SPECTRE: Supporting Consumption Policies in Window-Based Parallel Complex Event Processing

Ruben Mayer
Institute of Parallel and Distributed Systems
University of Stuttgart, Germany
ruben.mayer@ipvs.uni-stuttgart.de

Ahmad Slo
Institute of Parallel and Distributed Systems
University of Stuttgart, Germany
ahmad.slo@ipvs.uni-stuttgart.de

Muhammad Adnan Tariq
Department of Computer Science
FAST - National University of Computer & Emerging Sciences, Islamabad, Pakistan
muhammad.adnan@nu.edu.pk

Kurt Rothermel
Institute of Parallel and Distributed Systems
University of Stuttgart, Germany
kurt.rothermel@ipvs.uni-stuttgart.de

Manuel Gräber
Institute of Parallel and Distributed Systems
University of Stuttgart, Germany
graeber.manuel@gmx.de

Umakishore Ramachandran
College of Computing
Georgia Institute of Technology, USA
rama@cc.gatech.edu

ABSTRACT

Distributed Complex Event Processing (DCEP) is a paradigm to infer the occurrence of complex situations in the surrounding world from basic events like sensor readings. In doing so, DCEP operators detect event patterns on their incoming event streams. To yield high operator throughput, data parallelization frameworks divide the incoming event streams of an operator into overlapping windows that are processed in parallel by a number of operator instances. In doing so, the basic assumption is that the different windows can be processed independently from each other. However, consumption policies enforce that events can only be part of one pattern instance; then, they are consumed, i.e., removed from further pattern detection. That implies that the constituent events of a pattern instance detected in one window are excluded from all other windows as well, which breaks the data parallelism between different windows. In this paper, we tackle this problem by means of speculation: Based on the likelihood of an event's consumption in a window, subsequent windows may speculatively suppress that event. We propose the SPECTRE framework for speculative processing of multiple dependent windows in parallel. Our evaluations show an up to linear scalability of SPECTRE with the number of CPU cores.

CCS CONCEPTS

• **Information systems → Stream management**;

KEYWORDS

Complex Event Processing, Data Parallelization, Event Consumption, Consumption Policy, Speculation

Middleware '17, Las Vegas, NV, USA

978-1-4503-4720-4/17/12…$15.00
DOI: 10.1145/3135974.3135983

ACM Reference format:
Ruben Mayer, Ahmad Slo, Muhammad Adnan Tariq, Kurt Rothermel, Manuel Gräber, and Umakishore Ramachandran. 2017. SPECTRE: Supporting Consumption Policies in Window-Based Parallel Complex Event Processing. In *Proceedings of Middleware '17, Las Vegas, NV, USA, December 11–15, 2017,* 13 pages.
DOI: 10.1145/3135974.3135983

1 INTRODUCTION

Distributed Complex Event Processing (DCEP) [19, 29] is a paradigm applied in many different application areas like logistics, traffic monitoring, and algorithmic trading, to infer the occurrence of complex situations in the surrounding world from basic events like sensor readings or stock quotes. Such situations can be, for instance, the delayed delivery of a packet, traffic jams or accidents and leading market signals. In order to stepwise infer their occurrence from the sensor streams, a distributed network of interconnected DCEP operators, the *operator graph*, is deployed. Each operator processes incoming event streams and detects a designated part of an event pattern that corresponds to a situation of interest. If such a pattern is detected, a new (complex) event is produced and emitted to successor operators or to a consumer, i.e., an entity interested in the corresponding situation. In doing so, operators face increasingly high event loads from their incoming event streams.

In order to be capable of processing high load, the parallelization of DCEP operators has been proposed. In this regard, data parallelization has proven to be a powerful technique to parallelize operators [5, 15, 18, 25–27]. Data-parallel DCEP systems split the incoming event streams into independently processable windows that capture the temporal relations between single events posed by the queried event pattern. The windows are processed in parallel by a number of identical operator instances. An event can be part of different windows, so that windows may overlap.

A crucial question in overlapping windows is whether an event can be used in multiple pattern instances or not. In many cases, it is preferable to *consume* an event once it is part of a pattern instance. In particular, this means to not use the same event for the detection of further pattern instances in other windows. This way, semantic

ambiguities and inconsistencies in the complex events that are emitted can be resolved or prevented. The problem tackled in this paper is that event consumptions impose dependencies between the different windows and thus, prevent their parallel processing. When the same event is processed in parallel in two different windows, consuming it in the first window also consumes it from the second window; hence, there is a dependency between both windows, which can hinder their parallel processing. Understanding that problem, it is no surprise that existing parallel implementations of DCEP systems [5, 13, 25] do not support event consumptions, whereas sequential systems often do [1, 10, 12]. This limits the scalability of operators that impose event consumptions. Moreover, it even impedes event consumptions from their further development in academia and industry, as in times of Big Data and Internet of Things, parallel DCEP systems are becoming the gold standard.

In this paper, we propose a speculative processing method that allows for parallel processing of window-based DCEP operators in case of event consumptions. The basic idea is to speculate in each window which events are consumed in the previous windows—instead of waiting until the previous windows are completely processed. This way, multiple overlapping windows can be processed in parallel despite inter-window dependencies. To this end, we propose the SPECTRE (SPECulaTive Runtime Environment) framework, comprising the following contributions: (1) A speculative processing concept that allows the execution of multiple versions of multiple windows using different event sets in parallel. (2) A probabilistic model to process always those window versions that have the highest probability to be correct. (3) Extensive evaluations that show the scalability with a growing number of CPU cores.

2 BACKGROUND AND PROBLEM ANALYSIS

To solve the problem of parallel event processing in face of event consumptions, we first discuss a common DCEP model in Section 2.1. In Section 2.2, we analyze existing DCEP operator parallelization methods and highlight the properties of *window-based data parallelization* as an expressive and scalable parallelization method [15, 25, 27]. Finally, in Section 2.3 we explain the challenges on parallel processing imposed by event consumptions.

2.1 DCEP Systems

A DCEP system is modeled as an operator graph which interconnects event sources, operators and consumers by event streams. An event e consists of attribute-value pairs containing meta-data, such as event *type*, sequence numbers or timestamps, and the event *payload*, such as sensor readings, stock quotes, etc. Based on the event meta-data, events from different streams arriving at an operator have a well-defined global ordering (e.g., by timestamps and tie-breaker rules). Each operator ω processes events in-order on its incoming streams, detecting event patterns according to a pattern specification. If a pattern instance is detected, the operator emits a (complex) event to its successor in the operator graph.

Event patterns are specified in an *event specification language* such as Snoop [10], Amit [1], SASE [31], or Tesla [11]. Those languages involve operators like event sequences, conjunctions, and negations, in order to define the event patterns to be detected. To express the set of relevant events in pattern detection, the pattern specification imposes a *sliding window* of valid events [3, 15]. This can depend on time or the number of events [11, 12, 31], but also on more complex predicates, e.g., on (combinations of) specific event occurrences that mark the beginning and end of a window [25]. In this paper, we denote the valid window at a specific point in time as w_i. When the window slides, the subsequent valid windows are denoted as w_{i+1}, w_{i+2} etc. Depending on the sliding semantics, different subsequent windows can *overlap*, i.e., events are part of multiple different windows.

Example: In intra-day stock trading, an operator ω receives an event stream containing live stock quote changes of stock A and B throughout the trading day. An analyst wants to detect correlations between a change in A and a change in B. To this end, he formulates a query in the Tesla [11] event specification language:

$[Q_E]$

```
define Influence(Factor)
from B() and
A() within 1min from B
where Factor = B : change / A : change
```

This pattern can be detected by opening a window with a scope of 1 minute whenever an A event occurs; when a B event is detected in a window opened by an A event, a complex event can be created.

Suppose the events A_1, A_2, B_1, B_2 and B_3 occur in the event stream in that order, i.e., A_i denotes the i-th occurrence of an event of type A in the stream (cf. Figure 1). Let us assume that the first A in a window is correlated with every B in the same window—this can be defined in a so-called *selection policy*. As shown in Figure 1a, 5 complex events are detected:[1] $\frac{A_1}{B_1}$, $\frac{A_1}{B_2}$, $\frac{A_2}{B_1}$, $\frac{A_2}{B_2}$, and $\frac{A_2}{B_3}$. Notice, that all events are correlated multiple times, i.e., they are not consumed after building a complex event.

Generally, such multiple correlations of the same event can be problematic. If there is a many-to-one relation between incoming events and detected situations, i.e., many events build a pattern instance but a single event can only be part of one pattern instance, contradicting complex events are produced when events are not consumed. Many-to-one or one-to-one relations are a common case in situation detections.

Therefore, many event specification languages allow for the specification of a *consumption policy* [1, 10, 11, 34]. The consumption policy defines which selected events are consumed after they have participated in a complex event detection: It might be *none*, *all* or *some* of them—e.g., depending on the event type or other parameters. A detailed discussion on consumption policies supported in event specification languages is provided in Section 5. In the example in Figure 1b, selected events of type B are consumed when a complex event is detected, referred to as consumption policy *"selected B"*. Now, only 3 complex events are produced: $\frac{A_1}{B_1}$, $\frac{A_1}{B_2}$, and $\frac{A_2}{B_3}$. In that case, B_1 and B_2 are not re-used after being correlated with A_1 in the first window w_1.

When a complex event is detected, all constituent events of the event pattern are checked against the consumption policy. Then, all events defined by the consumption policy are consumed as a whole. This implies that events are not consumed while they only build a *partial match*, but only when the match is completed and a

[1] $\frac{X}{Y}$ denotes a complex event created from incoming events X and Y.

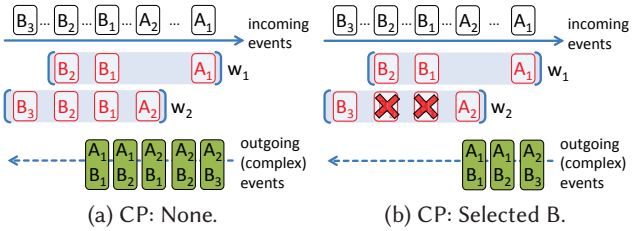

Figure 1: Query Q_E with different consumption policies (CP).

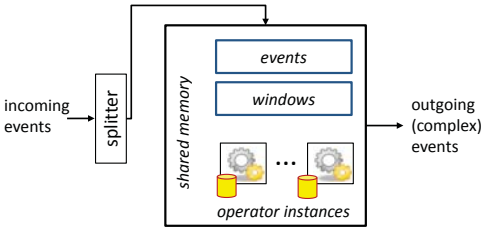

Figure 2: Data parallelization framework.

complex event is produced. This inherent property is independent of the concrete selection and consumption policy.

2.2 Operator Parallelization

The paradigm of *data parallelization* is very powerful in increasing operator throughput. The incoming event stream is split and processed by an elastic number of identical copies of the operator—called *operator instances*. This paradigm has been applied to a wide range of parallel CEP and stream processing systems [6, 9, 15–17, 22, 25–27, 32]. We assume a shared memory (multi-core) architecture, where the splitter and operator instances are executed by independent threads running on dedicated CPU cores (cf. Figure 2). We assume that the underlying system can provide $k+1$ threads, so that 1 thread is pinned to the splitter and k threads are pinned to the operator instances. In the rest of this paper, we do not differentiate between operator instances (i.e., instances of the pattern detection logic) and the threads that execute them—we simply refer to both as operator instances.

As mentioned above, we follow a *window-based* data parallelization approach. The incoming event streams are partitioned into windows that capture (temporal) relations defined in the queried pattern. They can naturally be processed by operator instances, as DCEP operators in their core typically work on a sliding window on the event stream [11, 12, 15, 21, 25, 27, 31]. The windows can be based on time, event count or logical predicates that evaluate whether arbitrary window start and end conditions are fulfilled—a more detailed analysis of window-based data parallelization is provided in [25] and in [15]. For instance, for the time-based window definition of example query Q_E, a new window is opened on each event of type A, whereas an open window is closed after 1 minute based on the events' timestamps. The windows are assigned with increasing window IDs and their boundaries are stored in the shared memory (e.g., "w_i from event X to event Y").

The splitter periodically schedules to each operator instance a specific window for processing. The operator instances can hold local state of the processing in shared memory, e.g., partial pattern matches detected in the assigned window. This allows a specific window to be processed by any operator instance at any time; in particular, the processing of a window can be interrupted for some time and resumed later by the same or a different operator instance.

2.3 Challenges and Goal

In systems without consumptions, processing of a window cannot impact the events within another window, i.e. in principle each pair of windows can be processed in parallel. However, event consumptions impose a dependency between the windows, restricting parallelism, as we discuss in the following.

Recall the example in Figure 1b. The Selection Policy is "first A, each B" and the Consumption Policy is "selected B". In the first window w_1, A_1 and B_1 build a complex event $\frac{A_1}{B_1}$, such that B_1 is consumed; furthermore, A_1 and B_2 build a complex event $\frac{A_1}{B_2}$, such that B_2 is consumed. If w_1 and w_2 are processed in parallel, the consumption of B_1 and B_2 in w_1 might not be known in w_2, so that B_1 and B_2 are erroneously processed in w_2, too, leading to inconsistent results. To prevent anomalies due to concurrent processing, w_2 can only be processed after the consumptions in w_1 are known. When the event patterns are more complex than in the given minimal working examples, the dependencies become hard to control. For instance, if the pattern requires 3 rising stock quotes of B in a sequence, the completion of the pattern in w_1—and hence, the event consumptions—might be unsure until w_1 is completely processed. If 2 events of type B with rising quotes have already been detected in w_1, the completion of the pattern depends on whether a third B occurs; this might only be known at the end of w_1. The standard procedure to deal with data dependencies is to wait with processing w_2 until w_1 is completely processed and hence, all consumptions in w_1 are known. This, however, impedes the parallel processing of overlapping windows.

In this paper, we aim to develop a framework to enable parallel processing of all DCEP operators, regardless of their selection and consumption policy. To this end, we develop a speculative processing method that overcomes the data dependencies imposed by event consumptions, so that data-parallel processing becomes possible. The framework shall deliver exactly those complex events that would be produced in sequential processing; in particular, no false-positive and false-negatives shall occur.

3 THE SPECTRE SYSTEM

To tackle the dependencies between different windows imposed by event consumptions, we propose the SPECTRE (SPECulaTive Runtime Environment) system, a highly parallel framework for DCEP operators. SPECTRE aims to detect the dependencies between different windows and to resolve them by means of speculative execution.

This section is organized as follows. In Section 3.1, we introduce the speculative processing approach we follow in SPECTRE. It is based on creating multiple speculative window versions in order to resolve inter-dependencies between windows. Based on that concept, in Section 3.2, we explain how SPECTRE determines and schedules the k "best" window versions to k operator instances for

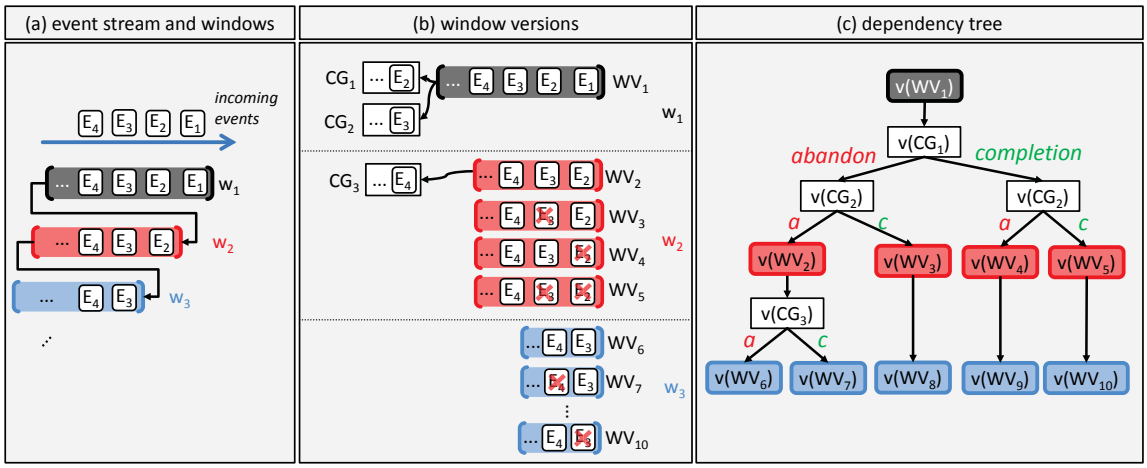

Figure 3: Consumption Problem: (a) Structural View. (b) Processing View. (c) Management View.

parallel processing. Finally, in Section 3.3, we provide details on how the k operator instances perform the parallel processing of the assigned window versions.

3.1 Speculation Approach

As pointed out above, operators process their incoming data stream based on windows. In particular, operators search for queried patterns to occur in the sequence of events comprised by a window. Windows can overlap, i.e. a pair of windows might have a sequence of events in common. The windows of an operator are totally ordered according to their start events. We call a window, say w_j, a successor of another window, w_i, iff the start event of w_i occurs before the starting event of w_j in the corresponding event stream. For example, in Figure 3(a), w_1 starts earlier than w_2; hence, w_2 is a successor of w_1. In the same way, w_3 is a successor both of w_2 and of w_1.

Now, we can define a *consumption dependency* (or *dependency* for short) between windows. Roughly speaking, a window w_j depends on another window w_i, if the consumption of some events in w_i might affect the processing of window w_j. Formally, we define that w_j depends on w_i iff w_j is a successor of w_i and w_j overlaps with w_i. For example, in Figure 3(a), w_2 depends on w_1, and w_3 depends both on w_2 and on w_1.

Now, we will introduce the concept of a *consumption group*. A consumption group is maintained for each partial match of a search pattern found in a window. It records all events of this window that need to be consumed if the partial match becomes a total match, i.e. the corresponding search pattern is eventually detected in the window. Let's assume that an operator is acting on some window w. Whenever the operator processes an event starting a new partial match of some search pattern, it *creates* a new consumption group associated with w. When it processes an event that completes a pattern, it *completes* the corresponding consumption group. On the other hand, a consumption group is *abandoned* if the corresponding pattern cannot be completed anymore. Consequently, while processing the events of a window, multiple consumption groups can be created that are associated with w. However, all of them will be completed or abandoned at the latest when processing of w is finished.

While acting upon w, the operator adds events to be potentially consumed to the consumption groups associated with w, in conformance with the specified consumption policy. When a consumption group is completed, all events contained in this group are consumed together. If the consumption group is abandoned instead, it is just dropped and no events are consumed.

For example, let us assume that a query for pattern of a sequence of three events of type A, B and C in a window of time scope 1 minute, is processed by an operator. Let us further assume the consumption policy is set to consume all participating events in case of a pattern match. When detecting an event of type A, say A_1, in a window, the operator creates a new consumption group. The first event of type B, B_1, is added to the consumption group. If the window ends (i.e., 1 minute has passed) and no event of type C is detected, the consumption group is abandoned and no events are consumed in the window. If an event of type C, say C_1, occurs after B_1 and within the window scope, the consumption group is completed, and all three events participating in the pattern match, A_1, B_1 and C_1, are consumed together.

At the time a consumption group is created that is associated with window w, it is unknown whether the corresponding pattern will eventually be completed in w. Clearly, the outcome of the consumption group (complete or abandon) might affect events of all windows that depend on w. One way to handle this uncertainty is to defer the processing of all depending windows until the consumption group terminates (completed or abandoned). However, in general this amounts to processing all windows sequentially. The approach that we follow in SPECTRE is to generate two window versions for each window depending on w, one version assuming that the consumption group will be completed and the other one assuming the consumption group will be abandoned. These window versions can then be processed in parallel to w. Once the outcome of the consumption group is known, i.e., completed or abandoned, processing continues on the corresponding window versions that assume the correct outcome while the other window versions that assume the wrong outcome are just dropped. Obviously, this approach allows for processing dependent windows in parallel even in the presence of event consumptions.

With this approach, windows that depend on other windows may have multiple versions that depend on the outcome of the associated consumption groups. In principle there is a window version for any combination of the complete and abandon case of the consumption groups that a window depends upon. When one of these consumption group is abandoned, all window versions assuming this consumption group to complete can be dropped, and vice versa.

To capture the dependency between consumption groups and window versions, we introduce the concept of a *dependency tree*. There exists an individual dependency tree for each independent window, i.e., each window that does not depend on any other window according to our definition above. The vertices of the dependency tree are window versions or consumption groups, while the directed edges of the tree specify the dependencies between them. The root of the dependency tree is the only version of an independent window—by definition, there is only one version of an independent window.

The vertex of a window version WV, say v(WV), has at most one child. The sub-hierarchy rooted by this child includes all versions of windows depending on WV, if any. We will denote this sub-hierarchy as v(WV)'s subtree. The subtree is rooted by a consumption group if a consumption group is associated with v(WV). Otherwise the root of the subtree is a window version directly dependent on v(WV), if any.

A vertex representing a consumption group CG, say v(CG), always has two children, one for each possible outcome of CG (completed or abandoned). The so-called completion edge of v(CG) links the subtree of window versions for which completion of CG is assumed, whereas the so-called abandon edge of v(CG) links the subtree of window versions which assume CG to be abandoned. That is, all window versions that can be reached via v(CG)'s completion edge do not include any event included in CG, while events in CG have no effect on window versions linked by v(CG)'s abandon edge.

When a consumption group CG associated with a window version WV is created, the following is performed: v(CG) is added as a new child of v(WV) to the dependency tree. The old subtree of v(WV) is linked by v(CG)'s abandon edge, while a modified copy of the subtree is linked by v(CG)'s completion edge. The modification makes sure that no events included in CG occur in the window versions of the subtree linked by v(CG)'s completion edge. In other words, for each window version existing in v(WV)'s old dependent versions subtree, a copy that *suppresses* all events listed in CG is added. Therefore, each new consumption group associated with v(WV) doubles the window versions in v(WV)'s subtree.

Examples and Algorithms: In the following, a set of examples on the management of the dependency tree is provided along with a formalization of the associated management algorithms. We discuss the following cases: (1) a new dependent window is opened, (2) a new consumption group associated to a window version is created, (3) an existing consumption group is completed or abandoned.

New dependent window. When a new window w_{new} is opened that depends on another window w_x, for every leaf vertex of the dependency tree rooted by the window version of w_x, new window versions are created as child vertices (Figure 4, lines 1–10). For example, in Figure 3, at the start of w_3, new window versions (WV_6 to WV_{10}) of w_3 are created and the corresponding vertices (v(WV_6) to v(WV_{10})) are attached to all leaf nodes of the dependency tree rooted by the window version of w_1. If a leaf vertex is a consumption group CG, two window versions of w_3 are created and attached (a version for completion of CG, and a version for abandoning of CG); if a leaf vertex is a window version, one window version of w_3 is created and attached.

Consumption group created. Recall, that when a consumption group CG associated with a window version WV is created, the old subtree of v(WV) is linked by v(CG)'s abandon edge, while a modified copy of the subtree is linked by v(CG)'s completion edge (Figure 4, lines 12–16). In the example in Figure 3, WV_2 creates CG_3. Then, v(CG_3) is attached as a new child to v(WV_2), and the former child, v(WV_6), becomes the root of the unmodified subtree of v(CG_3). For all window versions in the unmodified subtree of v(CG_3), a new alternative version is created that assumes that CG_3 will be completed. Suppose CG_3 contains event E_4. Then, window version WV_6 (from the unmodified subtree) contains event E_4, whereas the alternative window version WV_7 (from the modified subtree) suppresses event E_4.

Consumption group completed / abandoned. When a consumption group is completed or abandoned, the respective opposite abandon or completion path of that consumption group is removed from the dependency tree. There are two different reasons why a consumption group is abandoned: (1) Due to the termination of the corresponding window version/end of window, or (2) due to a condition from a negation statement being fulfilled. For instance, a pattern specification of a sequence of events of type A and B can define that no event of type C shall occur between the A and B events. If a consumption group is opened with an A event, the occurrence of a C event would trigger the consumption group to be abandoned as the pattern instance cannot be completed any more, even if a B event would occur later. The algorithms for subtree removal are listed in Figure 4, lines 18–26.

Discussion: To be able to process k window versions in parallel we obviously need k operator instances. That means, that typically only a small fraction of all possible window versions can be considered for speculative processing. To be able select the k most promising window versions, we need a method for predicting the probability of possible window versions to survive (i.e., not to be dropped). In Section 3.2, we propose a scheme for scheduling the k most promising window versions on a collection of k operator instances.

3.2 Selecting and Scheduling the Top-k Window Versions

The intuition behind SPECTRE is to predict the k "best" speculative window versions and schedule them for parallel processing on k operator instances. To determine the top-k window versions, SPECTRE periodically determines the k window versions with the highest probability to survive in the entire dependency tree. In other words, SPECTRE does not create and schedule windows, as assumed in Section 2.2, but window versions; in doing so, multiple versions of the same window can be scheduled to different operator instances in parallel.

```
1:  newWindow ( ) begin
2:   for each leafVertex ∈ dependencytree do
3:       if leafVertex is window version then
4:           leafVertex.child ← new v(WV)
5:       else                                  ▷ else, it is a Consumption Group
6:           leafVertex.completionEdge ← new v(WV)
7:           leafVertex.abandonEdge ← new v(WV)
8:       end if
9:   end for
10: end function
11:
12: consumptionGroupCreated (CGroup CG, WinVersion WV) begin
13:   create a modified copy of the subtree attached to v(WV)
14:   v(CG).completionEdge ←v(WV).modifiedSubtree
15:   v(CG).abandonEdge ←v(WV).originalSubtree
16: end function
17:
18: consumptionGroupCompleted (CGroup CG) begin
19:   v(CG).abandonEdge ← null
20:   v(CG).parent.child ←v(CG).completionEdge
21: end function
22:
23: consumptionGroupAbandoned (CGroup CG) begin
24:   v(CG).completionEdge ← null
25:   v(CG).parent.child ←v(CG).abandonEdge
26: end function
```

Figure 4: Algorithms for managing the dependency tree.

Whether or not a window version WV survives depends on the outcome of the preceding consumption groups, i.e. the consumption groups on the path from WV to the root of the dependency tree. In the following, we will denote this path as WV's *root path*. Remember, each vertex representing a consumption group has two outgoing edges, a complete and an abandon edge. We say that the complete or abandon edge of a consumption group, say CG, becomes *valid* when CG is completed or abandoned, respectively. Once one of these edges becomes valid, the other one turns *invalid*. Consequently, WV survives only if all abandon and complete edges on its root path eventually become valid, i.e., WV is dropped if at least one of these edges turn invalid.

The probability of WV to survive depends on the completion probabilities of the consumption groups on WV's root path. The survival probability of WV, denoted as $SP(WV)$ is determined as follows: Let $P(CG)$ be the probability that CG is completed. Moreover, let CG_c and CG_a be the set of consumption groups that contribute a complete and abandon edge to WV's root path, respectively. Then[2], $SP(WV) = \prod_{c \in CG_c} P(c) \times \prod_{c' \in CG_a} (1 - P(c'))$.

3.2.1 Prediction Model. Now, we discuss how we predict the completion probability of a consumption group. Generally, we observe that the probability that a consumption group is completed equals to the probability that the underlying partial match for a search pattern is completed. Our scheme for predicting the completion probability $P(CG)$ of a consumption group CG at a given time takes into account two factors: (1) The inverse degree of completion,

[2]Note that this calculation bases on the assumption that the different consumption groups are completed or abandoned independently from each other. If there are dependencies between different occurrences of a pattern and, hence, between the completion of different consumption groups, this can be incorporated in the probability calculation by using dependent / conditional probabilities. However, for the sake of simplicity of the presentation of technical concepts and algorithms, we use the formula for independent probabilities here.

```
1:  predictCompletionProbability (ConsumptionGroup CG) begin
2:   n ← Splitter.avgWindowSize − posInWindow
3:   if n ≤ 0 then
4:       n ← 1                                 ▷ At least 1 more event expected
5:   end if
6:   T_n ← (1 − (n mod ℓ)/ℓ) * T_⌊n/ℓ⌋*ℓ + (n mod ℓ)/ℓ * T_⌈n/ℓ⌉*ℓ
7:   δ ← CG.completionState
8:   v_0 ← δ-th unit vector
9:   v_n ← T_n * v_0
10:  return v_n[last]
11: end function
```

Figure 5: Calculation of completion probability of a consumption group.

i.e., how many more events are at least required in order to complete the pattern—denoted by δ—and (2) the expected number of events left in the window, denoted by n. If δ is low and many events are still expected to occur in the window, the probability of completion is high. On the other hand, if δ is high and only very few events are still expected in the window, the probability of completion is low. In the following, we describe how the probabilistic model is built and updated at system run-time.

The dynamic process of pattern completion while processing events is modeled as a discrete-time *Markov process*. The state of the Markov process is spanned from δ to 0. For instance, if a pattern instance consists of at least 3 events (e.g., a sequence of 3 events, or a set of 3 events), the state-space has the elements "3", "2", "1" and "0", with "0" representing the state of total pattern completion. Based on statistics monitored at system run-time, a *stochastic matrix* T_1 is built that describes the *transition probabilities* between the states of the Markov process when processing *one* event. To this end, window versions of independent windows gather statistics about the probability of changing from δ_{old} to δ_{new} when an event is processed. The transition probabilities between any pair of δ_{old} and δ_{new} are captured in a matrix T_1^{new}. After ρ new measurements are available, an updated T_1 is computed from the old T_1^{old} and the newly calculated T_1^{new} as $T_1 = (1-\alpha) * T_1^{old} + \alpha * T_1^{new}$ (*exponential smoothing*). $\alpha \in [0, 1]$ is a system parameter to control the impact of recent and of old statistics on T_1.

Now, the probability of state transitions when processing n events can be computed by raising T_1 to the n-th power: $T_n = (T_1)^n$. The initial state is modeled as a row vector $v_0 = (0, ..., 0, 1, 0, ..., 0)$—the δ-th unit vector, where the δ-th position is 1 and all other positions are 0. The probabilities of reaching the different states in n steps can be computed as $v_n = T_n * v_0$. The last entry of v_n, referring to state "0", is the probability to complete the pattern in n steps starting from state v_0.

To reduce the number of matrix multiplications, each time when T_1 is updated, a set of predefined "step sizes" is precomputed, e.g., T_{10}, T_{20}, T_{30}, etc., providing transition probabilities when 10, 20, 30, ... events are processed. If the number of expected events n is in between two precomputed steps, the transition probabilities are linearly interpolated, e.g., $T_{14} = 0.6 * T_{10} + 0.4 * T_{20}$. The step size, denoted as ℓ, is a system parameter.

Figure 5 formalizes the described methods in an algorithm. The expected number of events left in the window, n, is calculated from the average window size monitored in the splitter and the position

```
1:  findTopKVersions (dependencyTree, k) begin
2:    result ← {}                                          ▷ set
3:    candidates ← {dependencyTree.root}                   ▷ priority queue
4:    for i ← 1...k do
5:      tmp ← candidates.pop()
6:      result.append(tmp)
7:      for each M ← tmp.child do
8:        candidates.add(M)
9:      end for
10:   end for
11:   return result
12: end function
```

Figure 6: Top-k window version selection algorithm.

of the last processed event in the window (line 2). The probability matrix T_n is calculated by linear interpolation of precomputed matrices (line 6). δ is obtained directly from *CG* (line 7), and used in order to build v_0 (line 8); v_n is calculated according to the description above (line 9). The resulting completion probability (transition to state " 0" / pattern completed) is returned (line 10).

3.2.2 Scheduling. Here, we describe how SPECTRE periodically selects and schedules the k window versions with the highest survival probability.

Notice that the survival probability of window versions is decreasing in a root-to-leaf direction in the dependency tree, i.e. in a window version's subtree there exist only window versions that have the same or a lower survival probability. Therefore, window versions are already sorted by their survival probability in the dependency tree, so that it already represents a max-heap, which simplifies the selection of the top-k versions substantially. From top to the bottom, window versions are added to the top-k list as detailed in the algorithm in Figure 6. The algorithm works with two data structures: (1) a set storing the resulting top-k versions (line 2), and (2) a priority queue storing candidates for being added to the top-k versions (line 3). The priority queue sorts the contained versions by their probability, highest probability first. Until k versions are found, the highest version from the candidate list is added to the result set (lines 4–6). The children of that version are also added as candidates (lines 7–9). This way, the top-k window versions are determined with only visiting the minimal number of vertices in the dependency tree.

The scheduling algorithm, listed in Figure 7, does not re-schedule window versions that are already scheduled to avoid unnecessary operations and to increase memory and cache locality of operator instances. Hence, the to-be-scheduled versions are determined (lines 7–9). Further, "free" operator instances are determined that will get a new window version scheduled (lines 10–11). Then, every window version that needs to be scheduled is scheduled to one of the free operator instances (lines 14–17).

3.3 Parallel Processing of Window Versions

Here, we describe how operator instances process their assigned window version according to the dependencies in the dependency tree. In particular, we describe how events are processed and suppressed, and how consumption groups are updated when subpatterns are detected in a window version.

```
1:  List⟨OperatorInstance⟩ operatorInstances
2:  Tree dependencyTree
3:  schedule ( ) begin
4:    List⟨WindowVersion⟩ toBeScheduled                    ▷ empty list
5:    List⟨OperatorInstance⟩ freeOperatorInstances ← operatorInstances
6:    List⟨WindowVersion⟩ topkVersions ←
      findTopKVersions(dependencyTree)
7:    for each WindowVersion WV in topkVersions do         ▷ first pass
8:      if not WV.isScheduled() then                       ▷ WV must be scheduled
9:        toBeScheduled.add(WV)
10:     else                                               ▷ the operator instance keeps WV
11:       freeOperatorInstances.remove(WV.getOperatorInstance())
12:     end if
13:   end for
14:   for each WindowVersion WV in toBeScheduled do        ▷ second pass
15:     OperatorInstance OP ← freeOperatorInstances.pop()
16:     OP.scheduledWV ← WV
17:   end for
18: end function
```

Figure 7: Splitter: Scheduling algorithm.

The scheduled window versions are processed in parallel by the associated operator instances. This means, that an operator instance processes or suppresses events according to the dependencies of the window version. In particular, when the root path of the window version meets the completion edge of a consumption group, events in that consumption group are not processed: they are suppressed. Complex events produced when processing a speculative window version are kept buffered until the window version either becomes valid—then, the complex events are emitted—or is dropped—then, the complex events are dropped, too. Further, when an event is processed, updates of the consumption groups can occur (creation, completion or abandoning a consumption group, or adding the event to an existing consumption group). In the following, we detail the underlying algorithms.

Figure 8 lists the algorithm for event processing in the operator instances. In the beginning of a processing cycle, the operator instance checks whether the splitter has scheduled a new window version (lines 7–9). Then, the next event of the currently scheduled window version is processed (lines 11–29). The operator instance checks whether the event is part of any consumption group that shall be suppressed (line 13). If this is the case, the event is suppressed, i.e., its processing is skipped. If the event is not suppressed, it is processed according to the operator logic (line 14). In doing so, there can be four different actions triggered based on feedback the operator logic provides. (1) The processed event can complete one or multiple partial matches: This induces the creation of one or multiple complex events and the completion of the associated consumption groups. In that case, the emitted complex events are buffered, and the dependency tree is updated, calling the *consumptionGroupCompleted* function (cf. Section 3.1). (2) The processed event can lead to the abandoning of consumption groups, either by closing the window, or by invalidating the underlying partial match. In this case, the dependency tree is updated, calling the *consumptionGroupAbandoned* function (cf. Section 3.1). (3) The processed event can lead to the creation of a new consumption group by initiating a new partial match. In this case, the dependency tree is updated, calling the *consumptionGroupCreated* function (cf. Section 3.1). (4) The processed event can become part of one or several

```
1:  WindowVersion currentWV                          ▷ currently processed WV
2:  WindowVersion scheduledWV                        ▷ currently scheduled WV
3:  int i ← 0                                        ▷ processing counter
4:   main () begin
5:    while true do
6:      i ← i + 1
7:      if scheduledWV ≠ currentWV then              ▷ changed WV?
8:        currentWV ← scheduledWV
9:      end if
10:
11:     // process the next event
12:     Event nextEvent ← currentWV.Window.getNextEvent()
13:     if nextEvent not in currentWV.suppressedCGs then
14:       Feedback fb ← process(nextEvent)
15:       if fb: emitted complex event E, completed CG_c then
16:         buffer E
17:         dependencyTree.consumptionGroupCompleted(CG_c)
18:       end if
19:       if fb: abandoned CG_a then
20:         dependencyTree.consumptionGroupAbandoned(CG_a)
21:       end if
22:       if fb: created CG_new then
23:         dependencyTree.
24:           consumptionGroupCreated(CG_new, currentWV)
25:       end if
26:       if fb: added nextEvent to CG then
27:         CG.add(nextEvent)
28:       end if
29:     end if
30:
31:     // consistency check after each i steps
32:     if (i mod consistencyCheckFreq) == 0 then  ▷ consistency check
33:       bool inconsistencyDetected ← false
34:       for CG ∈ currentWV.suppressedCGs do
35:         if CG.version! = CG.lastCheckedVersion then
36:           if currentWV.usedEvents ∩ CG.events ≠ ∅ then
37:             inconsistencyDetected ← true
38:           end if
39:         end if
40:         CG.lastCheckedVersion ← CG.version
41:       end for
42:       if inconsistencyDetected then
43:         rollback currentWV
44:       end if
45:     end if                                       ▷ end of consistency check
46:   end while
47:  end function
```

Figure 8: Operator Instances: Event Processing.

existing partial matches, possibly adding the event to the associated consumption groups. In this case, the affected consumption groups are updated directly without changing the structure of the dependency tree. Note, that in the implementation of SPECTRE, the function calls of the operator instances on the dependency tree are buffered—they are actually executed on the dependency tree in a batch at each new scheduling cycle of the splitter.

The k scheduled window versions are processed concurrently by the k operator instances, without synchronizing the processing progress of the different window versions. This can lead to a situation where an update on an existing consumption group is propagated too late, causing inconsistencies. For instance, when an event is added to a consumption group CG in one window version WV_a after it has been processed in another window version WV_b adjacent to CG's completion edge, an inconsistency can be induced in WV_b (i.e., an event is processed that should be suppressed). To detect such situations, SPECTRE employs periodic *consistency checks*; the underlying algorithm is sketched in lines 31 – 45. For every consumption group to be suppressed in the currently processed window version, the algorithm checks whether an update has occurred since the last consistency check. If this is the case, the algorithm checks whether in the current window version, any event in the updated consumption group has been erroneously processed. If yes, then an inconsistency has been detected: The event should have been suppressed, but has actually been processed. If an inconsistency is detected, the state of the window version is rolled back to the start, i.e., the window version is reprocessed from the start. Instead of reprocessing a window version from the start in case of an inconsistency, it could also be recovered from an intermediate checkpoint. However, when implementing that approach, we realized that the overhead in periodically checkpointing all window versions is much higher than the gain from recovering from checkpoints.

4 EVALUATIONS

In this section, we evaluate the performance of SPECTRE under different real-world and synthetic workloads and varying queries in the setting of an algorithmic trading scenario. We analyze the scalability of SPECTRE with a growing number of operator instances and the overhead involved in speculation and dependency management.

4.1 Experimental Setup

Here, we describe the evaluation platform, the SPECTRE implementation and the datasets and queries used in the evaluations.

Evaluation Platform. We run SPECTRE on a shared memory multi-core machine with 2x10 CPU cores (Intel Xeon E5-2687WV3 3.1 GHz) that support hyper-threading (i.e., 40 hardware threads). The total available memory in the machine is 128 GB and the operating system is CentOS 7.3.

Implementation. SPECTRE is implemented using C++. The pattern detection and window splitting logic of the queries in these evaluations are implemented as a user-defined function (UDF) inside SPECTRE. Further, we provide a client program that reads events from a source file and sends them to SPECTRE over a TCP connection. Our implementation of SPECTRE is open source[3].

Datasets. We employ two different datasets centered around an algorithmic trading scenario.

First, a real-world stock quotes stream originating from the New York Stock Exchange (NYSE). This dataset contains real intra-day quotes of around 3000 stock symbols from NYSE collected over two months from Google Finance[4]; in total, it contains more than 24 million stock quotes. The quotes have a resolution of 1 quote per minute for each stock symbol. We refer to this dataset as the *NYSE Stock Quotes* dataset, denoted as *NYSE*. NYSE represents realistic data for stock market pattern analytics.

Second, we generated a random sequence of 3 million events consisting of 300 different stock symbols; the probability of each

[3] https://github.com/spectreCEP
[4] https://www.google.com/finance

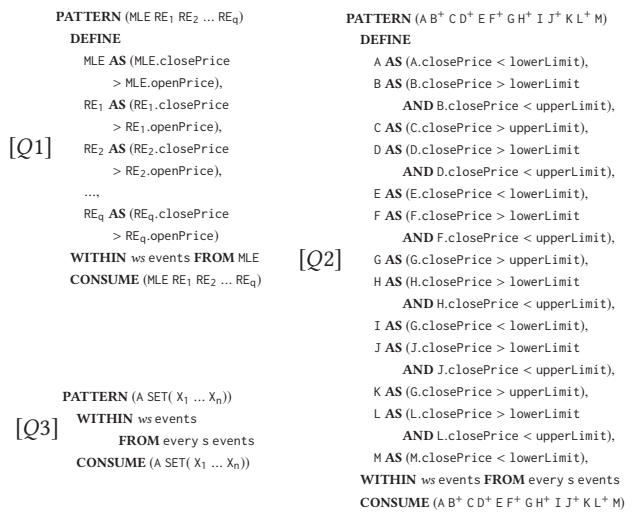

Figure 9: Queries.

stock symbol is equally distributed in the sequence. We refer to this dataset as the *Random Stock Symbols* dataset, denoted as *RAND*.

Queries. We employ three different queries, Q1 to Q3, in the evaluations (cf. Figure 9). The queries are listed in the `MATCH-RECOGNIZE` notation [33], which is concise and easy to understand. Note, that we extended the `MATCH-RECOGNIZE` notation by two additional constructs stemming from the Tesla language [11]: `WITHIN ... FROM` to specify a window size and window start condition, and `CONSUME` to specify consumption policies.

Q1 detects a complex event when the first q rising or the first q falling stock quotes of any stock symbol (defined as `RE` or `FE`, respectively) are detected within ws minutes from a rising or falling quote of a leading stock symbol (defined as `MLE`). The leading stock symbols are composed of a list of 16 technology blue chip companies. In the listing of Q1, we show only the stock rising pattern; the falling pattern is constructed accordingly. In case a complex event is detected, all constituent incoming events are consumed. Note, that this query always has a fixed pattern length of q, and each matching event moves the pattern detection to a higher completion stage.

Q2 is a query from related work (Balkesen and Tatbul [5], Query 9) that we extended by a window size of ws events, a window slide of s events and a consumption policy. It detects a complex event when specific changes occur in the price of a stock symbol between defined *upper* and *lower* limits. As in Q1, all constituent incoming events are consumed when a complex event is detected. We use the *lower* and *upper* limits to control the average pattern size. A small *lower* and a large *upper* limit results in a larger average pattern size, and vice versa. In contrast to Q1, Q2 has a variable length even for a fixed *lower* and *upper* limit. A matching event might or might not influence the pattern completion: the Kleene$^+$ implies that many events can match while the pattern completion does not progress.

Q3 detects a set of n specific stock symbols following stock symbol A. In contrast to the other queries, the ordering of those n symbols is not important. The pattern length n, window size ws, and window slide s can be freely varied. All constituent events are consumed when a complex event is detected.

4.2 Performance Evaluation

In this section, we evaluate the throughput and scalability of SPECTRE. First of all, we evaluate how SPECTRE performs with a growing number of parallel operator instances and with different consumption group completion probabilities. After that, we provide a detailed analysis of the Markov model SPECTRE uses to predict the completion probability of consumption groups. Finally, we discuss a comparison to the CEP engine T-REX [12].

If not noted otherwise, we employ the following settings. The number of created consumption groups is limited to one per window version. The Markov model is employed with the parameters $\alpha = 0.7$ and $\ell = 10$.

To measure the system throughput, we streamed the datasets as fast as possible to the system. Each experiment was repeated 10 times. The figures show the 0th, 25th, 50th, 75th and 100th percentiles of the experiment results in a "candlesticks" representation.

4.2.1 Scalability. Here, we evaluate the scalability of SPECTRE. To this end, we analyze the system throughput, i.e., the number of events processed per second, with a growing number of operator instances. The following questions are addressed: (1) How does the scalability depend on the completion probability of the consumption groups? (2) How much computational and memory overhead is induced by maintaining the dependency tree and determining the top-k window versions?

Effect of Completion Probability of Consumption Groups. We expect that the completion probability of consumption groups influences the system throughput. To make that clear, regard two extreme cases: All consumption groups are abandoned, or all consumption groups are completed. In the first case, SPECTRE should only schedule window versions on the *left-most* path of the dependency tree. In the second case, SPECTRE should only schedule window versions on the *right-most* path of the dependency tree. In both cases, the scheduling algorithm should traverse the dependency tree in *depth*; i.e., it should schedule k window versions from k different windows. Further, none of the scheduled window versions should be dropped; all of them should survive. Hence, the throughput should be maximal. On the other hand, suppose that the completion probability of all consumption groups is constantly at 50 %. In that case, SPECTRE should traverse the dependency tree in *breadth*; i.e., it should schedule 1 window version of the first window, 2 window versions of the second window, 4 window versions of the third window, etc. However, only 1 window version of each window can survive; all others will be dropped. Hence, the higher k is, the more futile processing is performed, as the probability to predict the correct window version drops exponentially with k. In the following, we analyze whether SPECTRE shows the expected behavior and discuss implications.

To this end, we run a set of experiments with queries Q1 and Q2, using the NYSE dataset. In both queries, there are parameters that can be changed such that the average completion probability of consumption groups is manipulated. In Q1, we achieve this by directly setting the pattern size q, such that the ratio between pattern size and window size changes. Larger patterns are less likely to complete. In Q2, we cannot directly set the pattern size. However, we influence the average pattern size—and thus, the average

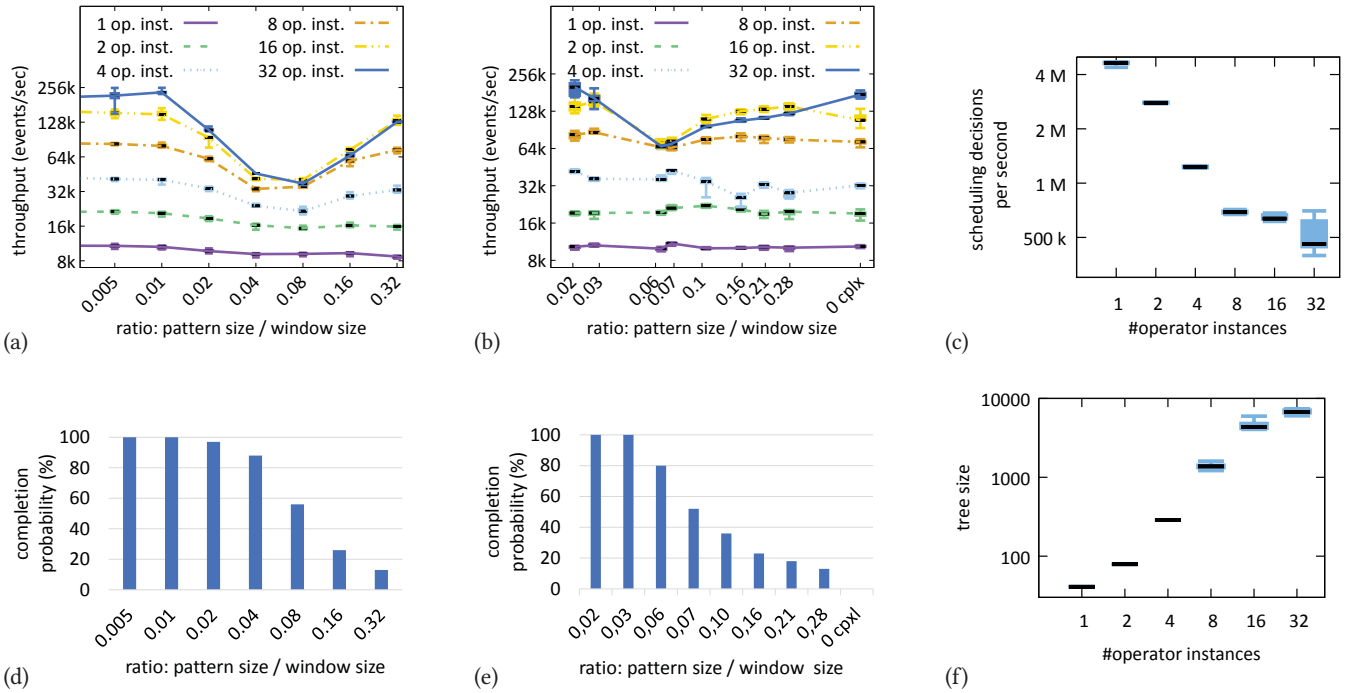

Figure 10: Evaluations. (a)+(d): Scalability (Q1 on NYSE). (b)+(e): Scalability (Q2 on NYSE). (c)+(f): Overhead (Q1 on NYSE).

completion probability—by changing the upper and lower limit parameters in the pattern definition.

In Q1, we employ a sliding window with a window size *ws* of 8,000 events, setting pattern sizes *q* of 40, 80, 160, 320, 640, 1280, and 2560 events. We calculate a "ground truth" value of the completion probability of consumption groups by performing a sequential pass without speculations: The number of created consumption groups divided by the number of produced complex events provides the ground truth value. The system throughput employing 1, 2, 4, 8, 16, and 32 operator instances, is depicted in Figure 10 (a). The corresponding ground truth probabilities are depicted in Figure 10 (d).

At a ratio of pattern size to window size of 40 / 8,000 (i.e., 0.005), the ground truth of consumption group completion probability is at 100 %, i.e., all partial matches are completed. The throughput scales almost linearly with a growing number of operator instances, from 10,800 events/second at 1 operator instance to 154,000 events/second at 16 operator instances (scaling factor 14.3) and 218,000 events/second at 32 operator instances (scaling factor 20.2). Increasing the pattern size decreases the completion probability of consumption groups. At a ratio of pattern size to window size of 640 / 8,000 (i.e., 0.08), the ground truth of consumption group completion probability is at 56 %, i.e., half of partial matches are completed and the other half are abandoned. The throughput scales from 9,200 events/second at 1 operator instance to 35,000 events/second at 8 operator instances (scaling factor 3.8). However, employing more than 8 operator instances does not increase the throughput further: With 16 and 32 operator instances, it is comparable to 8 operator instances. Further increasing the pattern size, we reach a ground truth of consumption group completion probability of 13 % at a ratio of pattern size to window size of 2560 / 8,000 (i.e., 0.32). Here, the throughput scales better, from 8,700 events/second at 1 operator instance to 131,900 events/second at 16 operator instances (scaling factor 15.2). Here, 32 operator instances do not improve the throughput further compared to 16 operator instances.

In Q2, we employ a sliding window with a window size *ws* of 8,000 events and a sliding factor *s* of 1,000 events. We arranged the lower and upper limit parameters in the pattern definition such that the corresponding average pattern sizes were 180, 226, 496, 560, 839, 1261, 1653, and 2223 events, plus one setting that made it impossible for a pattern to be completed. The system throughput employing 1, 2, 4, 8, 16, and 32 operator instances, is depicted in Figure 10 (b). The corresponding ground truth probabilities are depicted in Figure 10 (e).

At a ratio of pattern size to window size of 180 / 8,000 (i.e., 0.02), the ground truth of consumption group completion probability is at 100 %, i.e., all partial matches are completed. The throughput scales almost linearly with a growing number of operator instances, from 10,300 events/second at 1 operator instance to 139,800 events/second at 16 operator instances (scaling factor 13.8) and 200,400 events/second at 32 operator instances (scaling factor 19.5). At a ratio of pattern size to window size of 560 / 8,000 (i.e., 0.07), the ground truth of consumption group completion probability is at 50 %, i.e., half of partial matches are completed and the other half are abandoned. The throughput scales from 10,900 events/second at 1 operator instance to 64,900 events/second at 8 operator instances (scaling factor 6.0). Employing more than 8 operator instances does not increase the throughput further: With 16 and 32 operator instances, it is comparable to 8 operator instances. When none of the partial matches can complete (denoted by "0 cplx"), the throughput

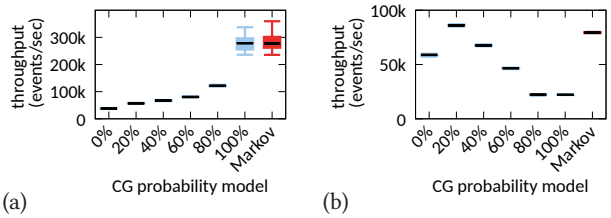

Figure 11: Evaluation of Markov Model.

scales from 10,400 events/second at 1 operator instance to 108,400 events/second at 16 operator instances (scaling factor 10.4) and 174,300 events/second at 32 operator instances (scaling factor 16.8).

Discussion of the results. We draw the following conclusions from the results. First of all, our assumptions on the system behavior are backed by the measurements. Further, the different queries impose "throughput profiles" that have a similar shape. The scaling behavior in SPECTRE, using the speculation approach, is very different from other event processing systems that have been analyzed in related work. In SPECTRE, the parallelization-to-throughput ratio largely depends on the completion probability of partial matches. This new factor leads to interesting implications when adapting the parallelization degree (i.e., elasticity), which is typically done based on event rates [14, 24, 25] or CPU utilization [2, 9]. Existing elasticity mechanisms do not take into account the completion probability to determine the optimal resource provisioning. Using the described throughput curves, SPECTRE could adapt the number of operator instances based on the current pattern completion probability.

Overhead of Speculation. Here, we analyze the computational and memory overhead of maintaining the dependency tree in the splitter and scheduling the top-k window versions.

In a first experiment (Q1, NYSE dataset, q = 80, window size = 8,000), we measure how often the splitter can perform a complete cycle of tree maintenance and top-k scheduling per second. The cycle is described as follows: (a) Maintenance: performing all updates on the dependency tree that have been issued since the last maintenance, i.e., creating new consumption groups and window versions and delete dropped ones, and (b) scheduling: schedule the new top-k window versions to the k operator instances according to the updated dependency tree.

In Figure 10 (c), the results are depicted. With 1 operator instance, SPECTRE achieves a maintenance and scheduling frequency of 4 million cycles per second. With increasing number of operator instances, the scheduling frequency decreases but is still considerably high, where SPECTRE achieves a scheduling frequency of 650, 000 and 450, 000 times per second with 16 and 32 operator instances, respectively. We conclude that there is some overhead involved in the management of the dependency tree and the scheduling algorithm, but there are no indications that this would become a bottleneck in the system.

Another concern about the dependency tree might be its growth and size in memory. To this end, we measured the maximal number of window versions maintained in the dependency tree at the same time (Q1, NYSE dataset, q = 80, window size = 8,000). The results of the experiments are depicted in Figure 10 (f). With 1 operator instance, the maximal tree size was at 41 window versions, growing up to 4,332 at 16 operator instances and 6,730 window versions at 32 operator instances. This is not a serious issue in terms of memory consumption. Indeed, the importance of a suitable top-k window version selection becomes obvious here: Determining the k window versions that will survive out of a large number of window versions that will eventually be dropped is a huge challenge, which SPECTRE could handle reasonably well in the performed experiments.

4.2.2 Markov Model. After we have discussed the overall system throughput and different factors that impact it, we go into a more detailed analysis of the completion probability model of consumption groups. In particular, we want to know how well the proposed Markov model behaves when the probabilities of complex events are changing. To this end, we perform two different experiments of query Q3 with different ratios of pattern size to window size: A ratio of 0.002 that has a high consumption group completion probability and a ratio of 0.1 that has a lower consumption group completion probability. We employed 32 operator instances and the window size ws was set to 1000 events where a new window is opened every 100 events (s = 100). We compare the proposed Markov model with a probability model that assigns each consumption group a fixed completion probability. The results of the two experiments are depicted in Figure 11 (a) and (b), respectively.

At a ratio 0.002, the completion probability of a consumption group was at 100%. Accordingly, assigning a fixed probability of 100% to the consumption groups yielded a throughput of 279,000 events per second, which was significantly better than other fixed probabilities. The Markov model with a throughput of 277,000 events per second proved to be competitive with the best fixed model.

At a ratio of 0.1, the probability of a complex event was at of 32%. Accordingly, assigning a fixed probability of 20% to the consumption groups yielded a throughput of 86,000 events per second, which was significantly better than other fixed probabilities. The Markov model with a throughput of 79,000 events per second performed almost as good as the best fixed model.

From those results, we draw two conclusions. First, the Markov model is able to automatically learn suitable consumption group probabilities in different settings. Second, we can see that wrong probability predictions can cause a large throughput penalty.

4.2.3 Comparison to T-REX. We have also implemented query Q1 in the T-REX event processing engine [12]. In total numbers, T-REX performed much worse than SPECTRE, reaching a throughput of only about 1,000 events per second. While this shows that the throughput of SPECTRE is competitive, it is worth to mention that both systems are different. T-REX is a general-purpose event processing engine that automatically translates queries into state machines, whereas SPECTRE employs user-defined functions to implement queries which allows for more code optimizations. T-REX does not support event consumptions in parallel processing, while SPECTRE can utilize multi-core machines to scale the throughput.

5 RELATED WORK

In the past decades, a number of different Complex Event Processing systems and languages has been proposed. Besides CEP languages that do not support event consumptions, such as SASE [31], the concept of event consumption gained growing importance. Based

on practical use cases, Snoop [10] defined 4 different so-called *parameter contexts*, which are predefined combinations of Selection and Consumption Policies. Building on a more systematic analysis of the problem, Zimmer and Unland [34] proposed an event algebra that differentiated between 5 different Selection and 3 different Consumption Policies that can be combined. Picking up and extending that work, the Amit system [1] allowed for distinct specifications of the Selection and Consumption Policy. Finally, Tesla [11] and its implementation T-REX [12] introduced a formal definition of its supported policies. The proposed speculation methods and the SPECTRE framework are applicable to any combination of selection and consumption policies.

The crucial question in exploiting data parallelism in a DCEP operator is how to split the incoming event streams, such that the different partitions, assigned to different operator instances, can be processed in parallel. Besides window-based splitting, as used in SPECTRE, other splitting methods have been proposed. However, they lack the expressiveness to capture temporal relations between events that many DCEP queries expose.

In key-based splitting [9, 16, 17, 26, 32], the event stream is split by a key that is encoded in the events, e.g., a stock symbol in algorithmic trading [17] or a post ID in social network analysis [26]. Different key value ranges are assigned to different operator instances. However, the parallelism is restricted to the number of different key values; moreover, not all pattern definitions exhibit key-based data parallelism. For instance, in example query Q_E (cf. Section 2.1), events of both stock symbols A and B have to be correlated, so that key-based splitting cannot be applied.

Pane-based splitting has been proposed in stream processing systems [6, 22]. For instance, when the max or median value of a window of 1 minute shall be computed, that window is split into 6 fragments of 10 seconds, the fragments' max or median values are computed in parallel, and the global window's value is aggregated from the fragments' results. This parallel aggregation procedure bases on the idea of pane-based aggregations [23]. However, DCEP patterns often impose a temporal dependency between the events of a window that hinders the vertical splitting, e.g., when a sequence of events A and B is queried as in example query Q_E (cf. Section 2.1). Furthermore, additional constraints on the events can be formulated, e.g., A and B have a parameter x, such that $A : x > B : x$ (e.g., to detect chart patterns in stock markets [17]). If the events are scattered among different vertical windows, such dependencies and constraints cannot be analyzed.

Besides data parallelization, *intra-operator parallelization*, also known as pipelining, has been proposed. Internal processing steps that can be run in parallel are identified by deriving operator states and transitions from the query (e.g., state-based approach in [5]). According to the identified processing steps, the operator logic is split and the processing steps are executed in parallel. This offers only a limited achievable parallelization degree depending on the number of processing steps in the query. For instance, in example query Q_E (cf. Section 2.1), only 2 processing steps, detecting A and detecting B, are available, leading to a maximum parallelization degree of 2. A common variant of intra-operator parallelization uses *lazy evaluation* techniques on event sequence patterns to increase the operator throughput [13, 20]. Those techniques check the event stream for *terminator* events, i.e., the last event of the event sequence in a pattern, and only evaluate preceding events when such a terminator event is found. The underlying assumption is that a terminator event can be determined independently from other events, e.g., solely based on its event type. However, often, sequence patterns depend on the comparison of the events' payload, e.g., a stock quote increasing 3 times in a row; whether a quote is the third in a row that is increasing can only be determined when the two preceding quotes are analyzed. Hence, such techniques are only addressing a subset of possible event patterns.

Speculation has been widely applied to deal with out-of-order events in stream processing. Mutschler and Philippsen [28] propose an adaptive buffering mechanism to sort the events before processing them, introducing a *slack time*. When an event arrives outside of the slack time, results are recomputed. However, slack times cannot be used to overcome window dependencies in the event consumption problem: If one window is processed later, all depending windows would also need to be deferred. Brito et al. [8] as well as Wester et al. [30] propose transaction-based systems to roll-back processing when out-of-order events arrive. Their systems are not parallel, meaning that they only employ one speculation path for each operator. We also roll-back when window versions reach an inconsistent state. However, we propose a highly parallel multi-path speculation method (not only one path) and employ a probabilistic model to schedule the most promising window versions; hence, our system scales with an increasing number of CPU cores. Balazinska et al. [4] propose a system that quickly emits approximate results that are later refined when out-of-order events arrive. Our model would generally allow to be extended toward supporting probabilistic approximations, as a survival probability is given on the window versions. However, in this paper, we focus on consistent event detection (no false-positives, no false-negatives) and leave approximate applications of our model to the future work. Brito et al. [7] propose for non-deterministic stream processing operators to mark events as speculative before logs have been committed to disc for consistent recovery. The speculative events can be forwarded to successor operators in the operator graph that treat them specifically. In SPECTRE, speculative complex events are kept buffered until the window version is confirmed. We focus on providing deterministic event streams to the successor operators; in particular, we do not assume that subsequent operators or event consumers can handle events that are marked as speculative.

6 CONCLUSION

The SPECTRE system uses window-based data parallelization and optimized speculative execution of interdependent windows to scale the throughput of DCEP operators that impose consumption policies. The novel speculation approach employs a probabilistic consumption model that allows for processing the k most promising window versions by k operator instances in parallel on a multi-core machine. Evaluations of the system show good scalability at a moderate overhead for speculation management.

ACKNOWLEDGMENTS

This work was funded by DFG grant RO 1086/19-1 (PRECEPT).

REFERENCES

[1] Asaf Adi and Opher Etzion. 2004. Amit - the Situation Manager. *The VLDB Journal* 13, 2 (May 2004), 177–203. DOI: https://doi.org/10.1007/s00778-003-0108-y

[2] Tyler Akidau, Alex Balikov, Kaya Bekiroglu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. 2013. MillWheel: Fault-tolerant Stream Processing at Internet Scale. *Proc. VLDB Endow.* 6, 11 (Aug. 2013), 1033–1044. DOI: https://doi.org/10.14778/2536222.2536229

[3] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2006. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal* 15, 2 (June 2006), 121–142. DOI: https://doi.org/10.1007/s00778-004-0147-z

[4] Magdalena Balazinska, YongChul Kwon, Nathan Kuchta, and Dennis Lee. 2007. Moirae: History-Enhanced Monitoring.. In *CIDR*. Citeseer, 375–386.

[5] Cagri Balkesen, Nihal Dindar, Matthias Wetter, and Nesime Tatbul. 2013. RIP: Run-based intra-query parallelism for scalable complex event processing *(DEBS '13)*. ACM, 3–14. DOI: https://doi.org/10.1145/2488222.2488257

[6] Cagri Balkesen and Nesime Tatbul. 2011. Scalable data partitioning techniques for parallel sliding window processing over data streams. In *International Workshop on Data Management for Sensor Networks (DMSN)*.

[7] Andrey Brito, Christof Fetzer, and Pascal Felber. 2009. Minimizing Latency in Fault-Tolerant Distributed Stream Processing Systems. In *2009 29th IEEE International Conference on Distributed Computing Systems*. 173–182. DOI: https://doi.org/10.1109/ICDCS.2009.35

[8] Andrey Brito, Christof Fetzer, Heiko Sturzrehm, and Pascal Felber. 2008. Speculative Out-of-order Event Processing with Software Transaction Memory. In *Proceedings of the Second International Conference on Distributed Event-based Systems (DEBS '08)*. ACM, New York, NY, USA, 265–275. DOI: https://doi.org/10.1145/1385989.1386023

[9] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. 2013. Integrating Scale out and Fault Tolerance in Stream Processing Using Operator State Management *(SIGMOD '13)*. ACM, 725–736. DOI: https://doi.org/10.1145/2463676.2465282

[10] Sharma Chakravarthy and Deepak Mishra. 1994. Snoop: An expressive event specification language for active databases. *Data Knowl. Eng.* 14, 1 (1994), 1–26. DOI: https://doi.org/10.1016/0169-023X(94)90006-X

[11] Gianpaolo Cugola and Alessandro Margara. 2010. TESLA: A Formally Defined Event Specification Language. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems (DEBS '10)*. ACM, New York, NY, USA, 50–61. DOI: https://doi.org/10.1145/1827418.1827427

[12] Gianpaolo Cugola and Alessandro Margara. 2012. Complex Event Processing with T-REX. *J. Syst. Softw.* 85, 8 (Aug. 2012), 1709–1728. DOI: https://doi.org/10.1016/j.jss.2012.03.056

[13] Gianpaolo Cugola and Alessandro Margara. 2012. Low latency complex event processing on parallel hardware. *J. Parallel and Distrib. Comput.* 72, 2 (2012), 205 – 218. DOI: https://doi.org/10.1016/j.jpdc.2011.11.002

[14] Tiziano De Matteis and Gabriele Mencagli. 2016. Keep Calm and React with Foresight: Strategies for Low-latency and Energy-efficient Elastic Data Stream Processing. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '16)*. ACM, New York, NY, USA, Article 13, 12 pages. DOI: https://doi.org/10.1145/2851141.2851148

[15] Tiziano De Matteis and Gabriele Mencagli. 2017. Parallel Patterns for Window-Based Stateful Operators on Data Streams: An Algorithmic Skeleton Approach. *International Journal of Parallel Programming* 45, 2 (01 Apr 2017), 382–401. DOI: https://doi.org/10.1007/s10766-016-0413-x

[16] Buğra Gedik. 2014. Partitioning functions for stateful data parallelism in stream processing. *The VLDB Journal* 23, 4 (2014), 517–539. DOI: https://doi.org/10.1007/s00778-013-0335-9

[17] Martin Hirzel. 2012. Partition and Compose: Parallel Complex Event Processing. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems (DEBS '12)*. ACM, New York, NY, USA, 191–200. DOI: https://doi.org/10.1145/2335484.2335506

[18] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. 2014. A Catalog of Stream Processing Optimizations. *ACM Comput. Surv.* 46, 4, Article 46 (March 2014), 34 pages. DOI: https://doi.org/10.1145/2528412

[19] Navendu Jain, Lisa Amini, Henrique Andrade, Richard King, Yoonho Park, Philippe Selo, and Chitra Venkatramani. 2006. Design, Implementation, and Evaluation of the Linear Road Benchmark on the Stream Processing Core *(SIGMOD '06)*. ACM, 431–442. DOI: https://doi.org/10.1145/1142473.1142522

[20] Ilya Kolchinsky, Izchak Sharfman, and Assaf Schuster. 2015. Lazy Evaluation Methods for Detecting Complex Events. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems (DEBS '15)*. ACM, New York, NY, USA, 34–45. DOI: https://doi.org/10.1145/2675743.2771832

[21] Boris Koldehofe, Ruben Mayer, Umakishore Ramachandran, Kurt Rothermel, and Marco Völz. 2013. Rollback-recovery Without Checkpoints in Distributed Event Processing Systems. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems (DEBS '13)*. ACM, New York, NY, USA, 27–38. DOI: https://doi.org/10.1145/2488222.2488259

[22] Alexandros Koliousis, Matthias Weidlich, Raul Castro Fernandez, Alexander L. Wolf, Paolo Costa, and Peter Pietzuch. 2016. SABER: Window-Based Hybrid Stream Processing for Heterogeneous Architectures *(SIGMOD '16)*. ACM, 555–569. DOI: https://doi.org/10.1145/2882903.2882906

[23] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. 2005. No Pane, No Gain: Efficient Evaluation of Sliding-window Aggregates over Data Streams. *SIGMOD Rec.* 34, 1 (March 2005), 39–44.

[24] Björn Lohrmann, Peter Janacik, and Odej Kao. 2015. Elastic Stream Processing with Latency Guarantees. In *2015 IEEE 35th International Conference on Distributed Computing Systems (ICDCS '15)*. 399–410. DOI: https://doi.org/10.1109/ICDCS.2015.48

[25] Ruben Mayer, Boris Koldehofe, and Kurt Rothermel. 2015. Predictable Low-Latency Event Detection with Parallel Complex Event Processing. *Internet of Things Journal, IEEE* 2, 4 (Aug 2015), 274–286.

[26] Ruben Mayer, Christian Mayer, Muhammad Adnan Tariq, and Kurt Rothermel. 2016. GraphCEP: Real-time Data Analytics Using Parallel Complex Event and Graph Processing. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems (DEBS '16)*. ACM, New York, NY, USA, 309–316. DOI: https://doi.org/10.1145/2933267.2933509

[27] Ruben Mayer, Muhammad Adnan Tariq, and Kurt Rothermel. 2017. Minimizing Communication Overhead in Window-Based Parallel Complex Event Processing. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems (DEBS '17)*. ACM, New York, NY, USA, 54–65. DOI: https://doi.org/10.1145/3093742.3093914

[28] Christopher Mutschler and Michael Philippsen. 2014. Adaptive Speculative Processing of Out-of-Order Event Streams. *ACM Trans. Internet Technol.* 14, 1, Article 4 (Aug. 2014), 24 pages. DOI: https://doi.org/10.1145/2633686

[29] Nicholas Poul Schultz-Møller, Matteo Migliavacca, and Peter Pietzuch. 2009. Distributed Complex Event Processing with Query Rewriting *(DEBS '09)*. ACM, Article 4, 12 pages. DOI: https://doi.org/10.1145/1619258.1619264

[30] Benjamin Wester, James Cowling, Edmund B. Nightingale, Peter M. Chen, Jason Flinn, and Barbara Liskov. 2009. Tolerating Latency in Replicated State Machines Through Client Speculation. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)*. USENIX Association, Berkeley, CA, USA, 245–260.

[31] Eugene Wu, Yanlei Diao, and Shariq Rizvi. 2006. High-performance Complex Event Processing over Streams *(SIGMOD '06)*. ACM, 407–418. DOI: https://doi.org/10.1145/1142473.1142520

[32] Erik Zeitler and Tore Risch. 2011. Massive scale-out of expensive continuous queries. *VLDB Endowment* 4, 11 (2011), 1181–1188.

[33] Fred Zemke, Andrew Witkowski, and Mitch Cherniak. 2007. Pattern matching in sequences of rows. (2007).

[34] D. Zimmer and R. Unland. 1999. On the semantics of complex events in active database management systems. In *Data Engineering, 1999. Proceedings., 15th International Conference on*. 392–399. DOI: https://doi.org/10.1109/ICDE.1999.754955