams and $\oplus_i$ the *temporal relation* connecting S_i and S_{i+1}. $\oplus_i$ is set to `equals`, `meets` and a randomly drawn *temporal relation* for query patterns 1,2 and 3 respectively. For Query pattern 4 (**Star**), S_1 is connected with every other situation:

$$Q_n = S_1 \oplus_1 S_2 \wedge S_1 \oplus_2 S_3 \wedge \cdots \wedge S_1 \oplus_{n-1} S_n$$

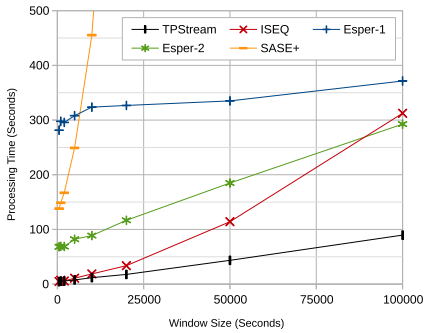

Fig. 9 Processing time for disconnected pattern detection as a function of the window size

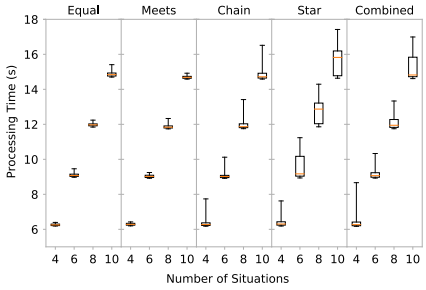

Fig. 10 Processing time for various query patterns

Again, n denotes the number of situation streams and $\oplus_i$ the *temporal relation* connecting S_i and S_{i+1}. Like for the **Chain** pattern $\oplus_i$ is a randomly drawn *temporal relation*. Query 5 (**Combined**) combines the **Chain** and **Star** 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 50 M synthetic events and a window size of 2000 s.

The box plots in Fig. 10 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.

8.3 Low latency

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

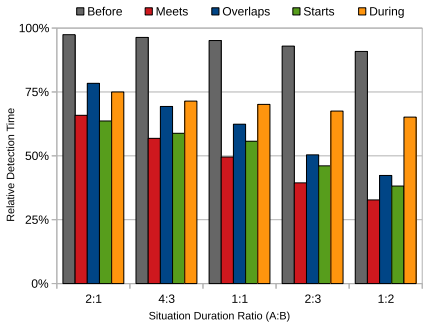

Fig. 11 Relative detection latency per *temporal relation* compared to *ISEQ*

8.3.1 Application time

At first, we measure the latency improvement of *TPStream* compared to *ISEQ* in terms of application time. We evaluated each *temporal relation* independently using two synthetic situation streams (`A`, `B`). The average duration ratio was varied from 2:1 to 1:2, while `A`'s average duration was fixed at 55 s. The window size was set to 1000 s for all application time experiments.

Figure 11 reports the relative detection time of *TPStream* compared to *ISEQ* for the tested *temporal relations* and duration ratios. We define the detection time as $t_d(\mathcal{P}) - \min(A.ts, B.ts)$. That is, the total time from the beginning of the first situation participating in the match until the time of detection. The results show that, independent of the *temporal relation*, with increasing length of `B`-situations the latency gain increases. This is because, the longer `B` becomes, the longer *ISEQ* has to wait for its end timestamp. Among the *temporal relations*, `before` shows the smallest and `meets` the largest latency improvements. However, in absolute numbers, both reduce the detection time by the length of the `B` situation, since in both cases matches are detected at `B.ts`. The difference is, that `meets` allows no gap between the situations which restricts the detection time to the sum of both durations, while `before`-matches can span the entire window (1000 s in this case). 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.

8.3.2 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 Sect. 8.2.2 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 100,000 s and varied the event rate from 1 M 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)

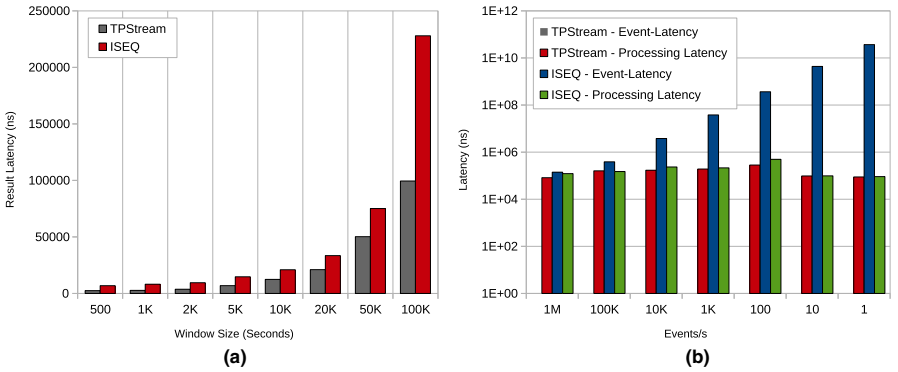

Fig. 12 Comparison of result latency **a** under maximum possible throughput as a function of the window size, **b** under varying event rates with a fixed size window

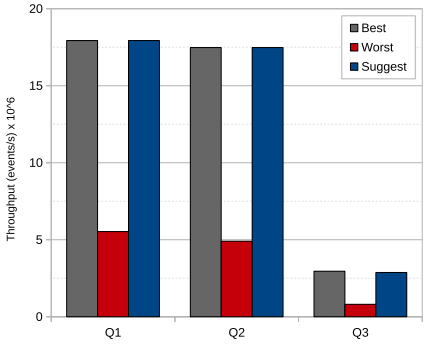

Fig. 13 Quality of the initial plans for **Q1**–**Q3**

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 Fig. 12a, b. Both figures show the average latency per result (y-axis, note the log-scale for b). While (a) shows, that *TPStream*'s evaluation techniques provide latency savings through reduced processing time, (b) 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 (approx. 35 s, cf. Fig. 11, 1:1, `overlaps`), while *TPStream* introduces no event latency at all.

8.4 Plan quality and adaption

In this set of experiments, we evaluate the optimization techniques presented in Sect. 6.4. Like in Sect. 8.2, events were pushed with the maximum possible rate.

8.4.1 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`

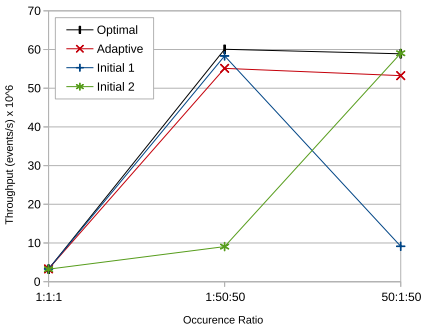

Fig. 14 Throughput comparison: dynamic plan adaption versus best initial plans

`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 six valid plans and measured the throughput (processed events/s) by evaluating synthetic events with a window size of 5000 s.

Figure 13 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.

8.4.2 Dynamic plan adaption

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

Figure 14 shows the throughput for all four configurations and the three different stream-characteristics: Initial-1 and Initial-2 both have drawbacks in either one of the skewed phases, while our adaptive approach is very close to the optimal solution (suffering slightly from dynamic adaption). However, the total runtime of Optimal (33,959 ms) compared to Adaptive (34,106 ms) reveals only a negligible overhead of 147 ms (less than 1%) for plan adaption.

8.5 Parallel approaches

Finally, we evaluate the parallel approaches presented in Sect. 7. We first analyze the speed-up achieved for partitioned and unpartitioned data, before we show the efficiency of our adaptive variants. For each expirement, we used a dedicated producer thread. This thread reads/generates the input events and pushes them into *TPStream*. When processing the data, we varied the number of threads used and measured the

resulting speedup compared to single threaded execution. Finally, we also evaluate our distributed strategies using Apache Kafka.

8.5.1 Partition parallelism

To evaluate our approach for partitioned data, we re-use the query for detecting aggressive drivers and partition it by the `CAR_ID` attribute. We read 50 M events from the event file and push them into *TPStream* as fast as possible. We varied the number of processing threads from 1 to 16, the batch size between 128, 1024, 8192 events and measured the speed up compared to single threaded execution. The size of the input queues was set according to Sect. 7.1, summing up to 2.5 MiB per thread (40 bytes per event). With a K-slack size of 20 events, we did not face out-of-order results in any of the tested configurations. Further, we executed the experiment twice: once with events regularly read from the data file and once with preloading the events into memory, since we expected the disk I/O to become a bottleneck. In both cases, a dedicated producer thread is responsible for ingesting events into *TPStream*. For the first case, this thread is also responsible for the disk I/O.

Figure 15 shows the results for this experiment. The number of processing threads is shown on the x-axis, the speed up on the y-axis. With events read from the data file, *TPStream* scales nicely up to four threads, independent of the batch size (a). For more processing threads, the producer thread becomes the bottleneck of the pipeline, since it has to handle both, the disk I/O and the assignment of event batches to processing threads. Consequently, larger batch sizes perform slightly better, because less batches need to be assigned (i.e., less queue operations are needed). In case of preloaded data (b), our approach scales nicely up to seven processing threads, because the producer is not blocked due to I/O operations and can provide sufficient data to the input queues of the workers. With eight processing threads, we expect a small dip, because the CPU has eight physical cores and we use a dedicated producer thread. Afterwards, hyper-threading comes into play. With more than eight processing threads, the speedup achieved per thread reduces significantly. The batch size has no impact on the process-

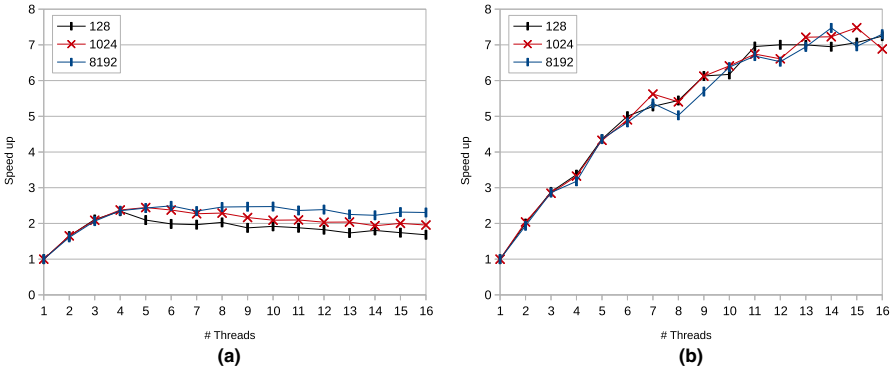

Fig. 15 Speed up compared to single-threaded mode for partition parallel execution of the aggressive drivers query with with different batch sizes and **a** data loaded from file, **b** preloaded data

ing performance here, since the producer is able to provide data sufficiently fast and processing threads are completely independent. Note that due to this independence, the window size also does not affect the achieved speed up.

8.5.2 Unpartitioned data

For this experiment, we reused the query from Sect. 8.2.2, because it's computational complexity increases with the configured window size. We ingested 100 M synthetic events from our event generator using a separate producer thread. The number of processing threads and the batch size was varied from 1 to 16 and between 128, 1024 and 8192 respectively. We executed the query using small (500), medium (5000, 20,000) and large (100,000) windows and measured the speed up compared to single threaded execution. The size of the input queues again was set according to Sect. 7.1, summing up to 2.5 MiB per thread (40 bytes per event). In this experiment, the number of results is much higher than for the aggressive drivers query, because once a combination that satisfies `A starts B` is found, almost all subsequent batches carry at least one match of `C overlaps D`. Hence, they produce results until `A starts B` leaves the window. This means, that the size of the K-slack depends on the batch- and window-sizes as well as on the number of threads. For this experiment, we chose the size as $2 \cdot \#threads \cdot \text{avg}(r_b)$. That is, twice the number of active threads times the average number of results per batch ($\text{avg}(r_b)$), resulting in an out-of-order rate of less than 0.5%. As an example, the size was set to 600 events for a window-size of 100,000, a batch size of 1024 and eight threads.

Figure 16 shows the results for this experiment. In case of the small batch size (128), Fig. 16a–c show a sudden drop at the respective tail end of each experiments. This can be attributed to the batch size being too small, thus incurring congestion at the synchronization phase. For the less complex queries with window sizes 500 and 5000, the producer thread can hardly saturate 4/6 threads. However, they show that different from the partition-based parallel approach, the performance depends on the configured batch size: The lower the computational complexity (i.e., the smaller the window size), the larger is the required batch size for scaling. This is because, with smaller complexity, the time spent in the matching phase (i.e., the time of independent processing) reduces, and thus the possibility of contention in the sync-phase rises. For the more complex queries, our approach is able to take advantage of all available resource and utilizes all available threads.

Furthermore, Fig. 17 shows the relative time spent in the different phases (y-axis) for varying batch-sizes (128, 1024, 8192), processing threads (x-axis) and windows of size 5000 (a) and 20,000 (b). In accordance with Fig. 16, the results confirm that a sufficiently large batch size is essential for scaling. With the batch size being too small (128), the synchronization phase quickly becomes a bottleneck, especially for small windows. Additionally, this experiment showcases that the time spent in the matching phase grows proportional to the configured window size.

8.5.3 Auto-tuning

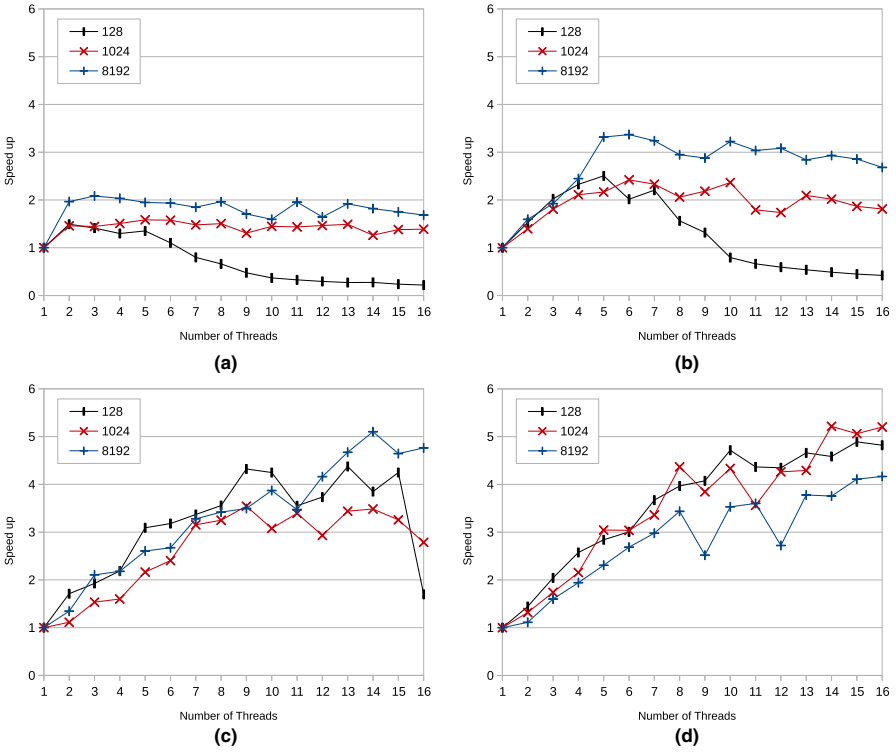

Fig. 16 Processing time for parallel disconnected pattern detection for varying batch and window sizes: **a** 500, **b** 5000, **c** 20,000, **d** 100,000

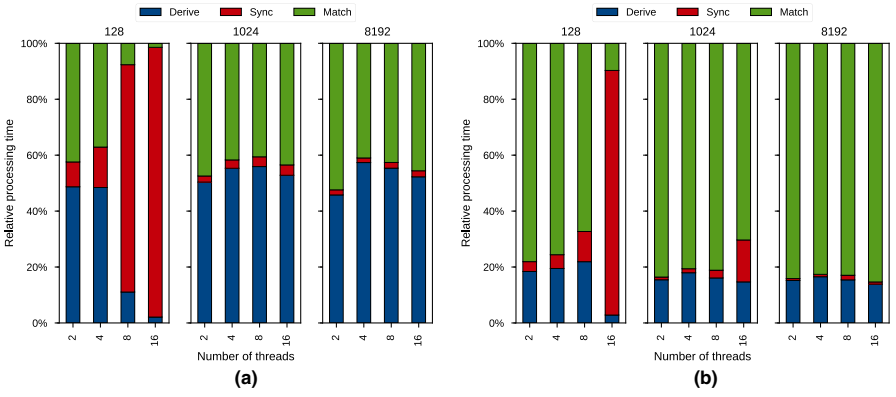

Fig. 17 Relative processing times spent in the different phases of unpartitioned parallel processing for varying batch and window sizes: **a** 5000, **b** 20,000

Table 5 Specification of the queries executed for the auto-tuning experiment

id	Query	Parallel approach	Data	Window	Initial threads	Initial batch size
a	Aggr. drivers	Partitioned	Disk load	5 min	8	256
b	Aggr. drivers	Partitioned	Preloaded	5 min	8	256
c	Disconnected	Unpartitioned	Generated	20,000 s	8	256
d	Disconnected	Unpartitioned	Generated	500 s	8	512

To prove the validity of our auto-tuning approach, we used the 4 scenarios found in Table 5 and generated a changing workload with the following progression: 100 M events at the maximum possible rate, 100 M events at 10 M events/s, 100 M events at 5 M events/s, 30 M events at 1 M events/s, 100 M events at 5 M events/s, 100 M events at 10 M events/s and 100 M events at the maximum possible rate, summing up to 630 M events to process. We additionally executed the queries with batch size and number of processing threads fixed to the initial values given in Table 5.

We configured the auto-tuning component with the following configuration: Adjustments at 1 Hz, T_{batch}: 100,000 batches/s, β: 1.2, $Threads_{max}$: 8. Every time the auto-tuning component changed a parameter, we tracked this change and created the timelines shown in Fig. 18. The x-axis shows the elapsed processing time, highlighting the workload changes. The batch size is aligned on the left and the number of threads on the right y-axis.

As a first result, the processing times of the auto-tuned runs and the runs with fixed parameters were almost identical (varying by less than 1 s), showing that our approach introduces very low overhead. In all four cases the timelines show, that most of the time, the initially configured parameters are way too high, and thus waste resources and introduce unnecessary latency. Furthermore, the adjustments always immediately follow the changes in the workload and we observe only few back-and-forth jumps in the configured parameter, confirming the validity and robustness of our model.

For scenario (a) we see that the configured parameters stay stable until the event rate is throttled to 1 M events/s. This is because when the data is loaded from disk, the maximum event rate we reached was around 3 M events/s. However, during the last period the parameters tend to exceed the previous values and quickly change between 32–64 and 2–4 for the batch-size and number of threads respectively. We attribute this to the page cache of the operating system, which seems to keep a fraction of the data file in memory. For the case of preloaded data (scenario b), the workload changes are clearly reflected by the batch-size and the used number of threads. However, compared to the other measurements, the changes in the number of threads are more noisy. This is because the pattern is highly selective and matcher invocations occur rarely, which in turn causes the average processing time per batch to vary.

In the unpartitioned cases (c, d), we do not encounter such noisy phases. This is because the threads cooperatively process the whole input stream, and if a computation takes more time, this is compensated by another thread. As expected, the utilized

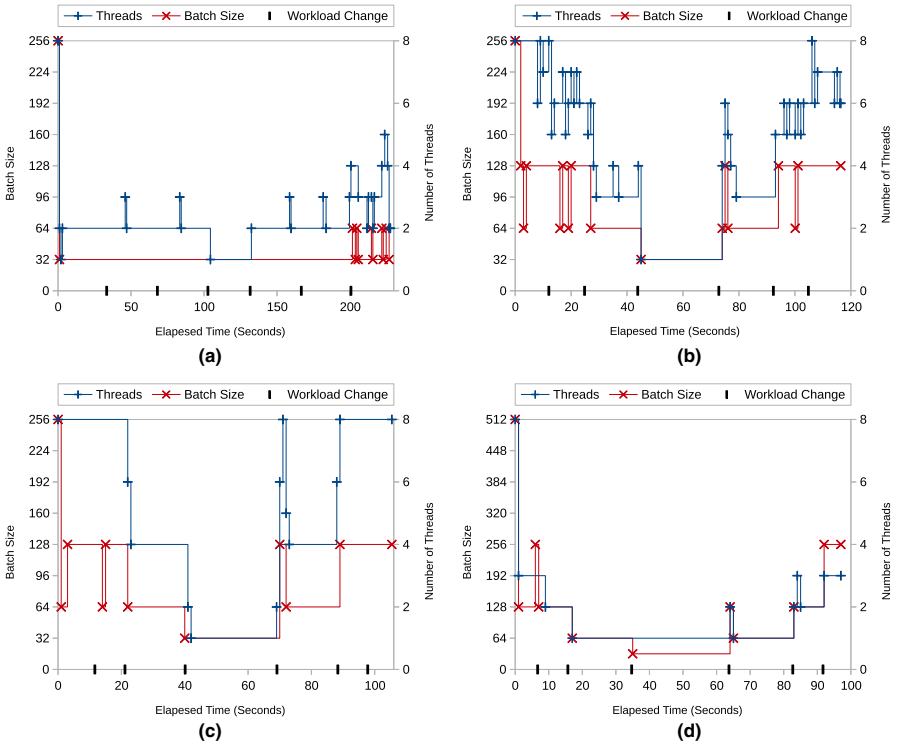

Fig. 18 Evaluation of the configured batch size and number of threads over time: **a** aggressive drivers, data from file, **b** aggressive drivers, data preloaded, **c** disconnected pattern, 20,000 second window, **d** disconnected pattern, 500 s window

resources adapt accordingly to the changes in the workload. For example, slightly after 20 s the event rate halves and so does the used number of threads. However, note that even though the complexity of the 500 s window query is less than of the 20,000 s window, the batch sizes tend to be greater. This can be attributed to the congestion control feature, since the processing time for the 500 s window query is very low and with small batches, the processing threads would spent a reasonable amount of time waiting to enter the synchronization phase.

8.5.4 Distributed environment

For our experiments in a distributed environment, we used a cluster with one coordinator and ten worker nodes, running an Ubuntu Linux (18.04, kernel version 4.15). The hardware specifications can be found in Table 6.

On the cluster we deployed Apache Kafka 2.0.0 using Zookeeper[6] 3.4.5 for coordination. Each worker node hosts one Kafka broker. For experiments, we deployed single-core consumer applications on a variable number of worker nodes. Before each

[6] https://zookeeper.apache.org/.

Table 6 Hardware specification of the cluster

Component	Coordinator	Worker (10 ×)
CPU	Intel Xeon E52640v3 @2,6 GHz	AMD A10-7870K @ 3.9 GHz "Godavari"
GPU	N/A	AMD Radeon R7 (integrated into CPU)
RAM	16GB DDR4 FSB2133	32GB DDR4 FSB2133
Storage	2 × 480 GB SSD; 8 × 8TB HDD	500GB SSD
Network connection	10 Gbit Ethernet	1 Gbit Ethernet

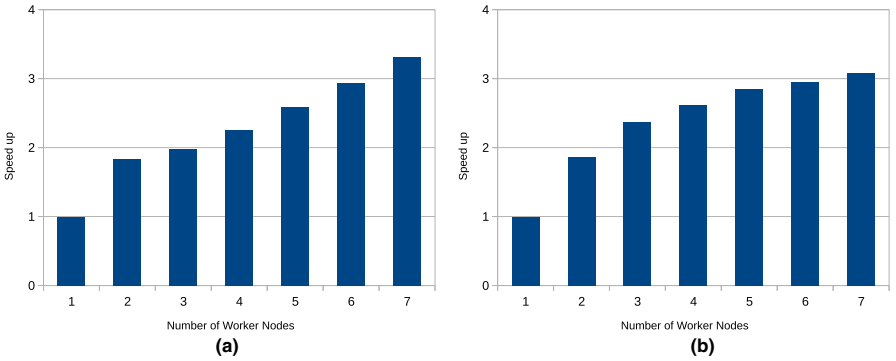

Fig. 19 Speed up in a cluster for: **a** aggressive drivers with 50 M events (partitioned data), **b** disconnected pattern detection with 50 M events (unpartitioned data)

experiment we inserted 10 million input events through a Kafka topic as a warmup during which Kafka balances partition distributions among the registered brokers. Afterwards we measured the speed up relative to a single worker node setup.

For partitioned data, we evaluated the aggressive drivers experiment with 50 M events and the standard Kafka batch size of 10,000 events. We deployed the K-Slack consumer on a single worker node and left the worker node that acts as a controller for all brokers with no workload. On seven remaining worker nodes, we created a varying degree of topic partitions and corresponding consumers. Since the controller handles consumer and partition assignments, a consumer does not, in general, reside on the same node as the partition it is consuming. Figure 19a shows the results of the experiment. Clearly, the approach scales with the used worker nodes since each worker has to read and process only a fraction of the total data.

For unpartitioned data, we again used the query from Sect. 8.2.2 and ingested 60 M events from the generator. On display are results for a batch size of 10,000 events and a moderate window size of 10,000s. In the measurements the processing time includes the time to create event batches from the raw data and the processing time until the last result is generated. Again, we left the controller broker with no workload and used the seven remaining worker nodes for a varying number of partitions. Similar to the partitioned data experiment, the approach scales since it distributes the reading and the

deriving workload. The effects are less prominent than in the partitioned case, since only the deriving is handled in parallel.

It is notable that both experiments take a longer processing time than their respective counterparts on a single machine. This is due to limitations of hardware for each cluster node and due to additional overhead for reading data over the network rather than a file. However, since most streaming environments work with frameworks like Kafka to initially ingest data, achieve fault tolerance and provenance, the overall overhead is likely to occur in practice anyway. Thus, our adaptations show that when used in an environment featuring Kafka, *TPStream* can benefit and use it to its advantage.

9 Conclusion

We presented *TPStream*, a novel event processing operator for detecting complex *temporal patterns* among event streams with low latency and high throughput. By coupling the deriving phase with the matching phase, *TPStream* can detect complex *temporal patterns* at the earliest possible point in time. To handle huge data volumes originating from a variety of sources, we developed parallel and distributed implementations for *TPStream* that can be applied to both partitioned and unpartitioned data streams. In order to maximize resources in a distributed computing environment while reacting to changing data rates of streams, the parallel implementations are tied to a tuning component that automatically sets batch sizes and number of threads. In experiments, we showcased *TPStream*'s performance benefits, scalability and adaptive capabilities while comparing it to current state of the art solutions.

Research on situations in CEP is scarce and *TPStream*'s algebra and implementation provides a first fundamental solution that is equipped to handle a variety of use cases while being compatible with most CEP systems. For future work, we intend to extend *TPStream* to natively handle scenarios such as events arriving out-of-order [13,39]. Furthermore, we intend to look into custom recovery mechanisms beyond what systems like Apache Kafka provide and adapt our parallel implementations for modern co-processors like GPUs.

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

References

1. Abadi, D.J., et al.: The design of the borealis stream processing engine. In: CIDR 2005, pp. 277–289 (2005)
2. Ali, M.H., et al.: Microsoft CEP server and online behavioral targeting. Proc. VLDB Endow. **2**(2), 1558–1561 (2009)
3. Allen, J.F.: Maintaining knowledge about temporal intervals. Commun. ACM **26**(11), 832–843 (1983)
4. Anicic, D., Fodor, P., Rudolph, S., Stojanovic, N.: EP-SPARQL: a unified language for event processing and stream reasoning. In: Proceedings of the 20th International Conference on World Wide Web, WWW 2011, pp. 635–644 (2011)
5. Appelrath, H.J., et al.: Odysseus: a highly customizable framework for creating efficient event stream management systems. In: DEBS'12, pp. 367–368 (2012)

6. Arasu, A., Cherniack, M., Galvez, E., Maier, D., Maskey, A.: Linear road: a stream data management benchmark. In: VLDB'04, pp. 480 – 491 (2004)
7. Avnur, R., Hellerstein, J.M.: Eddies: Continuously adaptive query processing. In: SIGMOD'00, pp. 261–272 (2000)
8. Babu, S., Motwani, R., Munagala, K., Nishizawa, I., Widom, J.: Adaptive ordering of pipelined stream filters. In: SIGMOD'04, ACM, pp. 407–418 (2004)
9. Babu, S., Srivastava, U., Widom, J.: Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. ACM Trans. Database Syst. **29**(3), 545–580 (2004)
10. Balkesen, C., Dindar, N., Wetter, M., Tatbul, N.: RIP: run-based intra-query parallelism for scalable complex event processing. In: DEBS'13, pp. 3–14 (2013)
11. Barbieri, D.F., Braga, D., Ceri, S., Della Valle, E., Grossniklaus, M.: C-sparql: Sparql for continuous querying. In: Proceedings of the 18th International Conference on World Wide Web, WWW '09, pp. 1061–1062 (2009)
12. Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., Tzoumas, K.: Apache flink™: stream and batch processing in a single engine. IEEE Data Eng. Bull. **38**(4), 28–38 (2015)
13. Chandramouli, B., Goldstein, J., Maier, D.: High-performance dynamic pattern matching over disordered streams. Proc. VLDB Endow. **3**(1), 220–231 (2010)
14. Cugola, G., Margara, A.: Low latency complex event processing on parallel hardware. J. Parallel Distrib. Comput. **72**(2), 205–218 (2012)
15. Dayarathna, M., Perera, S.: Recent advancements in event processing. ACM Comput. Surv. **51**(2), 33:1–33:36 (2018)
16. Demers, A.J., Gehrke, J., Panda, B., Riedewald, M., Sharma, V., White, W.M.: Cayuga: a general purpose event monitoring system. In: CIDR'07, pp. 412–422 (2007)
17. Diao, Y., Immerman, N., Gyllstrom, D.: Sase+: an agile language for kleene closure over event streams. Tech. rep., University of Massachusetts (2007)
18. Dignös, A., Böhlen, M.H., Gamper, J.: Overlap interval partition join. In: SIGMOD, 2014, 1459–1470 (2014)
19. Erwig, M.: Toward spatio-temporal patterns. In: Spatio-Temporal Databases: Flexible Querying and Reasoning. Springer, Berlin, pp. 29–53 (2004)
20. Etzion, O., Fournier, F., Skarbovsky, I., von Halle, B.: A model driven approach for event processing applications. In: DEBS'16, pp. 81–92 (2016)
21. Flouris, I., Giatrakos, N., Deligiannakis, A., Garofalakis, M., Kamp, M., Mock, M.: Issues in complex event processing: status and prospects in the Big Data era. J. Syst. Softw. (2016)
22. Gao, D., Jensen, C.S., Snodgrass, R.T., Soo, M.D.: Join operations in temporal databases. VLDB J. **14**(1), 2–29 (2005)
23. Gao, S., Scharrenbach, T., Kietz, J.U., Bernstein, A.: Running out of bindings? Integrating facts and events in linked data stream processing. CEUR Workshop Proc. **1488**, 63–74 (2015)
24. Gao, S., Gu, J., Zaniolo, C.: RDF-TX: a fast, user-friendly system for querying the history of RDF knowledge bases. In: EDBT 2016, pp. 269–280 (2016)
25. Ghanem, T.M., Aref, W.G., Elmagarmid, A.K.: Exploiting predicate-window semantics over data streams. SIGMOD Rec. **35**(1), 3–8 (2006)
26. Golab, L., Özsu, M.T.: Processing sliding window multi-joins in continuous queries over data streams. In: VLDB'03, pp. 500–511 (2003)
27. Golab, L., Özsu, M.T.: Update-pattern-aware modeling and processing of continuous queries. In: SIGMOD'05, pp. 658–669 (2005)
28. Grossniklaus, M., Maier, D., Miller, J., Moorthy, S., Tufte, K.: Frames: data-driven windows. In: DEBS'16, pp. 13–24 (2016)
29. Helmer, S., Persia, F.: Iseql, an interval-based surveillance event query language. IJMDEM **7**(4), 1–21 (2016)
30. Hinze, A., Voisard, A.: Eva: An event algebra supporting complex event specification. Inf. Syst. **48**, 1–25 (2015)
31. Hirzel, M.: Partition and compose: parallel complex event processing. In: DEBS'12, pp. 191–200 (2012)
32. Hoßbach, B., Glombiewski, N., Morgen, A., Ritter, F., Seeger, B.: JEPC: the java event processing connectivity. Datenbank-Spektrum **13**(3), 167–178 (2013)
33. Jayasekara, S., Kannangara, S., Dahanayakage, T., Ranawaka, I., Perera, S., Nanayakkara, V.: Wihidum: distributed complex event processing. J. Parallel Distrib. Comput. 79–80, 42–51 (2015)

34. Kaufmann, M., Manjili, A.A., Vagenas, P., Fischer, P.M., Kossmann, D., Färber, F., May, N.: Timeline index: a unified data structure for processing queries on temporal data in SAP HANA. In: SIGMOD 2013, pp. 1173–1184 (2013)
35. Kietz, J., Scharrenbach, T., Fischer, L., Bernstein, A., Nguyen, K.: Tef-sparql: The ddis query-language for time annotated event and fact triple-streams. Tech. rep., Technical report, University of Zurich, Department of Informatics (2013)
36. Körber, M., Glombiewski, N., Seeger, B.: Tpstream: low-latency temporal pattern matching on event streams. In: EDBT 2018, pp. 313–324 (2018)
37. Li, M., Mani, M., Rundensteiner, E.A., Lin, T.: Complex event pattern detection over streams with interval-based temporal semantics. In: DEBS'11, pp. 291–302 (2011)
38. Lin, Q., Ooi, B.C., Wang, Z., Yu, C.: Scalable distributed stream join processing. In: SIGMOD'15, pp. 811–825 (2015)
39. Liu, M., Li, M., Golovnya, D., Rundensteiner, E.A., Claypool, K.: Sequence pattern query processing over out-of-order event streams. In: ICDE'09, pp. 784–795 (2009)
40. Mei, Y., Madden, S.: ZStream: a cost-based query processor for adaptively detecting composite events. In: SIGMOD'09, pp. 193–206 (2009)
41. Persia, F., Bettini, F., Helmer, S.: An interactive framework for video surveillance event detection and modeling. In: CIKM 2017, pp. 2515–2518 (2017)
42. Piatov, D., Helmer, S., Dignös, A.: An interval join optimized for modern hardware. In: ICDE 2016, pp. 1098–1109 (2016)
43. Poppe, O., Lei, C., Rundensteiner, E.A., Dougherty, D.J.: Context-aware event stream analytics. In: EDBT 2016, pp. 413–424 (2016)
44. Ray, M., Lei, C., Rundensteiner, E.A.: Scalable pattern sharing on event streams. In: SIGMOD'16, pp. 495–510 (2016)
45. Sakr, M.A., Güting, R.H.: Spatiotemporal pattern queries. GeoInformatica **15**(3), 497–540 (2011)
46. Schneider, S., Hirzel, M., Gedik, B., Wu, K.: Auto-parallelizing stateful distributed streaming applications. In: PACT '12, pp. 53–64 (2012)
47. Schultz-Møller, N.P., Migliavacca, M., Pietzuch, P.: Distributed complex event processing with query rewriting. In: DEBS'09, pp. 1–12 (2009)
48. Zaharia, M., et al.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, pp. 15–28 (2012)
49. Zhang, H., Diao, Y., Immerman, N.: On complexity and optimization of expensive queries in complex event processing. In: SIGMOD'14, ACM Press, pp. 217–228 (2014)